Chapter 6 – Concurrent Programming

Outline
6.1 Introduction
6.2 Monitors
6.2.1 Condition Variables
6.2.2 Simple Resource Allocation with Monitors
6.2.3 Monitor Example: Circular Buffer
6.2.4 Monitor Example: Readers and Writers
6.3 Java Monitors
6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java
6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Objectives

- After reading this chapter, you should understand:
  - how monitors synchronize access to data.
  - how condition variables are used with monitors.
  - solutions for classic problems in concurrent programming such as readers and writers and circular buffer.
  - Java monitors.
  - remote procedure calls.

6.2.1 Introduction

- Recent interest in concurrent programming languages
  - Naturally express solutions to inherently parallel problems
  - Due to proliferation of multiprocessing systems, distributed systems and massively parallel architectures
  - More complex than standard programs
    - More time required to write, test and debug

6.2 Monitors

- Monitor
  - Contains data and procedures needed to allocate shared resources
    - Accessible only within the monitor
    - No way for threads outside monitor to access monitor data

6.2 Monitors

- Resource allocation using monitors
  - Thread must call monitor entry routine
  - Mutual exclusion is rigidly enforced at monitor boundary
  - A thread that tries to enter monitor when it is in use must wait

6.2 Monitors

- Threads return resources through monitors as well
  - Monitor entry routine calls `signal`
    - Alerts one waiting thread to acquire resource and enter monitor
  - Higher priority given to waiting threads than ones newly arrived
    - Avoids indefinite postponement

6.2.1 Condition Variables

- Before a thread can reenter the monitor, the thread calling `signal` must first exit monitor
  - Signal-and-exit monitor
    - Requires thread to exit the monitor immediately upon signaling
- Signal-and-continue monitor
  - Allows thread inside monitor to signal that the monitor will soon become available
  - Still maintain lock on the monitor until thread exits monitor
  - Thread can exit monitor by waiting on a condition variable or by completing execution of code protected by monitor

6.2.2 Simple Resource Allocation with Monitors

- Thread inside monitor may need to wait outside until another thread performs an action inside monitor
- Monitor associates separate condition variable with distinct situation that might cause thread to wait
  - Every condition variable has an associated queue

6.2.2 Simple Resource Allocation with Monitors

Figure 6.1 Simple resource allocation with a monitor in pseudocode.

```
// Fig. 6.1: Resource allocator monitor

// monitor initialization (performed only once)
boolean inUse = false; // simple state variable
Condition available; // condition variable

// request resource
monitorEntry void getResource()
{
   if ( inUse ) // is resource in use?
   {
      wait( available ); // wait until available is signaled
   } // end if

   inUse = true; // indicate resource is now in use

} // end getResource

// return resource
monitorEntry void returnResource()
{
   inUse = false; // indicate resource is not in use
   signal( available ); // signal a waiting thread to proceed

} // end returnResource
```


6.2.3 Monitor Example: Circular Buffer

- Circular buffer implementation of the solution to producer/consumer problem
  – Producer deposits data in successive elements of array
  – Consumer removes the elements in the order in which they were deposited (FIFO)
  – Producer can be several items ahead of consumer
  – If the producer fills last element of array, it must "wrap around" and begin depositing data in the first element of array

6.2.3 Monitor Example: Circular Buffer

- Due to the fixed size of a circular buffer
  - Producer will occasionally find all array elements full, in which case the producer must wait until consumer empties an array element
  - Consumer will occasionally find all array elements empty, in which case the consumer must wait until producer deposits data into an array element

6.2.3 Monitor Example: Circular Buffer

Figure 6.2 Monitor pseudocode implementation of a circular buffer. (Part 1 of 2.)

```
// Fig. 6.2: Circular buffer monitor

char circularBuffer[] = new char[ BUFFER_SIZE ]; // buffer
int writerPosition = 0; // next slot to write to
int readerPosition = 0; // next slot to read from
int occupiedSlots = 0;  // number of slots with data
Condition hasData;      // condition variable
Condition hasSpace;     // condition variable

// monitor entry called by producer to write data
monitorEntry void putChar( char slotData )
{
   // wait on condition variable hasSpace if buffer is full
   if ( occupiedSlots == BUFFER_SIZE )
   {
      wait( hasSpace ); // wait until hasSpace is signaled
   } // end if

   // write character to buffer
   circularBuffer[ writerPosition ] = slotData;
   ++occupiedSlots; // one more slot has data
   writerPosition = (writerPosition + 1) % BUFFER_SIZE;
   signal( hasData ); // signal that data is available
} // end putChar

