## CS 580 Client-Server Programming

### Spring Semester, 2004

**Doc 17 Threads part 2**

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Control</td>
<td>3</td>
</tr>
<tr>
<td>Java interrupt()</td>
<td>3</td>
</tr>
<tr>
<td>Smalltalk interrupt</td>
<td>10</td>
</tr>
<tr>
<td>Safety - Mutual Access</td>
<td>11</td>
</tr>
<tr>
<td>Java Safety - Synchronize</td>
<td>12</td>
</tr>
<tr>
<td>Synchronized Instance Methods</td>
<td>12</td>
</tr>
<tr>
<td>Synchronized Static Methods</td>
<td>13</td>
</tr>
<tr>
<td>Synchronized and Inheritance</td>
<td>16</td>
</tr>
<tr>
<td>wait and notify Methods in Object</td>
<td>17</td>
</tr>
<tr>
<td>Smalltalk Safety – Semaphore &amp; RecursionLock</td>
<td>24</td>
</tr>
<tr>
<td>Some Thread Issues &amp; Ideas</td>
<td>27</td>
</tr>
<tr>
<td>Passing Data – Multiple Thread Access</td>
<td>27</td>
</tr>
<tr>
<td>Pass copies</td>
<td>28</td>
</tr>
<tr>
<td>Immutable Objects</td>
<td>29</td>
</tr>
<tr>
<td>Background Operations</td>
<td>30</td>
</tr>
<tr>
<td>Futures</td>
<td>31</td>
</tr>
<tr>
<td>Callbacks</td>
<td>33</td>
</tr>
</tbody>
</table>

### References


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


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

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

VisualWorks Application Developers Guide, Cincom, Chapter 11 (Smalltalk), see docs/AppDevGuide.pdf in the VW installation

**Copyright ©, All rights reserved.2004 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent ([http://www.opencontent.org/opl.shtml](http://www.opencontent.org/opl.shtml)) license defines the copyright on this document.**
Source Code In Lecture

Smalltalk
See course Store repository package ThreadLecture1
version 1.4
Thread Control
Java interrupt()

Sent to a thread to interrupt it

A thread has interrupted status flag

JDK 1.4 Doc state

InterruptedException is thrown if thread is blocked a call to:
• wait
• join
• sleep
and the interrupted status flag is cleared

ClosedByInterruptException is thrown if the thread is blocked
• I/O operation on an interruptible channel
and the interrupted status flag is set

Interruptible channels are part of JDK 1.4 NIO package

If the thread is blocked by a selector:
• 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!

```java
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

This example uses the test Thread.interrupted() to allow the thread to be continue execution later.

```java
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

Let thread A be in the not runnable state due to being sent either the sleep(), join() or wait() methods. Then if thread A is sent the interrupt() method, it is moved to the runnable state and InterruptedException is raised in thread A.

In the example below, NiceThread puts itself to sleep. While asleep it is sent the interrupt() method. The code then executes the catch block.

```java
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
Who Sends sleep() is Important

Since main sends the sleep method, not the thread itself, the InterruptedException is not thrown.

```java
public class WhoSendsSleep extends Thread {
    public void run() {
        try {
            while ( !isInterrupted() ) {
                System.out.println( "From: " + getName() );
            }
            System.out.println( "Clean up operations" );
        } catch ( Exception interrupted ) {
            System.out.println( "In catch" );
        }
    }
}

