CS 580 Client-Server Programming
Spring Semester, 2006
Doc 8 Threads 2
Feb 16, 2006

References

Cancellable Activities, Doug Lea, October 1998, http://gee.cs.oswego.edu/dl/cpj/cancel.html

Concurrent Programming in Java: Design Principles and Patterns, Doug Lea, Addison-Wesley, 1997

The Java Programming Language, 2nd Ed. Arnold & Gosling, Addison-Wesley, 1998

Java's Atomic Assignment, Art Jolin, Java Report, August 1998, pp 27-36.

Java 1.4.2 on-line documentation http://java.sun.com/j2se/1.4.2/docs/api/overview-summary.html

Java Network Programming 2nd Ed., Harold, O'Reilly, Chapter 5

Programming Ruby, 2nd Ed, Thomas

Java interrupt ()

Sent to a thread to interrupt it

If thread is blocked on a call to wait, join or sleep
- InterruptedException is thrown &
- The interrupted status flag is cleared

if the thread is blocked on I/O operation on an interruptible channel (NIO)
- ClosedByInterruptException is thrown
- The interrupted status flag is set

If the thread is blocked by a selector (NIO)
- Interrupt status is set
- The thread returns from the selector call as normal

If none of the other conditions hold then the thread's interrupt status is set

Details

If thread is blocked on a call to wait, join or sleep

- InterruptedException is thrown &
- The interrupted status flag is cleared

if the thread is blocked on I/O operation on an interruptible channel (NIO)

- ClosedByInterruptException is thrown
- The interrupted status flag is set

If the thread is blocked by a selector (NIO)

- Interrupt status is set
- The thread returns from the selector call as normal

If none of the other conditions hold then the thread's interrupt status is set

Interrupt and Pre JDK 1.4 NIO operations

If a thread is blocked on a read/write to a:

- Stream
- Reader/Writer
- Pre-JDK 1.4 style socket read/write

The interrupt does not interrupt the read/write operation!

The threads interrupt flag is set

Until the IO is complete the interrupt has no effect

This is one motivation for the NIO package

Interrupt does not stop a Thread

The following program does not end
The interrupt just sets the interrupt flag!

```
public class NoInterruptThread extends Thread {
        public void run() {
                while ( true) {
                        System.out.println(  "From: " + getName()  );
                }
        }

        public static void main(String args[]) throws InterruptedException{
                NoInterruptThread focused  = new NoInterruptThread( );
                focused.setPriority( 2 );
                focused.start();
                Thread.currentThread().sleep( 5 ); // Let other thread run
                focused.interrupt();
                System.out.println( "End of main");
        }
}
```

Output

```
From: Thread-0        (repeated many times)
End of main
From: Thread-0        (repeated until program is killed)
```

Using Thread.interrupted

```
public class RepeatableNiceThread extends Thread {
        public void run() {
                while ( true ) {
                        while ( !Thread.interrupted() )
                                System.out.println(  "From: " + getName()  );

                        System.out.println( "Clean up operations" );
                }
        }

        public static void main(String args[]) throws InterruptedException{
                RepeatableNiceThread  missManners  =
                                new RepeatableNiceThread( );
                missManners.setPriority( 2 );
                missManners.start();
                Thread.currentThread().sleep( 5 );
                missManners.interrupt();
        }
}
```

Output

```
From: Thread-0
Clean up operations
From: Thread-0
From: Thread-0 (repeated)
```

Interrupt and sleep, join & wait

```
public class  NiceThread  extends Thread {
        public void run() {
                try {
                        System.out.println( "Thread started");
                        while ( !isInterrupted() ) {
                                sleep( 5 );
                                System.out.println(  "From: " + getName()  );
                        }
                        System.out.println( "Clean up operations" );
                } catch ( InterruptedException interrupted ) {
                        System.out.println( "In catch" );
                }
        }

        public static void main( String args[] )  {
                NiceThread  missManners  = new NiceThread( );
                missManners.setPriority( 6 );
                missManners.start();
                missManners.interrupt();
        }
}
```

