Multithreading
History of Multi-processing & Multithreading

• 60’s - I/O concurrency – (Operating Systems)
• Multi-user systems
• Pipelined/Vector processors
• 70’s - Co-processors
• 80’s - Parallel Processing Systems
  – SIMD, MIMD, shared/distributed memory
• Distributed Computing
• Threads, Dataflow Computing
• 90’s - Grid Computing
• 00’s – Multi-scale computing
23.2 Thread States: Life Cycle of a Thread
23.3 Thread Priorities and Thread Scheduling
23.4 Creating and Executing Threads
23.5 Thread Synchronization
23.6 Producer/Consumer Relationship without Synchronization
23.7 Producer/Consumer Relationship with Synchronization
23.8 Producer/Consumer Relationship: Circular Buffer
23.9 Producer/Consumer Relationship: ArrayBlockingQueue
23.10 Multithreading with GUI
23.11 Other Classes and Interfaces in java.util.concurrent
23.12 Monitors and Monitor Locks
23.13 Wrap-Up
23.1 Introduction: Multi-threading in Java

- Provides application with multiple threads of execution
- Allows programs to perform tasks concurrently
- Often requires programmer to synchronize threads to function correctly
A problem with single-threaded applications is that lengthy activities must complete before other activities can begin. In a multithreaded application, threads can be distributed across multiple processors (if they are available) so that multiple tasks are performed concurrently (simultaneously) and the application can operate more efficiently. Multithreading can also increase performance on single-processor systems that simulate concurrency—when one thread cannot proceed, another can use the processor.
Portability Tip 23.1

Unlike languages that do not have built-in multithreading capabilities (such as C and C++) and must therefore make nonportable calls to operating system multithreading primitives, Java includes multithreading primitives as part of the language itself and as part of its libraries. This facilitates manipulating threads in a portable manner across platforms.
23.2 Thread States: Life Cycle of a Thread

• Thread states
  – new state
    • New thread begins its life cycle in the new state
    • Remains in this state until program starts the thread, placing it in the runnable state
  – runnable state
    • A thread in this state is executing its task
  – waiting state
    • A thread transitions to this state to wait for another thread to perform a task
23.2 Thread States: Life Cycle of a Thread

• Thread states
  – **timed waiting** state
    • A thread enters this state to wait for another thread or for an amount of time to elapse
    • A thread in this state returns to the *runnable* state when it is signaled by another thread or when the timed interval expires
  – **terminated** state
    • A *runnable* thread enters this state when it completes its task
Fig. 23.1 | Thread life-cycle UML state diagram.
23.2 Thread States: Life Cycle of a Thread

• Operating system view of *runnable* state
  – *ready* state
    • A thread in this state is not waiting for another thread, but is waiting for the operating system to assign the thread a processor
  – *running* state
    • A thread in this state currently has a processor and is executing
  – A thread in the *running* state often executes for a small amount of processor time called a time slice or quantum before transitioning back to the *ready* state
Fig. 23.2 | Operating system’s internal view of Java’s runnable state.
23.3 Thread Priorities and Thread Scheduling

• Priorities
  – Every Java thread has a priority
  – Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10)
  – Threads with a higher priority are more important and will be allocated a processor before threads with a lower priority
  – Default priority is NORM_PRIORITY (a constant of 5)
23.3 Thread Priorities and Thread Scheduling

- Thread scheduler
  - Determine which thread runs next
  - Simple implementation runs equal-priority threads in a round-robin fashion
  - Higher-priority threads can preempt the currently running thread
  - In some cases, higher-priority threads can indefinitely postpone lower-priority threads which is also known as starvation
Portability Tip 23.2

Thread scheduling is platform dependent—an application that uses multithreading could behave differently on separate Java implementations.
Portability Tip 23.3

When designing applets and applications that use threads, you must consider the threading capabilities of all the platforms on which the applets and applications will execute.
Fig. 23.3 | Thread-priority scheduling.
23.4 Creating and Executing Threads

• **Runnable interface**
  – Preferred means of creating a multithreaded application
  – Declares method `run`
  – Executed by an object that implements the `Executor` interface

• **Executor interface**
  – Declares method `execute`
  – Creates and manages a group of threads called a thread pool
23.4 Creating and Executing Threads