```

6.2.3 Monitor Example: Circular Buffer

Figure 6.2 Monitor pseudocode implementation of a circular buffer. (Part 2 of 2.)

```
// monitor entry called by consumer to read data
monitorEntry void getChar( outputParameter slotData )
{
   // wait on condition variable hasData if the buffer is empty
   if ( occupiedSlots == 0 )
   {
      wait( hasData ); // wait until hasData is signaled
   } // end if

   // read character from buffer into output parameter slotData
   slotData = circularBuffer[ readPosition ];
   occupiedSlots--; // one fewer slots has data
   readerPosition = (readerPosition + 1) % BUFFER_SIZE;
   signal( hasSpace ); // signal that character has been read
} // end getChar
```


6.2.4 Monitor Example: Readers and Writers

- Readers
  - Read data, but do not change contents of data
  - Multiple readers can access the shared data at once
  - When a new reader calls a monitor entry routine, it is allowed to proceed as long as no thread is writing and no writer thread is waiting to write
  - After the reader finishes reading, it calls a monitor entry routine to signal the next waiting reader to proceed, causing a “chain reaction”
- Writers
  - Can modify data
  - Must have exclusive access

6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 1 of 3.)

```
// Fig. 6.3: Readers/writers problem

int readers = 0; // number of readers
boolean writeLock = false; // true if a writer is writing
Condition canWrite; // condition variable
Condition canRead; // condition variable

// monitor entry called before performing read
monitorEntry void beginRead()
{
   // wait outside monitor if writer is currently writing or if
   // writers are currently waiting to write
   if ( writeLock || queue( canWrite ) )
   {
      wait( canRead ); // wait until reading is allowed
   } // end if

   ++readers; // there is another reader

   signal( canRead ); // allow waiting readers to proceed
} // end beginRead

```


6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 2 of 3.)

```
// monitor entry called after reading
monitorEntry void endRead()
{
   --readers; // there are one fewer readers

   // if no more readers are reading, allow a writer to write
   if ( readers == 0 )
   {
      signal ( canWrite ); // allow a writer to proceed
   } // end if

} // end endRead

// monitor entry called before performing write
monitorEntry void beginWrite()
{
   // wait if readers are reading or if a writer is writing
   if ( readers > 0 || writeLock )
   {
      wait( canWrite ); // wait until writing is allowed
   } // end if

```

6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 3 of 3.)

```
   writeLock = true; // lock out all readers and writers
} // end beginWrite

// monitor entry called after performing write
monitorEntry void endWrite()
{
   writeLock = false; // release lock

   // if a reader is waiting to enter, signal a reader
   if ( queue( canRead ) )
   {
      signal( canRead ); // cascade in waiting readers
   } // end if
   else // signal a writer if no readers are waiting
   {
      signal( canWrite ); // one waiting writer can proceed
   } // end else

} // end endWrite
```


6.3 Java Monitors

- Java monitors
  - Primary mechanism for providing mutual exclusion and synchronization in multithreaded Java applications
  - Signal-and-continue monitors
    - Allow a thread to signal that the monitor will soon become available
    - Maintain a lock on monitor until thread exits monitor
- Keyword `synchronized`
  - Imposes mutual exclusion on an object in Java

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

- Java `wait` method
  - Calling thread releases lock on monitor
    - Waits for condition variable
  - After calling `wait`, thread is placed in wait set
    - Thread remains in wait set until signaled by another thread
- Condition variable is implicit in Java
  - A thread may be signaled, reenter the monitor and find that the condition on which it waited has not been met

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 `SynchronizedBuffer` synchronizes access to a shared integer. (Part 1 of 4.)

```java
// Fig. 6.4: SynchronizedBuffer.java
// SynchronizedBuffer synchronizes access to a shared integer.

public class SynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consumer
   private int occupiedBuffers = 0; // counts occupied buffers

   // place value into buffer
   public synchronized void set( int value )
   {
      // for display, get name of thread that called this method
      String name = Thread.currentThread().getName();

      // while no empty buffers, place thread in waiting state
      while ( occupiedBuffers == 1 )
      {
         // output thread and buffer information, then wait
         try
         {
            System.err.println( name + " tries to write." );
            displayState( "Buffer full. " + name + " waits." );
            wait(); // wait until buffer is empty
         } // end try