public static void main( String args[] ) {
    try {
        NiceThread missManners = new NiceThread( );
        missManners.setPriority( 1 );
        missManners.start();
        missManners.sleep( 50 );       //Which thread is sleeping?
        missManners.interrupt();
    } catch ( InterruptedException interrupted ) {
        System.out.println( "Caught napping" );
    }
}
```

Output

Thread started
From: Thread-0
From: Thread-0
Clean up operations
Threads & Method Sends

A method is executed in the thread that sends the method

missManners.sleep( 50);

Put the current thread to sleep not the missManners thread
Smalltalk interrupt

interruptWith: aBlock
  Force the receiver to interrupt whatever it is doing and to evaluate aBlock
  Receiver continues what ever it was doing after evaluating aBlock

| process result |
result := WriteStream on: String new.
process :=
  [1 to: 5000
   do:
     [:each |
       result
       nextPutAll: each printString;
       cr.
       Processor activeProcess yield]]
fork.
(Delay forMilliseconds: 1) wait.
process interruptWith: [result nextPutAll: 'Interrupt'].
(Delay forMilliseconds: 100) wait.
Transcript
clear;
show: result contents

Output

The numbers then ‘Interrupt’ then the rest of the numbers.
Safety - Mutual Access

With multiprocessing we need to address mutual access by different threads. When two or more threads simultaneously access the same data there may be problems.

Some types of access are safe. If a method accesses just local data, then multiple threads can safely call the method on the same object. Assignment statements of all types, except long and double, are atomic. That is a thread can not be interrupted by another thread while performing an atomic operation.
Java Safety - Synchronize

Synchronize is Java's mechanism to insure that only one thread at a time will access a piece of code. We can synchronize methods and block's of code (synchronize statements).

Synchronized Instance Methods
When a thread executes a synchronized instance method on an object, that object is locked. The object is locked until the method ends. No other thread can execute any synchronized instance method on that object until the lock is released. The thread that has the lock can execute multiple synchronized methods on the same object. The synchronization is on a per object bases. If you have two objects, then different threads can simultaneously execute synchronized methods on different objects.

Unsynchronized methods can be executed on a locked object by any thread at any time. The JVM insures that only one thread can obtain a lock on an object at a time.

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

A synchronized static method creates a lock on the class, not the object. When one thread has a lock on the class, no other thread can execute any synchronized static method of that class. Other threads can execute synchronized instance methods on objects of that class.

class SynchronizeStaticExample {
    int[] data;
    static int[] classData;

    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 synchronized void initializeStatic( int size, int startValue){
        classData = new int[size];
        for (int index = 0; index < size; index++)
            classData[ index ] = (int) Math.sin( index * startValue );
    }
}
Synchronized Statements

A block of code can be synchronized. The basic syntax is:

```java
synchronized ( expr ) {
    statements
}
```

The `expr` must evaluate to an object. This will lock the object. The lock is released when the thread finishes the block. Until the lock is released, no other thread can enter any method or synchronized block that is locked by the given object.

A synchronized method is syntactic sugar for a synchronized block.

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

Is the same as:

```java
class LockTest {
    public void enter() {
        synchronized ( this ) {
            System.out.println( "In enter" );
        }
    }
}
```
Lock for Block and Method

This example shows that a lock on an object also locks all access to the object via synchronized methods.

```java
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 does this come at the end?)
Synchronized and Inheritance

If you want a method in a subclass to be synchronized you must declare it to be synchronized.

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

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

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

    public void right()
    {
        // do stuff not synchronized
        super.right(); // synchronized here
        // do stuff not synchronized
    }
}
**wait and notify Methods in Object**

wait and notify are some of the most useful thread operations.

```java
public final void wait(timeout) throws InterruptedException

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

**Parameters:**
- timeout - the maximum time to wait in milliseconds

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

**Throws:** InterruptedException
- Another thread has interrupted this thread.
```

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

public final void wait() throws InterruptedException

public final void notify()

public final void notifyAll()
```

Notifies all of the threads waiting for a condition to change. Threads that are waiting are generally waiting for another thread to change some condition. Thus, the thread effecting a change that more than one thread is waiting for notifies all the waiting threads using the method notifyAll(). Threads that want to wait for a condition to change before proceeding can call wait(). The method notifyAll() can only be called from within a synchronized method.
wait - How to use

The thread waiting for a condition should look like:

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

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

**Note**

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**

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

    notify();
}
```
**wait and notify Example**

Over the next five slides is a typical consumer-producer example. Producers "make" items, which they put into a queue. Consumers remove items from the queue. What happens when the consumer wishes to remove when the queue is empty? Using threads, we can have the consumer thread wait until a producer thread adds items to the queue.

```java
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