- **ExecutorService interface**
  - Subinterface of Executor that declares other methods for managing the life cycle of an Executor
  - Can be created using static methods of class Executors
  - Method shutdown ends threads when tasks are completed

- **Executors class**
  - Method newFixedThreadPool creates a pool consisting of a fixed number of threads
  - Method newCachedThreadPool creates a pool that creates new threads as they are needed
// Fig. 23.4: PrintTask.java
// PrintTask class sleeps for a random time from 0 to 5 seconds
import java.util.Random;

class PrintTask implements Runnable
{
    private int sleepTime; // random sleep time for thread
    private String threadName; // name of thread
    private static Random generator = new Random();

    // assign name to thread
    public PrintTask( String name )
    {
        threadName = name; // set name of thread

        // pick random sleep time between 0 and 5 seconds
        sleepTime = generator.nextInt( 5000 );
    } // end PrintTask constructor

    Implement runnable to create separate thread
// method run is the code to be executed by new thread
public void run()
{
    try // put thread to sleep for sleepTime amount of time
    {
        System.out.printf("%s going to sleep for %d milliseconds.\n", threadName, sleepTime);
        Thread.sleep(sleepTime); // put thread to sleep
    } // end try
    // if thread interrupted while sleeping, print stack trace
    catch (InterruptedException exception)
    {
        exception.printStackTrace();
    } // end catch
    // print thread name
    System.out.printf("%s done sleeping\n", threadName);
} // end method run
} // end class PrintTask
// Fig. 23.5: RunnableTester.java
// Multiple threads printing at different intervals.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester
{
    public static void main( String[] args )
    {
        // create and name each runnable
        PrintTask task1 = new PrintTask( "thread1" );
        PrintTask task2 = new PrintTask( "thread2" );
        PrintTask task3 = new PrintTask( "thread3" );

        System.out.println( "Starting threads" );

        // create ExecutorService to manage threads
        ExecutorService threadExecutor = Executors.newFixedThreadPool( 3 );

        // start threads and place in runnable state
        threadExecutor.execute( task1 ); // start task1
        threadExecutor.execute( task2 ); // start task2
        threadExecutor.execute( task3 ); // start task3

        threadExecutor.shutdown(); // shutdown worker threads
    }
}

Create three PrintTask; each will run in a separate thread
Create fixed thread pool to execute and manage threads
Execute each task; this method will assign a thread to the runnable
Shutdown thread pool when runnables complete their tasks

© 2005 Pearson Education, Inc. All rights reserved.
Starting threads
Threads started, main ends

thread1 going to sleep for 1217 milliseconds
thread2 going to sleep for 3989 milliseconds
thread3 going to sleep for 662 milliseconds
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping

Starting threads

thread1 going to sleep for 314 milliseconds
thread2 going to sleep for 1990 milliseconds
Threads started, main ends

thread3 going to sleep for 3016 milliseconds
thread1 done sleeping
thread2 done sleeping
thread3 done sleeping
23.6 Producer/Consumer Relationship without Synchronization

- **Producer/consumer relationship**
  - Producer generates data and stores it in shared memory
  - Consumer reads data from shared memory
  - Shared memory is called the buffer
// Fig. 23.6: Buffer.java
// Buffer interface specifies methods called by Producer and Consumer.

public interface Buffer {
    public void set(int value); // place int value into Buffer
    public int get(); // return int value from Buffer
} // end interface Buffer
public class Producer implements Runnable {
    private static Random generator = new Random();
    private Buffer sharedLocation; // reference to shared object

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

    // store values from 1 to 10 in sharedLocation
    public void run()
    {
        int sum = 0;
    }
for ( int count = 1; count <= 10; count++ )
{
    try // sleep 0 to 3 seconds, then place value in Buffer
    {
        Thread.sleep( generator.nextInt( 3000 ) ); // sleep thread
        sharedLocation.set( count ); // set value in buffer
        sum += count; // increment sum of values
        System.out.printf( "%d
", sum );
    } // end try
    // if sleeping thread interrupted, print stack trace
    catch ( InterruptedException exception )
    {
        exception.printStackTrace();
    } // end catch
} // end for

