CS360 Lecture 12
Multithreading

Thursday, March 11, 2004

Reading

Multithreading: Chapter 16

Thread States

At any time, a thread can be in one of several thread states:

- **Born:** this is the time when the thread is created.
  ```
  SimpleThread first = new SimpleThread( 5 );
  ```
- **Ready (runnable):** a thread transitions to the ready state once the start method is called:
  ```
  first.start();
  ```
- **Running:** a thread transitions to the running state once the operating system allocates a processor to the thread.
- **Dead:** A thread is dead once it's run method has completed, or it's terminated by an uncaught exception
- **Blocked:** A thread becomes blocked when it attempts to perform a task that cannot be completed immediately. The task is waiting for something outside of the code, for example I/O. Once the task can resume, the thread returns to the ready state and waits for processor time.
- **Waiting:** This happens because while executing the code, the thread comes across the `wait` method. It remains in the wait status until the either method `notify` or `notifyAll` is called.
- **Sleeping:** A thread can go to sleep for a specified time. It transitions to the ready state once that time has passed.

Thread Scheduler

The job of the scheduler is to make sure that the thread with the highest priority is running at all times. The thread scheduler also ensures that threads of the same priority are executed in a round-robin fashion. Each thread gets a specific quantum of time.

Higher priority threads could postpone the execution of lower priority threads indefinitely. Starvation!

If there are several threads of equal priority, one thread could yield to give other threads a chance to run. This is only useful in non-timesliced systems, where one thread could execute continuously until it completes execution. The thread yields by calling the method yield.

Yield Example

```
public class YieldThread extends Thread
{
  public void run()
  {
    for ( int count = 0; count < 4; count++)
    {
      System.out.println( count + " From: " + getName() );
      yield();
    }
  }
  public static void main( String[] args )
  {
    YieldThread first = new YieldThread();
    YieldThread second = new YieldThread();
    first.setPriority( 1);
    second.setPriority( 1);
    first.start();
    second.start();
    System.out.println( "End" );
  }
}
```

```
End
0 From: Thread-1
0 From: Thread-2
1 From: Thread-1
1 From: Thread-2
2 From: Thread-1
2 From: Thread-2
3 From: Thread-1
3 From: Thread-2
```

Thread methods

Threads are created in Java by extending the class Thread. This class contains several useful methods:

- **`interrupt()`:** Interrupts this thread.
- **`sleep(long millis)`:** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **`yield()`:** Causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **`destroy()`:** Destroys this thread, without any cleanup.
- **`wait()`:** Causes current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.

```
public class ThreadTester {

   public static void main( String [] args )
   {
      PrintThread thread1 = new PrintThread( "thread1" );
      PrintThread thread2 = new PrintThread( "thread2" );
      PrintThread thread3 = new PrintThread( "thread3" );

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

      thread1.start();
      thread2.start();
      thread3.start();

      System.err.println( "Threads started, main ends\n" );
   }
}

class PrintThread extends Thread {
   private int sleepTime;

   public PrintThread( String name )
   {
      super( name );

      sleepTime = ( int ) ( Math.random() * 5001 );
   }

   public void run()
   {
      try {
         System.err.println(
            getName() + " going to sleep for " +
            sleepTime );

         Thread.sleep( sleepTime );
      }

      catch ( InterruptedException exception ) {
         exception.printStackTrace();
      }
      System.err.println( getName() + " done sleeping" );
   }
}
```

```
Starting threads
Threads started, main ends

thread1 going to sleep for 3440
thread2 going to sleep for 3064
thread3 going to sleep for 4035
thread2 done sleeping
thread1 done sleeping
thread3 done sleeping

Starting threads
Threads started, main ends

thread1 going to sleep for 4571
thread2 going to sleep for 2209
thread3 going to sleep for 375
thread3 done sleeping
thread2 done sleeping
thread1 done sleeping

Starting threads
Threads started, main ends

thread1 going to sleep for 1183
thread2 going to sleep for 1834
thread3 going to sleep for 469
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
```

Thread Synchronization

Thread synchronization is needed for those times when multiple threads have access to the same object.

Synchronization is achieved through the use of monitors. These lock objects when they are being manipulated by a thread.

Let's look at what happens if we don't have synchronization.

```
public class SharedBufferTest {

    public static void main( String [] args )
    {
        Buffer sharedLocation = new UnsynchronizedBuffer();

        Producer producer = new Producer( sharedLocation );
```

```
         Consumer consumer = new Consumer( sharedLocation );

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

public class UnsynchronizedBuffer implements Buffer {
   private int buffer = -1;

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

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

public class Producer extends Thread {
   private Buffer sharedLocation;

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

   public void run()
   {
      for ( int count = 1; count <= 4; count++ ) {

         try {
            Thread.sleep(( int )( Math.random() * 3001 ) );
            sharedLocation.set( count );
         }
         catch ( InterruptedException exception ) {
            exception.printStackTrace();
         }
      }
      System.err.println( getName() + " done producing." +
```

```
                        "\nTerminating " + getName() + ".");
   }
}

public class Consumer extends Thread {
   private Buffer sharedLocation;

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

   public void run()
   {
      int sum = 0;

      for ( int count = 1; count <= 4; count++ ) {

         try {
            Thread.sleep(( int )( Math.random() * 3001 ) );
            sum += sharedLocation.get();
         }
         catch ( InterruptedException exception ) {
            exception.printStackTrace();
         }
      }
      System.err.println( getName()
                          + " read values totaling: " + sum
                          + ".\nTerminating " + getName()
                          + ".");
   }
}

public interface Buffer {
   public void set( int value );
   public int get();
}
```

```
Consumer reads -1
Consumer reads -1
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Consumer read values totaling: 1.
Terminating Consumer.
```

```
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.


SharedBufferTest
Consumer reads -1
Producer writes 1
Consumer reads 1
Producer writes 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer read values totaling: 7.
Terminating Consumer.
```

Synchronizing Threads

The following example uses the same Consumer and Producer threads as in the last example. However, this time we avoid the errors produced in the previous example.

```
public class SharedBufferTest2 {

   public static void main( String [] args )
   {
      SynchronizedBuffer sharedLocation = new
                                  SynchronizedBuffer();

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

      Producer producer = new Producer( sharedLocation );
      Consumer consumer = new Consumer( sharedLocation );

      producer.start();
      consumer.start();
   }
}
```

```
public class SynchronizedBuffer implements Buffer {

   private int buffer = -1;
   private int occupiedBufferCount = 0;

   public synchronized void set( int value )
   {
      String name = Thread.currentThread().getName();

      while ( occupiedBufferCount == 1 ) {

         try {
            System.err.println( name +
                                " tries to write." );
            displayState( "Buffer full. " + name +
                          " waits." );
            wait();
         }

         catch ( InterruptedException exception ) {
            exception.printStackTrace();
         }
      }
      buffer = value;

      ++occupiedBufferCount;

      displayState( name + " writes " + buffer );

      notify();
   }

   public synchronized int get()
   {
      String name = Thread.currentThread().getName();

      while ( occupiedBufferCount == 0 ) {

         try {
            System.err.println( name + " tries to read." );
            displayState( "Buffer empty. " + name
                          + " waits." );
            wait();
         }
         catch ( InterruptedException exception ) {
            exception.printStackTrace();
         }
```

```
      }
      --occupiedBufferCount;

      displayState( name + " reads " + buffer );

      notify();

      return buffer;

   }

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

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
Consumer reads 3	3	0

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

Producer writes 4        4                 1

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

Consumer read values totaling: 10.

```
Terminating Consumer.
```