```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 `SynchronizedBuffer` synchronizes access to a shared integer. (Part 2 of 4.)

```
            // if waiting thread interrupted, print stack trace
            catch ( InterruptedException exception )
            {
               exception.printStackTrace();
            } // end catch

         } // end while

         buffer = value; // set new buffer value

         // indicate producer cannot store another value
         // until consumer retrieves current buffer value
         ++occupiedBuffers;

         displayState( name + " writes " + buffer );

         notify(); // tell waiting thread to enter ready state
      } // end method set; releases lock on SynchronizedBuffer

```


6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 `SynchronizedBuffer` synchronizes access to a shared integer. (Part 3 of 4.)

```

   // return value from buffer
   public synchronized int get()
   {
      // for display, get name of thread that called this method
      String name = Thread.currentThread().getName();

      // while no data to read, place thread in waiting state
      while ( occupiedBuffers == 0 )
      {
         // output thread and buffer information, then wait
         try
         {
            System.err.println( name + " tries to read." );
            displayState( "Buffer empty. " + name + " waits." );
            wait();// wait until buffer contains new values
         } // end try

         // if waiting thread interrupted, print stack trace
         catch ( InterruptedException exception )
         {
            exception.printStackTrace();
         } // end catch

      } // end while

```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 `SynchronizedBuffer` synchronizes access to a shared integer. (Part 4 of 4.)

```
      // indicate that producer can store another value
      // because consumer just retrieved buffer value
      --occupiedBuffers;

      displayState( name + " reads " + buffer );

      notify(); // tell waiting thread to become ready

      return buffer;
   } // end method get; releases lock on SynchronizedBuffer

   // display current operation and buffer state
   public void displayState( String operation )
   {
      StringBuffer outputLine = new StringBuffer( operation );
      outputLine.setLength( 40 );
      outputLine.append( buffer + "\t\t" + occupiedBuffers );
      System.err.println( outputLine );
      System.err.println();
   } // end method displayState

} // end class SynchronizedBuffer
```


6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 1 of 8.)

```
// Fig. 6.5: SharedBufferTest2.java
// SharedBufferTest2creates producer and consumer threads.

public class SharedBufferTest2
{
   public static void main( String [] args )
   {
      // create shared object used by threads
      SynchronizedBuffer sharedLocation = new SynchronizedBuffer();

      // Display column heads for output
      StringBuffer columnHeads =
         new StringBuffer( "Operation" );
      columnHeads.setLength( 40 );
      columnHeads.append( "Buffer\t\tOccupied Count" );
      System.err.println( columnHeads );
      System.err.println();
      sharedLocation.displayState( "Initial State" );

      // create producer and consumer objects
      Producer producer = new Producer( sharedLocation );
      Consumer consumer = new Consumer( sharedLocation );

      producer.start(); // start producer thread
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 2 of 8.)

```
      consumer.start(); // start consumer thread

   } // end main

} // end class SharedBufferTest2
```


6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 3 of 8.)

Sample Output 1:

```
Operation                                Buffer      Occupied Count

Initial State                              -1              0

Consumer tries to read.
Buffer empty. Consumer waits.              -1              0

Producer writes 1                           1              1

Consumer reads 1                            1              0

Consumer tries to read.
Buffer empty. Consumer waits.               1              0

Producer writes 2                           2              1

Consumer reads 2                            2              0

Producer writes 3                           3              1
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 4 of 8.)

```
Consumer reads 3                      3            0

Consumer tries to read.
Buffer empty. Consumer waits.         3            0

Producer writes 4                     4            1

Consumer reads 4                      4            0
Producer done producing.
Terminating Producer.

Consumer read values totaling: 10.
Terminating Consumer.
```


6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 5 of 8.)

Sample Output 2:

```
Operation                          Buffer      Occupied Count

Initial State                        -1            0

Consumer tries to read.
Buffer empty. Consumer waits.        -1            0

Producer writes 1                     1            1

Consumer reads 1                      1            0

Producer writes 2                     2            1

Producer tries to write.
Buffer full. Producer waits.          2            1

Consumer reads 2                      2            0

Producer writes 3                     3            1
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 6 of 8.)