System.out.printf( "Producer done producing."
"Terminating Producer." );
} // end method run
} // end class Producer
// Fig. 23.8: Consumer.java
// Consumer's run method loops ten times reading a value from buffer.
import java.util.Random;

public class Consumer implements Runnable {
    private static Random generator = new Random();
    private Buffer sharedLocation; // reference to shared object

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

    // read sharedLocation's value four times and sum the values
    public void run() {
        int sum = 0;
        Implement the runnable interface so that producer will run in a separate thread
        Declare run method to satisfy interface
    } // end run method
} // end class Consumer

© 2005 Pearson Education, Inc. All rights reserved.
for ( int count = 1; count <= 10; count++ )
{
    // sleep 0 to 3 seconds, read value from buffer and add to sum
    try
    {
        Thread.sleep( generator.nextInt( 3000 ) );
        sum += sharedLocation.get();
        System.out.printf( "%t\t%2d\n", sum );
    } // end try
    // if sleeping thread interrupted, print stack trace
    catch ( InterruptedException exception )
    {
        exception.printStackTrace();
    } // end catch
} // end for

System.out.printf( "\n%3d\nS", sum, "Terminating Consumer." );
} // end method run
} // end class Consumer
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.out.printf( "Producer writes\t%2d", value );
        buffer = value;
    } // end method set

    // return value from buffer
    public int get()
    {
        System.out.printf( "Consumer reads\t%2d", buffer );
        return buffer;
    } // end method get
} // end class UnsynchronizedBuffer
// Fig 23.10: SharedBufferTest.java
// Application shows two threads manipulating an unsynchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest
{
    public static void main( String[] args )
    {
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool( 2 );

        // create UnsynchronizedBuffer to store ints
        Buffer sharedLocation = new UnsynchronizedBuffer();
System.out.println( "Action\t\tValue\tProduced\tConsumed" );
System.out.println( "------\t\t-----\t--------\t--------\n" );

// try to start producer and consumer giving each of them access
// to sharedLocation
try
{
    application.execute( new Producer( sharedLocation ) );
    application.execute( new Consumer( sharedLocation ) );
} // end try

// terminate application when threads end
application.shutdown();
<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
<th>Produced</th>
<th>Consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer writes</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Producer writes</td>
<td>3</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Consumer reads</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>4</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Consumer reads</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>5</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Producer writes</td>
<td>6</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>Producer writes</td>
<td>7</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>Consumer reads</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>7</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>Producer writes</td>
<td>8</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>Consumer reads</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>8</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>Producer writes</td>
<td>9</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>Producer writes</td>
<td>10</td>
<td></td>
<td>55</td>
</tr>
</tbody>
</table>

Producer done producing.
Terminating Producer.
Consumer reads 10   47
Consumer reads 10   57
Consumer reads 10   67
Consumer reads 10   77

Consumer read values totaling 77.
Terminating Consumer.
<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
<th>Produced</th>
<th>Consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer reads</td>
<td>-1</td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>Producer writes</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Producer writes</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>2</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Producer writes</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>3</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Producer writes</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>4</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>Producer writes</td>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>6</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>7</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>8</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>9</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>10</td>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>

Consumer read values totaling 19.

Terminating Consumer.

Producer done producing.

Terminating Producer.
23.5 Thread Synchronization

- Thread synchronization
  - Provided to the programmer with mutual exclusion
    - Exclusive access to a shared object
    - Implemented in Java using locks

- Lock interface
  - `lock` method obtains the lock, enforcing mutual exclusion
  - `unlock` method releases the lock
  - `Class ReentrantLock` implements the `Lock` interface
Using a Lock with a fairness policy helps avoid indefinite postponement, but can also dramatically reduce the overall efficiency of a program. Because of the large decrease in performance, fair locks are only necessary in extreme circumstances.
23.5 Thread Synchronization

• **Condition variables**
  – If a thread holding the lock cannot continue with its task until a condition is satisfied, the thread can wait on a condition variable
  – Create by calling `Lock` method `newCondition`
  – Represented by an object that implements the `Condition` interface

• **Condition interface**
  – Declares methods `await`, to make a thread wait, `signal`, to wake up a waiting thread, and `signalAll`, to wake up all waiting threads
Common Programming Error 23.1