Output

```
Thread started
From: Thread-0
From: Thread-0
In catch
```

Safety - Mutual Access

Java Safety - Synchronize

A call to a synchronized method locks the object

Object remains locked until synchronized method is done

Any other thread's call to any synchronized method on the same object

will block until the object is unlocked

Java Safety - Synchronize

```
class SynchronizeExample {
        int[]  data;

        public String toString() {
                return "array length " + data.length + " array values " + data[0];
        }

        public synchronized void initialize( int size,  int  startValue){
                data  =  new  int[ size ];
                for  ( int  index  = 0; index < size;  index++ )
                        data[ index ]  =  (int ) Math.sin( index * startValue );
        }

        public  void unSafeSetValue( int  newValue) {
                for  ( int  index  = 0; index < data.length;  index++ )
                        data[ index ]  =  (int ) Math.sin( index * newValue );
        }

        public  synchronized void safeSetValue( int  newValue) {
                for  ( int  index  = 0; index < data.length;  index++ )
                        data[ index ]  =  (int ) Math.sin( index * newValue );
        }
}
```

Synchronized Static Methods

Locks class

Blocks other synchronized class methods

```
class SynchronizeExample {
        int[]  data;

        public String toString() {
                return "array length " + data.length + " array values " + data[0];
        }

        public synchronized void initialize( int size,  int  startValue){
                data  =  new  int[ size ];
                for  ( int  index  = 0; index < size;  index++ )
                        data[ index ]  =  (int ) Math.sin( index * startValue );
        }

        public  void unSafeSetValue( int  newValue) {
                for  ( int  index  = 0; index < data.length;  index++ )
                        data[ index ]  =  (int ) Math.sin( index * newValue );
        }

        public  synchronized void safeSetValue( int  newValue) {
                for  ( int  index  = 0; index < data.length;  index++ )
                        data[ index ]  =  (int ) Math.sin( index * newValue );
        }
}
```

Synchronized Statements

```
synchronized ( expression ) {
        statements
}
```

expression must evaluate to an object

That object is locked

```
class LockTest {
        public synchronized void enter() {
                System.out.println( "In enter");
        }
}
```

```
class LockTest {
        public void enter() {
                synchronized ( this ) {
                        System.out.println( "In enter");
                }
        }
}
```

Lock for Block and Method

```
public class LockExample extends Thread {
        private Lock myLock;

        public LockExample( Lock aLock ) {
                myLock = aLock;
        }
        public void run()       {
                System.out.println( "Start run");
                myLock.enter();
                System.out.println( "End run");
        }
        public static void main( String args[] ) throws Exception {
                Lock aLock = new Lock();
                LockExample tester = new LockExample( aLock );

                synchronized ( aLock ) {
                        System.out.println( "In Block");
                        tester.start();
                        System.out.println( "Before sleep");
                        Thread.currentThread().sleep( 5000);
                        System.out.println( "End Block");
                }
        }
}
```

```
class Lock {
        public synchronized void enter() {
                System.out.println( "In enter");
        }
}
```

Output

```
In Block
Start run
Before sleep
End Block
In enter
End run    (why is this at the end?)
```

Synchronized and Inheritance

```
class Top {
        public void synchronized left() {
                // do stuff
        }

        public void synchronized right() {
                // do stuff
        }
}
```

methods do not inherit synchronized

```
class Bottom extends Top {
        public void left() {
                // not synchronized
        }

        public void right() {
                // do stuff not synchronized
                super.right();     // synchronized here
                // do stuff not synchronized
        }
```