```
Consumer reads 3                    3            0

Producer writes 4                   4            1

Producer done producing.
Terminating Producer.
Consumer reads 4                    4            0

Consumer read values totaling: 10.
Terminating Consumer.
```

Sample Output 3:

```
Operation                          Buffer       Occupied Count

Initial State                       -1           0

Producer writes 1                   1            1
```


6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 7 of 8.)

Sample Output 3 (Cont.):

```
Operation                          Buffer       Occupied Count

Initial State                       -1           0

Producer writes 1                   1            1

Consumer reads 1                    1            0

Producer writes 2                   2            1

Consumer reads 2                    2            0

Producer writes 3                   3            1

Consumer reads 3                    3            0

Producer writes 4                   4            1

Producer done producing.
Terminating Producer.
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 8 of 8.)

```
Consumer reads 4                        4               0

Consumer read values totaling: 10.
Terminating Consumer.
```


6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

- Threads issue signals via methods `notify` or `notifyAll`
  - `notify` method
    - wakes one thread waiting to enter the monitor
  - `notifyAll` method
    - Wakes all waiting threads

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 `SynchronizedBuffer` controls access to a shared array of integers. (Part 1 of 7.)

```
// Fig. 6.6: CircularBuffer.java
// CircularBuffer synchronizes access to an array of
// shared buffers.

public class CircularBuffer implements Buffer
{
   // each array element is a buffer
   private int buffers[] = { -1, -1, -1 };

   // occupiedBuffers maintains count of occupied buffers
   private int occupiedBuffers = 0;

   // variables that maintain read and write buffer locations
   private int readLocation = 0;
   private int writeLocation = 0;

   // place value into buffer
   public synchronized void set( int value )
   {
      // get name of thread that called this method
      String name = Thread.currentThread().getName();

```


6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 `SynchronizedBuffer` controls access to a shared array of integers. (Part 2 of 7.)

```
      // while buffer full, place thread in waiting state
      while ( occupiedBuffers == buffers.length )
      {
         // output thread and buffer information, then wait
         try
         {
            System.err.println( "\nAll buffers full. " +
               name + " waits." );
            wait(); // wait until space is available
         } // end try

         // if waiting thread interrupted, print stack trace
         catch ( InterruptedException exception )
         {
            exception.printStackTrace();
         } // end catch

      } // end while

      // place value in writeLocation of buffers
      buffers[ writeLocation ] = value;

      // output produced value
      System.err.println( "\n" + name + " writes " +
         buffers[ writeLocation ] + " " );

```

© 2004 Deitel & Associates, Inc. All rights reserved.

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 `SynchronizedBuffer` controls access to a shared array of integers. (Part 3 of 7.)

```
      // indicate that one more buffer is occupied
      ++occupiedBuffers;

      // update writeLocation for future write operation
      writeLocation = ( writeLocation + 1 ) % buffers.length;

      // display contents of shared buffers
      System.err.println( createStateOutput() );

      notify(); // return a waiting thread to ready state
   } // end method set

   // return value from buffer
   public synchronized int get()
   {
      // get name of thread that called this method
      String name = Thread.currentThread().getName();

      // while buffer is empty, place thread in waiting state
      while ( occupiedBuffers == 0 )
      {
```


6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 `SynchronizedBuffer` controls access to a shared array of integers. (Part 4 of 7.)

```
         // output thread and buffer information, then wait
         try
         {
            System.err.println( "\nAll buffers empty. " +
               name + " waits." );
            wait(); // wait until buffer contains new data
         } // end try

         // if waiting thread interrupted, print stack trace
         catch ( InterruptedException exception )
         {
            exception.printStackTrace();
         } // end catch

      } // end while

      // obtain value at current readLocation
      int readValue = buffers[ readLocation ];

      // output consumed value
      System.err.println( "\n" + name + " reads " +
         readValue + " " );

      // decrement occupied buffers value
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 `SynchronizedBuffer` controls access to a shared array of integers. (Part 5 of 7.)

```java
         --occupiedBuffers;

         // update readLocation for future read operation
         readLocation = ( readLocation + 1 ) % buffers.length;

         // display contents of shared buffers
         System.err.println( createStateOutput() );

         notify(); // return a waiting thread to ready state

         return readValue;
      } // end method get

      // create state output
      public String createStateOutput()
      {
         // first line of state information
         String output = "(buffers occupied: " +
            occupiedBuffers + ")\nbuffers: ";

         for ( int i = 0; i < buffers.length; ++i )
         {
            output += " " + buffers[ i ] + "  ";
         } // end for
```


6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 `SynchronizedBuffer` controls access to a shared array of integers. (Part 6 of 7.)