**Deadlock** occurs when a waiting thread (let us call this thread1) cannot proceed because it is waiting (either directly or indirectly) for another thread (let us call this thread2) to proceed, while simultaneously thread2 cannot proceed because it is waiting (either directly or indirectly) for thread1 to proceed. Two threads are waiting for each other, so the actions that would enable each thread to continue execution never occur.
Error-Prevention Tip 23.1

When multiple threads manipulate a shared object using locks, ensure that if one thread calls method `await` to enter the `waiting` state for a condition variable, a separate thread eventually will call `Condition` method `signal` to transition the thread waiting on the condition variable back to the `Runnable` state. (cont...)
If multiple threads may be waiting on the condition variable, a separate thread can call `Condition` method `signalAll` as a safeguard to ensure that all the waiting threads have another opportunity to perform their tasks. If this is not done, indefinite postponement or deadlock could occur.
Software Engineering Observation 23.1

The locking that occurs with the execution of the lock and unlock methods could lead to deadlock if the locks are never released. Calls to method unlock should be placed in finally blocks to ensure that locks are released and avoid these kinds of deadlocks.
Performance Tip 23.3

Synchronization to achieve correctness in multithreaded programs can make programs run more slowly, as a result of thread overhead and the frequent transition of threads between the waiting and runnable states. There is not much to say, however, for highly efficient yet incorrect multithreaded programs!
It is an error if a thread issues an await, a signal, or a signalAll on a condition variable without having acquired the lock for that condition variable. This causes an IllegalMonitorStateException.
23.7 Producer/Consumer Relationship with Synchronization

• **Producer/consumer relationship**
  - This example uses **Locks and Conditions** to implement synchronization
// Fig. 23.11: SynchronizedBuffer.java
// SynchronizedBuffer synchronizes access to a single shared integer.
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class SynchronizedBuffer implements Buffer {
    // Lock to control synchronization with this buffer
    private Lock accessLock = new ReentrantLock();

    // conditions to control reading and writing
    private Condition canWrite = accessLock.newCondition();
    private Condition canRead = accessLock.newCondition();

    private int buffer = -1; // shared by producer and consumer threads
    private boolean occupied = false; // whether buffer

    // place int value into buffer
    public void set(int value) {
        accessLock.lock(); // lock this object
        // Try to obtain the lock before setting the value of the shared data
    }
// output thread information and buffer information, then wait
try {
    // while buffer is not empty, place thread in waiting state
    while (occupied)
    {
        System.out.println( "Producer tries to write." );
        displayState( "Buffer full. Producer waits." );
        canWrite.await(); // wait until buffer is empty
    } // end while

    buffer = value; // set new buffer value
    // indicate producer cannot store another value
    // until consumer retrieves current buffer value
    occupied = true;
}
displayState( "Producer writes " + buffer );

// signal thread waiting to read from buffer
canRead.signal();

} // end try

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

finally
{
   accessLock.unlock(); // unlock this object
} // end finally

} // end method set

// return value from buffer
public int get()
{
   int readValue = 0; // initialize value read from buffer
   accessLock.lock(); // lock this object
// output thread information and buffer information, then wait
try
{
    // while no data to read, place thread in waiting state
    while ( !occupied )
    {
        System.out.println( "Consumer tries to read." );
        displayState( "Buffer empty. Consumer waits." );
        canRead.await(); // wait until buffer is full
    } // end while

    // indicate that producer can store another value
    // because consumer just retrieved buffer value
    occupied = false;

    readValue = buffer; // retrieve value from buffer
    displayState( "Consumer reads " + readValue );
} // end try
// signal thread waiting for buffer to be empty
    canWrite.signal();
} // end try

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

finally
    {
        accessLock.unlock(); // unlock this object
    } // end finally

    return readValue;
} // end method get

// display current operation and buffer state
    public void displayState( String operation )
    {
        System.out.printf( "%-40s%d		%b\n\n", operation, buffer,
                          occupied );
    } // end method displayState
} // end class SynchronizedBuffer
Common Programming Error 23.3

Place calls to `Lock` method `unlock` in a `finally` block. If an exception is thrown, `unlock` must still be called or deadlock could occur.
Software Engineering Observation 23.2

Always invoke method `await` in a loop that tests an appropriate condition. It is possible that a thread will reenter the `Runnable` state before the condition it was waiting on is satisfied. Testing the condition again ensures that the thread will not erroneously execute if it was signaled early.
Common Programming Error 23.4