Ruby Synchronize

```
class Counter
  attr_reader :count
  def initialize
    @count = 0
    super
  end

  def tick
      @count += 1
  end
end

counter = Counter.new
tickA = Thread.new { 10000.times { counter.tick}}
tickB = Thread.new { 10000.times { counter.tick}}
tickA.join
tickB.join
puts counter.count -> 14451
```

```
require 'monitor'
class Counter < Monitor
  attr_reader :count
  def initialize
    @count = 0
    super
  end

  def tick
    synchronize do
      @count += 1
    end
  end
end

counter = Counter.new
tickA = Thread.new { 10000.times { counter.tick}}
tickB = Thread.new { 10000.times { counter.tick}}
tickA.join
tickB.join
puts counter.count -> 20000
```

Ruby Synchronize without inheritance

```
require 'monitor'

class Counter
  include MonitorMixin
  attr_reader :count
  def initialize
    @count = 0
    super
  end
  def tick
    synchronize do
      @count += 1
    end
  end
end
```

Ruby Synchronize examples from
Programming Ruby, 2nd Ed, Thomas, pp 142-144

Using Monitor directly

```
require 'monitor'

class Counter
  attr_reader :count
  def initialize
    @count = 0
    super
  end
  def tick
    @count += 1
  end
end

counter = Counter.new
lock = Monitor.new
tickA = Thread.new { 10000.times { lock.synchronize {counter.tick}}}
tickB = Thread.new { 10000.times { lock.synchronize {counter.tick}}}
tickA.join
tickB.join
puts counter.count -> 20000
```

wait and notify

public final void wait(timeout) throws InterruptedException
public final void wait(timeout, nanos) throws InterruptedException
public final void wait() throws InterruptedException

Causes a thread to wait until it is notified or the specified timeout expires.

Throws: IllegalMonitorStateException
If the current thread is not the owner of the Object's monitor.

Throws: InterruptedException
Another thread has interrupted this thread.

public final void notify()
public final void notifyAll()

Notifies threads waiting for a condition to change.

wait - How to use

The thread waiting for a condition should look like:

```
synchronized void waitingMethod()
        {
        while ( ! condition )
                wait();

        Now do what you need to do when condition is true
        }
```

Everything is executed in a synchronized method

The test condition is in loop not in an if statement

The wait suspends the thread it atomically releases the lock on the object

notify - How to Use

```
synchronized void changeMethod()
        {
        Change some value used in a condition test

        notify();
        }
```

wait and notify Example

When can Consumer read from queue?

wait and notify SharedQueue

```
import java.util.ArrayList;

public class SharedQueue {
        ArrayList elements = new ArrayList();
        public synchronized void append( Object item ) {
                elements.add( item);
                notify();
        }

        public synchronized Object get( ) {
                try {
                        while ( elements.isEmpty() )
                                wait();
                }
                catch (InterruptedException threadIsDone ) {
                        return null;
                }
                return elements.remove( 0);
        }
}
```

wait and notify - Producer

```
public class Producer extends Thread {
        SharedQueue factory;
        int workSpeed;

        public Producer( String name, SharedQueue output, int speed ) {
                setName(name);
                factory = output;
                workSpeed = speed;
        }

        public void run() {
                try {
                        int product = 0;
                        while (true) // work forever {
                                System.out.println( getName() + " produced " + product);
                                factory.append( getName() + String.valueOf( product) );
                                product++;
                                sleep( workSpeed);
                        }
                } catch ( InterruptedException WorkedToDeath ) {
                        return;
                }
        }
}
```