```java

         // second line of state information
         output += "\n          ";

         for ( int i = 0; i < buffers.length; ++i )
         {
            output += "---- ";
         } // end for

         // third line of state information
         output += "\n          ";

         // append readLocation (R) and writeLocation (W)
         // indicators below appropriate buffer locations
         for ( int i = 0; i < buffers.length; ++i )
         {
            if ( i == writeLocation &&
                 writeLocation == readLocation )
            {
               output += " WR  ";
            } // end if
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 `SynchronizedBuffer` controls access to a shared array of integers. (Part 7 of 7.)

```
            else if ( i == writeLocation )
            {
               output += " W   ";
            } // end if
            else if ( i == readLocation )
            {
               output += "  R  ";
            } // end if
            else
            {
               output += "     ";
            } // end else

         } // end for

         output += "\n";

         return output;
      } // end method createStateOutput

   } // end class CircularBuffer
```


6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 1 of 9.)

```
// Fig. 6.7: CircularBufferTest.java
// CircularBufferTest shows two threads manipulating a
// circular buffer.

// set up the producer and consumer threads and start them
public class CircularBufferTest
{
   public static void main ( String args[] )
   {
      // create shared object for threads; use a reference
      // to a CircularBuffer rather than a Buffer reference
      // to invoke CircularBuffer method createStateOutput
      CircularBuffer sharedLocation = new CircularBuffer();

      // display initial state of buffers in CircularBuffer
      System.err.println( sharedLocation.createStateOutput() );

      // set up threads
      Producer producer = new Producer( sharedLocation );
      Consumer consumer = new Consumer( sharedLocation );

```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 2 of 9.)

```
      producer.start(); // start producer thread
      consumer.start(); // start consumer thread
   } // end main

} // end class CircularBufferTest
```

Sample Output:

```
(buffers occupied: 0)
buffers:  -1   -1   -1
          ---- ---- ----
           WR


All buffers empty. Consumer waits.

Producer writes 11
(buffers occupied: 1)
buffers:  11   -1   -1
          ---- ---- ----
            R    W
```


6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 3 of 9.)

```
Consumer reads 11
(buffers occupied: 0)
buffers:  11   -1   -1
          ---- ---- ----
                WR

Producer writes 12
(buffers occupied: 1)
buffers:  11   12   -1
          ---- ---- ----
                R    W


Producer writes 13
(buffers occupied: 2)
buffers:  11   12   13
          ---- ---- ----
           W    R
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 4 of 9.)

Sample Output (Cont.):

```
Consumer reads 12
(buffers occupied: 1)
buffers:  11   12   13
          ---- ---- ----
           W          R


Producer writes 14
(buffers occupied: 2)
buffers:  14   12   13
          ---- ---- ----
                W     R


Producer writes 15
(buffers occupied: 3)
buffers:  14   15   13
          ---- ---- ----
                     WR
```


6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 5 of 9.)

```
All buffers full. Producer waits.

Consumer reads 13
(buffers occupied: 2)
buffers:  14   15   13
          ---- ---- ----
           R         W


Producer writes 16
(buffers occupied: 3)
buffers:  14   15   16
          ---- ---- ----
          WR


All buffers full. Producer waits.
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 6 of 9.)

Sample Output (Cont.):

```
Consumer reads 14
(buffers occupied: 2)
buffers:  14   15   16
         ---- ---- ----
          W     R

Producer writes 17
(buffers occupied: 3)
buffers:  17   15   16
         ---- ---- ----
               WR


Consumer reads 15
(buffers occupied: 2)
buffers:  17   15   16
         ---- ---- ----
               W     R
```


6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 7 of 9.)

```
Consumer reads 16
(buffers occupied: 1)
buffers:  17   15   16
         ---- ---- ----
           R   W


Consumer reads 17
(buffers occupied: 0)
buffers:  17   15   16
         ---- ---- ----
               WR


Producer writes 18
(buffers occupied: 1)
buffers:  17   18   16
         ---- ---- ----
                R   W
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 8 of 9.)

Sample Output (Cont.):

```
Consumer reads 18
(buffers occupied: 0)
buffers:  17   18   16
          ---- ---- ----
                     WR

All buffers empty. Consumer waits.

Producer writes 19
(buffers occupied: 1)
buffers:  17   18   19
          ---- ---- ----
           W          R

Consumer reads 19
(buffers occupied: 0)
buffers:  17   18   19
          ---- ---- ----
           WR
```


6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 9 of 9.)

```
Producer writes 20
(buffers occupied: 1)
buffers:  20   18   19
          ---- ---- ----
            R    W

Producer done producing.
Terminating Producer.

Consumer reads 20
(buffers occupied: 0)
buffers:  20   18   19
          ---- ---- ----
                 WR
```

```
Consumer read values totaling: 155.
Terminating Consumer.
```