Forgetting to signal a thread that is waiting for a condition is a logic error. The thread will remain in the \textit{waiting} state, which will prevent the thread from doing any further work. Such waiting can lead to indefinite postponement or deadlock.
public class SharedBufferTest2 {
    public static void main(String[] args) {
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool(2);
        // create SynchronizedBuffer to store ints
        Buffer sharedLocation = new SynchronizedBuffer();
    }
}

Create SynchronizedBuffer to be shared between producer and consumer
System.out.printf("%-40s\t\t%-40s\n", "Operation", "Buffer", "Occupied", "--------", "--------\t\t--------")

try // try to start producer and consumer
{
    application.execute(new Producer(sharedLocation));
    application.execute(new Consumer(sharedLocation));
} // end try
catch (Exception exception)
{
    exception.printStackTrace();
} // end catch

application.shutdown();

} // end main

} // end class SharedBufferTest2

Execute the producer and consumer in separate threads
<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td><strong>Producer tries to write.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Buffer full. Producer waits.</strong></td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td><strong>Producer tries to write.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Buffer full. Producer waits.</strong></td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 4</td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td><strong>Consumer tries to read.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Buffer empty. Consumer waits.</strong></td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 5</td>
<td>5</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 5</td>
<td>5</td>
<td>false</td>
</tr>
<tr>
<td><strong>Consumer tries to read.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Buffer empty. Consumer waits.</strong></td>
<td>5</td>
<td>false</td>
</tr>
<tr>
<td>Operation</td>
<td>Value</td>
<td>Success</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>Producer writes 6</td>
<td>6</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 6</td>
<td>6</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 7</td>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 7</td>
<td>7</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 8</td>
<td>8</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 8</td>
<td>8</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 9</td>
<td>9</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 9</td>
<td>9</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 10</td>
<td>10</td>
<td>true</td>
</tr>
</tbody>
</table>

Producer done producing.
Terminating Producer.
Consumer reads 10          | 10    | false   |

Consumer read values totaling 55.
Terminating Consumer.
23.8 Producer/Consumer Relationship

Circular Buffer

• Circular buffer
  – Provides extra buffer space into which producer can place values and consumer can read values
Performance Tip 23.4

Even when using a circular buffer, it is possible that a producer thread could fill the buffer, which would force the producer thread to wait until a consumer consumes a value to free an element in the buffer. Similarly, if the buffer is empty at any given time, the consumer thread must wait until the producer produces another value. The key to using a circular buffer is to optimize the buffer size to minimize the amount of thread wait time.
public class CircularBuffer implements Buffer {
  private Lock accessLock = new ReentrantLock();
  private Condition canWrite = accessLock.newCondition();
  private Condition canRead = accessLock.newCondition();
  private int[] buffer = { -1, -1, -1 };  
  private int occupiedBuffers = 0; // count number of buffers used
  private int writeIndex = 0; // index to write next value
  private int readIndex = 0; // index to read next value
  
  // place value into buffer
  public void set( int value ) {
    accessLock.lock(); // lock this object

    // Lock to impose mutual exclusion
    // Condition variables to control writing and reading
    // Circular buffer; provides three spaces for data
    // Obtain the lock before writing data to the circular buffer
  }
}
```java
// output thread information and buffer information, then wait
try {
    // while no empty locations, place thread in waiting state
    while (occupiedBuffers == buffer.length) {
        System.out.printf("All buffers full. Producer waits.\n");
        canWrite.await(); // await until a buffer element is free
    } // end while

    buffer[writeIndex] = value; // set new buffer
    // update circular write index
    writeIndex = (writeIndex + 1) % buffer.length;
    occupiedBuffers++; // one more buffer element
    displayState("Producer writes " + buffer[writeIndex]);
    canRead.signal(); // signal threads waiting to read from buffer
} // end try
catch (InterruptedException exception) {
    exception.printStackTrace();
} // end catch
finally {
    accessLock.unlock(); // unlock this object
} // end finally
} // end method set
```

- Wait until a buffer space is empty
- Update index; this statement imposes the circularity of the buffer
- Signal waiting thread it can now read data from buffer
- Release the lock
// return value from buffer
public int get()
{
    int readValue = 0; // initialize value read from buffer
    accessLock.lock(); // lock this object