wait and notify - Consumer

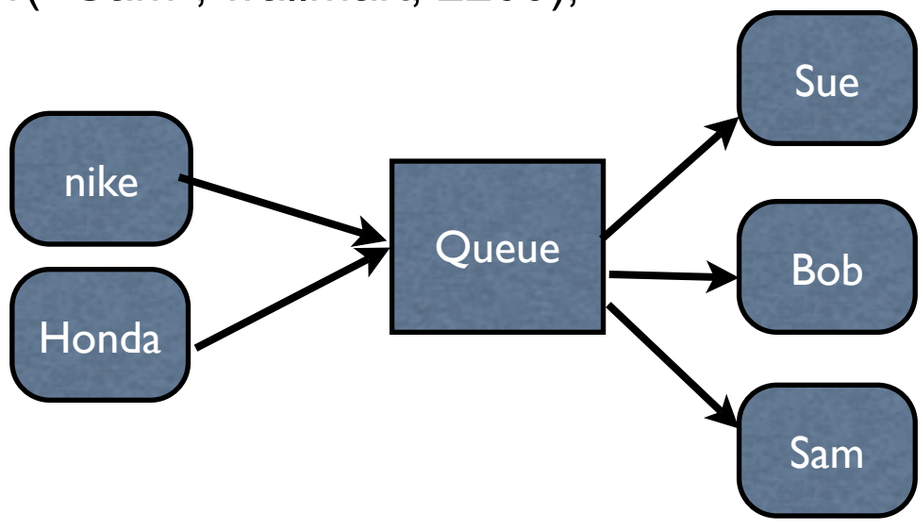
```
class Consumer extends Thread {
        Queue localMall;
        int sleepDuration;

        public Consumer( String name, Queue input, int speed ) {
                setName(name);
                localMall = input;
                sleepDuration = speed;
        }

        public void run() {
                try {
                        while (true) // Shop until you drop {
                                System.out.println( getName() + " got " +  localMall.get());
                                sleep( sleepDuration );
                        }
                }
                catch ( InterruptedException endOfCreditCard ) {
                        return;
                }
        }
}
```

wait and notify - Driver Program

```
public class ProducerConsumerExample {
        public  static  void  main( String  args[] ) throws Exception {
                SharedQueue wallmart = new SharedQueue();
                Producer nike = new Producer( "Nike", wallmart, 500 );
                Producer honda = new Producer( "Honda", wallmart, 1200 );
                Consumer valleyGirl = new Consumer( "Sue", wallmart, 400);
                Consumer valleyBoy = new Consumer( "Bob", wallmart, 900);
                Consumer dink = new Consumer( "Sam", wallmart, 2200);
                nike.start();
                honda.start();
                valleyGirl.start();
                valleyBoy.start();
                dink.start();
        }
}
```



Nike produced 0	Nike produced 2	Nike produced 4
Honda produced 0	Sue got Nike2	Sue got Nike4
Sue got Nike0	Honda produced 1	Honda produced
Bob got Honda0	Bob got Honda1	Bob got Honda2
Nike produced 1	Nike produced 3	Nike produced 5
Sam got Nike1	Sue got Nike3	Sue got Nike5

Ruby Producers & Consumers

```
require 'thread'

queue = Queue.new
consumers = (1..3).collect do |each|
  Thread.new("Consumer #{each}") do |name|
    begin
      product = queue.deq
      puts "#{name}: consumed #{product}"
      sleep(rand(0.05))
    end until product == :END_OF_WORK
  end
end


producers = (1..2).collect do |each|
  Thread.new("Producer #{each}") do |name|
    3.times do |k|
      sleep(0.1)
      queue.enq("Item #{k} from #{name}")
    end
  end
end

producers.each { |each| each.join }
consumers.size.times { queue.enq(:END_OF_WORK)}
consumers.each { |each| each.join }
```

Output

```
Consumer 1: consumed Item 0 from Producer 1
Consumer 2: consumed Item 0 from Producer 2
Consumer 3: consumed Item 1 from Producer 1
Consumer 2: consumed Item 1 from Producer 2
Consumer 3: consumed Item 2 from Producer 1
Consumer 1: consumed Item 2 from Producer 2
Consumer 1: consumed END_OF_WORK
Consumer 2: consumed END_OF_WORK
Consumer 3: consumed END_OF_WORK
```

Example from
Programming Ruby, 2nd Ed, Thomas, pp 743