16.6 Producer/Consumer Relationship without Synchronization

• **Buffer**
  - Shared memory region

• **Producer thread**
  - Generates data to add to buffer
  - Calls `wait` if consumer has not read previous message in buffer
  - Writes to empty buffer and calls `notify` for consumer

• **Consumer thread**
  - Reads data from buffer
  - Calls `wait` if buffer empty

• **Synchronize threads to avoid corrupted data**
public interface Buffer {
    public void set(int value);  // place value into Buffer
    public int get();              // return value from Buffer
}

// Fig. 16.4: Buffer.java
// Buffer interface specifies methods called by Producer and Consumer.
// Fig. 16.5: Producer.java
// Producer's run method controls a thread that
// stores values from 1 to 4 in sharedLocation.

public class Producer extends Thread {
    private Buffer sharedLocation; // reference to shared object

    // constructor
    public Producer( Buffer shared )
    {
        super( "Producer" );
        sharedLocation = shared;
    }

    // store values from 1 to 4 in sharedLocation
    public void run()
    {
        for ( int count = 1; count <= 4; count++ ) {

            // sleep 0 to 3 seconds, then place value in Buffer
            try {
                Thread.sleep( ( int )( Math.random() * 3001 ) );
                sharedLocation.set( count );
            }
        }
    }
}
// if sleeping thread interrupted, print stack trace

catch ( InterruptedException exception ) {
    exception.printStackTrace();
}

} // end for

System.err.println( getName() + " done producing." +
    "\nTerminating " + getName() + ").");

} // end method run

} // end class Producer
// Fig. 16.6: Consumer.java
// Consumer's run method controls a thread that loops four
// times and reads a value from sharedLocation each time.

public class Consumer extends Thread {
    private Buffer sharedLocation; // reference to shared object

    // constructor
    public Consumer( Buffer shared )
    {
        super( "Consumer" );
        sharedLocation = shared;
    }

    // read sharedLocation's value four times and sum the values
    public void run()
    {
        int sum = 0;

        for ( int count = 1; count <= 4; count++ ) {
            // sleep 0 to 3 seconds, read value from Buffer and add to sum
            try {
                Thread.sleep( ( int ) ( Math.random() * 3001 ) );
                sum += sharedLocation.get();
            }
        }
    }
}
// if sleeping thread interrupted, print stack trace
catch (InterruptedException exception) {
  exception.printStackTrace();
}

System.err.println(getName() + " read values totaling: " + sum + ".\nTerminating " + getName() + ".");

} // end method run

} // end class Consumer
// Fig. 16.7: UnsynchronizedBuffer.java
// UnsynchronizedBuffer represents a single shared integer.

public class UnsynchronizedBuffer implements Buffer {
    private int buffer = -1; // shared by producer and consumer threads

    // place value into buffer
    public void set( int value )
    {
        System.err.println( Thread.currentThread().getName() + " writes " + value );
        buffer = value;
    }

    // return value from buffer
    public int get()
    {
        System.err.println( Thread.currentThread().getName() + " reads " + buffer );
        return buffer;
    }

} // end class UnsynchronizedBuffer
// Fig. 16.8: SharedBufferTest.java
// SharedBufferTest creates producer and consumer threads.

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

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

        producer.start(); // start producer thread
        consumer.start(); // start consumer thread
    }
}

© 2003 Prentice Hall, Inc.
All rights reserved.
Consumer reads -1
Producer writes 1
Consumer reads 1
Consumer reads 1
Consumer reads 1
Consumer read values totaling: 2.
Terminating Consumer.
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.

Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 13.
Terminating Consumer.
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer read values totaling: 10.
Terminating Consumer.
16.7 Producer/Consumer Relationship with Synchronization

• Synchronize threads to ensure correct data
// Fig. 16.9: SynchronizedBuffer.java
// SynchronizedBuffer synchronizes access to a single shared int
public class SynchronizedBuffer implements Buffer {
    // shared by producer and consumer threads
    private int buffer = -1;
    private int occupiedBufferCount = 0;

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

        // while there are no empty locations, place thread in waiting state
        while ( occupiedBufferCount == 1 ) {
            // output thread information and buffer information, then wait
            try {
                System.err.println( name + " tries to write." );
                displayState( "Buffer full. " + name + " waits." );
                wait();
            } catch ( InterruptedException exception ) {
                exception.printStackTrace();
            }

            // if waiting thread interrupted, print stack trace
            catch ( InterruptedException exception ) {
                exception.printStackTrace();
            }
        }
    }
}
buffer = value; // set new buffer value

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

displayState( name + " writes " + buffer );

notify(); // tell waiting thread to enter ready state

} // end method set; releases lock on SynchronizedBuffer

// return value from buffer
public synchronized int get()
{
    // for output purposes, get name of thread that called this method
    String name = Thread.currentThread().getName();
// while no data to read, place thread in waiting state
while ( occupiedBufferCount == 0 ) {

    // output thread information and buffer information, then wait
    try {
        System.err.println( name + " tries to read." );
        displayState( "Buffer empty. " + name + " waits." );
        wait();
    }

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

} // end while

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

displayState( name + " reads " + buffer );

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

return buffer;
public void displayState( String operation )
{
    StringBuffer outputLine = new StringBuffer( operation );
    outputLine.setLength( 40 );
    outputLine.append( buffer + "	" + occupiedBufferCount );
    System.err.println( outputLine );
    System.err.println();
}

} // end class SynchronizedBuffer
```java
// Fig. 16.10: SharedBufferTest2.java
// SharedBufferTest2 creates producer and consumer threads.

public class SharedBufferTest2 {
    public static void main( String[] args ) {
        // create shared object used by threads; we use a SynchronizedBuffer
        // reference rather than a Buffer reference so we can invoke
        // SynchronizedBuffer method displayState from main
        SynchronizedBuffer sharedLocation = new SynchronizedBuffer();

        // Display column heads for output
        StringBuffer columnHeads = new StringBuffer( "Operation" );
        columnHeads.setLength( 40 );
        columnHeads.append( "Buffer		Occupied 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 );
    }
}
```

Create a **Buffer** object

Output initial state

Create a **Producer** and a **Consumer**
### Lines 25-26

```java
25    producer.start();  // start producer thread
26    consumer.start();  // start consumer thread
27
28    } // end main
29
30    } // end class SharedBufferTest2
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial State</strong></td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

© 2003 Prentice Hall, Inc.
All rights reserved.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Producer done producing.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminating Producer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consumer read values totaling: 10.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminating Consumer.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Producer tries to write.
Buffer full. Producer waits.  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.
Consumer reads 4         4  0
Consumer read values totaling: 10.
Terminating Consumer.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Action</td>
<td>Value 1</td>
<td>Value 2</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Producer done producing.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminating Producer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consumer reads 4</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Consumer read values totaling: 10.
Terminating Consumer.