    // wait until buffer has data, then read value
    try
    {
        // while no data to read, place thread in waiting state
        while (occupiedBuffers == 0)
        {
            System.out.printf( "All buffers empty. Consumer waits.\n" );
            canRead.await(); // await until a buffer element is filled
        } // end while

        readValue = buffer[readIndex]; // read value

        // update circular read index
        readIndex = (readIndex + 1) % buffer.length;
    }
    catch (InterruptedException e)
    {
        Thread.currentThread().interrupt();
    }

    accessLock.unlock(); // unlock this object
    return readValue; // return value read from buffer
}
occupiedBuffers--; // one more buffer element is empty

displayState( "Consumer reads " + readValue );

canWrite.signal(); // signal threads waiting to write to buffer

} // end try

// if waiting thread interrupted, print stack trace

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

finally
{
    accessLock.unlock(); // unlock this object
} // end finally

return readValue;
} // end method get

// display current operation and buffer state

displayState( String operation )
{
    // output operation and number of occupied buffers
    System.out.printf( "%s%d\n%s", operation,
        " (buffers occupied: ", occupiedBuffers, "buffers: ");

    for ( int value : buffer )
        System.out.printf( " %2d ", value ); // output values in buffer
System.out.print( "\n   " );
for ( int i = 0; i < buffer.length; i++ )
    System.out.print( "---- " );

System.out.print( "\n   " );
for ( int i = 0; i < buffer.length; i++ )
{
    if ( i == writeIndex && i == readIndex )
        System.out.print( " WR" ); // both write and read index
    else if ( i == writeIndex )
        System.out.print( " W   " ); // just write index
    else if ( i == readIndex )
        System.out.print( "  R  " ); // just read index
    else
        System.out.print( "     " ); // neither index
} // end for

System.out.println( "\n" );
} // end method displayState
} // end class CircularBuffer
Fig 23.14: CircularBufferTest.java

// Application shows two threads manipulating a circular buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CircularBufferTest
{
    public static void main( String[] args )
    {
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool( 2 );

        // create CircularBuffer to store ints
        Buffer sharedLocation = new CircularBuffer();

        try // try to start producer and consumer
        {
            application.execute( new Producer( sharedLocation ) );
            application.execute( new Consumer( sharedLocation ) );
        } // end try
        catch ( Exception exception )
        {
            exception.printStackTrace();
        } // end catch

        application.shutdown();
    } // end main
} // end class CircularBufferTest
Producer writes 1 (buffers occupied: 1)
buffers:  1  -1  -1
        ---- ---- ----
          R   W

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

All buffers empty. Consumer waits.
Producer writes 2 (buffers occupied: 1)
buffers:  1  2  -1
        ---- ---- ----
          R   W

Consumer reads 2 (buffers occupied: 0)
buffers:  1  2  -1
        ---- ---- ----
          WR

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

Consumer reads 3 (buffers occupied: 0)
buffers:  1  2  3
        ---- ---- ----
            WR

Producer writes 4 (buffers occupied: 1)
buffers:  4  2  3
        ---- ---- ----
              R   W
Producer writes 5 (buffers occupied: 2)
buffers: 4 5 3
       ---- ---- ----
          R    W

Consumer reads 4 (buffers occupied: 1)
buffers: 4 5 3
       ---- ---- ----
         R   W

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

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

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

Producer writes 8 (buffers occupied: 3)
buffers: 7 8 6
       ---- ---- ----
                        WR
Consumer reads 6 (buffers occupied: 2)
buffers:    7    8    6
---- ---- ----
        R    W

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

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

Consumer reads 8 (buffers occupied: 1)
buffers:    7    8    9
---- ---- ----
        W    R

Consumer reads 9 (buffers occupied: 0)
buffers:    7    8    9
---- ---- ----
        WR

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

Producer done producing.
Terminating Producer.

Consumer reads 10 (buffers occupied: 0)
buffers:   10    8    9
---- ---- ----
        WR

Consumer read values totaling: 55.
Terminating Consumer.
23.10 Multithreading with GUI

• Swing GUI components
  – Not thread safe
  – Updates should be performed in the event-dispatching thread
    • Use static method invokeLater of class SwingUtilities and pass it a Runnable object
public class RunnableObject implements Runnable {
    // get name of executing thread
    final String threadName = Thread.currentThread().getName();
}
while ( true ) // infinite loop; will be terminated from outside
{
    try
    {
        // sleep for up to 1 second
        Thread.sleep( generator.nextInt( 1000 ) );
        lockObject.lock(); // obtain the lock
        try
        {
            while ( suspended ) // loop until not suspended
            {
                suspend.await(); // suspend thread execution
            } // end while
        } // end try
        finally
        {
            lockObject.unlock(); // unlock the lock
        } // end finally
    } // end try
    // if thread interrupted during wait/sleep
    catch ( InterruptedException exception )
    {
        exception.printStackTrace(); // print stack trace
    } // end catch
// display character on corresponding JLabel
SwingUtilities.invokeLater(
    new Runnable()
    {
        // pick random character and display it
        public void run()
        {
            // select random uppercase letter
            char displayChar =
                ( char ) ( generator.nextInt( 26 ) + 65 );

            // output character in JLabel
            output.setText( threadName + " : " + displayChar );
        } // end method run
    } // end inner class
); // end call to SwingUtilities.invokeLater
} // end while
} // end method run
public void toggle()
{
    suspended = !suspended; // toggle boolean controlling state

    // change label color on suspend/resume
    output.setBackground( suspended ? Color.RED : Color.GREEN );

    lockObject.lock(); // obtain lock
    try
    {
        if ( !suspended ) // if thread resumed
        {
            suspend.signal(); // resume thread
        } // end if
    } // end try
    finally
    {
        lockObject.unlock(); // release lock
    } // end finally
} // end method toggle
} // end class RunnableObject
// Fig. 23.18: RandomCharacters.java
// Class RandomCharacters demonstrates the Runnable interface
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class RandomCharacters extends JFrame implements ActionListener {
    private final static int SIZE = 3; // number of threads
    private JCheckBox checkboxes[]; // array of JCheckBoxes
    private Lock lockObject = new ReentrantLock( true ); // single lock

    // array of RunnableObjects to display random characters
    private RunnableObject[] randomCharacters =
        new RunnableObject[ SIZE ];
}
public RandomCharacters()
{
    checkboxes = new JCheckBox[ SIZE ]; // allocate space for array
   setLayout( new GridLayout( SIZE, 2, 5, 5 ) ); // set layout

    // create new thread pool with SIZE threads
    ExecutorService runner = Executors.newFixedThreadPool( SIZE );

    // loop SIZE times
    for ( int count = 0; count < SIZE; count++ )
    {
        JLabel outputJLabel = new JLabel(); // create JLabel
        outputJLabel.setBackground( Color.GREEN ); // set color
        outputJLabel.setOpaque( true ); // set JLabel to be opaque
        add( outputJLabel ); // add JLabel to JFrame

        // create JCheckBox to control suspend/resume state
        checkboxes[ count ] = new JCheckBox( "Suspended" );

        // add listener which executes when JCheckBox is clicked
        checkboxes[ count ].addActionListener( this );
        add( checkboxes[ count ] ); // add JCheckBox to JFrame
    }
}
// create a new RunnableObject
randomCharacters[count] =
    new RunnableObject(lockObject, outputJLabel);

// execute RunnableObject
runner.execute(randomCharacters[count]);
} // end for

setSize(275, 90); // set size of window
setVisible(true); // show window

runner.shutdown(); // shutdown runner when threads finish
} // end RandomCharacters constructor

// handle JCheckBox events
public void actionPerformed(ActionEvent event)
{
    // loop over all JCheckBoxes in array
    for (int count = 0; count < checkboxes.length; count++)
    {
        // check if this JCheckBox was source of event
        if (event.getSource() == checkboxes[count])
            randomCharacters[count].toggle(); // toggle state
    } // end for
} // end method actionPerformed

Execute a Runnable

Startup thread pool when threads finish their tasks
public static void main( String args[] )
{
    // create new RandomCharacters object
    RandomCharacters application = new RandomCharacters();
    // set application to end when window is closed
    application.setDefaultCloseOperation( EXIT_ON_CLOSE );
} // end main
} // end class RandomCharacters
23.11 Other Classes and Interfaces in java.util.concurrent

- **Callable interface**
  - Declares method `call`
  - Method `call` allows a concurrent task to return a value or throw an exception
  - `ExecutorService` method `submit` takes a `Callable` and returns a `Future` representing the result of the task

- **Future interface**
  - Declares method `get`
  - Method `get` returns the result of the task represented by the `Future`
23.12 Monitors and Monitor Locks

• Monitors
  – Every object in Java has a monitor
  – Allows one thread at a time to execute inside a synchronized statement
  – Threads waiting to acquire the monitor lock are placed in the blocked state
  – Object method wait places a thread in the waiting state
  – Object method notify wakes up a waiting thread
  – Object method notifyAll wakes up all waiting threads
Software Engineering Observation 23.3

The locking that occurs with the execution of synchronized methods could lead to deadlock if the locks are never released. When exceptions occur, Java’s exception mechanism coordinates with Java’s synchronization mechanism to release locks and avoid these kinds of deadlocks.
It is 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`.
public class SynchronizedBuffer implements Buffer {
    private int buffer = -1; // shared by producer and consumer threads
    private boolean occupied = false; // count of occupied buffers

    // place value into buffer
    public synchronized void set( int value ) {
        while ( occupied ) {
            // output thread information and buffer information, then wait
            try {
                System.out.println( "Producer tries to write." );
                displayState( "Buffer full. Producer waits." );
                wait();
            } catch ( InterruptedException exception ) {
                exception.printStackTrace();
            }
        }
        // output thread information and buffer information, then wait
        buffer = value; // set new buffer value
    }
} // end public synchronized void set( int value )
// indicate producer cannot store another value
// until consumer retrieves current buffer value
occupied = true;

displayState( "Producer writes " + buffer );

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

// return value from buffer
public synchronized int get()
{
    // while no data to read, place thread in waiting
    while ( !occupied )
    {
        // output thread information and buffer information, then wait
        try
        {
            System.out.println( "Consumer tries to read." );
            displayState( "Buffer empty. Consumer waits." );
            wait();
        } // end try
        catch ( InterruptedException exception )
        {
            exception.printStackTrace();
        } // end catch
    } // end while
} // end method get; releases lock on SynchronizedBuffer
58. // indicate that producer can store another value
59. // because consumer just retrieved buffer value
60. occupied = false;
61. 
62. int readValue = buffer; // store value in buffer
63. displayState( "Consumer reads " + readValue );
64. notify(); // tell waiting thread to enter runnable state
65. 
66. return readValue;
67. } // end method get; releases lock on SynchronizedBuffer
68. 
69. // display current operation and buffer state
70. public void displayState( String operation )
71. {
72.     System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
73.             occupied );
74. } // end method displayState
75. 
76. } // end class SynchronizedBuffer

Buffer is now empty
Notify thread it may now write to buffer
// Fig 23.20: SharedBufferTest2.java
// Application shows two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest2
{
    public static void main( String[] args )
    {
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool( 2 );

        // create SynchronizedBuffer to store ints
        Buffer sharedLocation = new SynchronizedBuffer();

        System.out.printf( "%-40s%s	%-40s%s
", "Operation", "Buffer", "Occupied", "--------	" );

        try // try to start producer and consumer
        {
            application.execute( new Producer( sharedLocation ) );
            application.execute( new Consumer( sharedLocation ) );
        } // end try
        catch ( Exception exception )
        {
            exception.printStackTrace();
        } // end catch
    }
} // end main
```java
application.shutdown();
```
Producer writes 6                      6               true
Consumer reads 6                     6               false
Consumer tries to read.             6               false
Buffer empty. Consumer waits.
Producer writes 7                    7               true
Consumer reads 7                     7               false
Consumer tries to read.             7               false
Buffer empty. Consumer waits.
Producer writes 8                    8               true
Consumer reads 8                     8               false
Producer writes 9                    9               true
Producer tries to write.            9               true
Buffer full. Producer waits.
Consumer reads 9                     9               false
Producer writes 10                   10              true
Producer done producing.            10              false
Terminating Producer.
Consumer reads 10                   10              false
Terminating Consumer.

Consumer read values totaling 55.  

Outline

synchronizedBuffer .java

(3 of 3)