```java
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;
        }
    }
}
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();
    }
}

Output

<table>
<thead>
<tr>
<th>Nike produced 0</th>
<th>Sue got Nike3</th>
<th>Honda produced 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Honda produced 0</td>
<td>Nike produced 4</td>
<td>Bob got Honda3</td>
</tr>
<tr>
<td>Sue got Nike0</td>
<td>Sue got Nike4</td>
<td>Nike produced 8</td>
</tr>
<tr>
<td>Bob got Honda0</td>
<td>Honda produced</td>
<td>Sue got Nike8</td>
</tr>
<tr>
<td>Nike produced 1</td>
<td>Bob got Honda2</td>
<td>Nike produced 9</td>
</tr>
<tr>
<td>Sam got Nike1</td>
<td>Nike produced 5</td>
<td>Sue got Nike9</td>
</tr>
<tr>
<td>Nike produced 2</td>
<td>Sue got Nike5</td>
<td>Honda produced 4</td>
</tr>
<tr>
<td>Sue got Nike2</td>
<td>Nike produced 6</td>
<td>Bob got Honda4</td>
</tr>
<tr>
<td>Honda produced 1</td>
<td>Sam got Nike6</td>
<td>Nike produced 10</td>
</tr>
<tr>
<td>Bob got Honda1</td>
<td>Nike produced 7</td>
<td>Sue got Nike10</td>
</tr>
<tr>
<td>Nike produced 3</td>
<td>Sue got Nike7</td>
<td>Nike produced 11</td>
</tr>
</tbody>
</table>
Smalltalk Safety – Semaphore & RecursionLock

VisualWorks implements a Semaphore

**Important Methods**

Semaphore>>wait
Causes the process to block until it is signaled

Semaphore>>signal
If processes are waiting allow the first one to proceed.
If no process is waiting, remember the excess signal.

Semaphore>>critical: aBlock
If not currently executing another critical: or wait run aBlock
Otherwise wait until other critical: ends or signal to end wait

Normally not mixed with wait & signal

Use Semaphore forMutualExclusion to create receiver for critical:

**RecursionLock**

Allows calls to critical: inside critical:

lock critical: [lock critical: [Transcript cr; show: 'entered']]

If lock is a Semaphore above will deadlock

If lock is a RecursionLock the above will work correctly
Simple Semaphore Example

| lock consumer result |
lock := Semaphore new.
result := WriteStream on: String new.
consumer :=
   [:name |
      5 timesRepeat:
         [lock wait.
            result
            nextPutAll: name, ' got it';
            cr.
            Processor activeProcess yield]].
(consumer newProcessWithArguments: #('A')) resume.
(consumer newProcessWithArguments: #('B')) resume.

[5 timesRepeat:
   [result
      nextPutAll: 'Produced 2';
      cr.
      lock
      signal;
      signal.
      Processor activeProcess yield]]
fork.
(Delay forMilliseconds: 500) wait.
Transcript
   clear;
   show: result contents
Output

Produced 2
A got it
B got it
Produced 2
A got it
B got it
Produced 2
A got it
B got it
(pattern occurs a total of 5 times)
Some Thread Issues & Ideas
Passing Data – Multiple Thread Access

Situation

An object is passed between threads

Issue

If multiple threads have a reference to the same object

When one thread changes the object the change is global

Example

anObject = anotherThreadObject.getFoo();  // line A
System.out.println( anObject);            // line B

If multiple threads have access to anObject

The state of anObject can change after line A ends and before line B starts!

This can cause debugging nightmares
Passing Data – Multiple Thread Access

Possible Solutions

Pass copies

Returning data

```java
public foo getFoo() {
    return foo.clone();
}
```

Parameters

```java
anObject.doSomeMunging( bar.clone());
```

```java
anObject doSomeMunging: bar copy
```
Passing Data – Multiple Thread Access
Possible Solutions

Immutable Objects

Pass objects that cannot change

Java’s base type and Strings are immutable

In VisualWorks 7 any object can be made immutable

anObject beImmutable
Background Operations

Situation

Perform some operation in the background
At same time perform some operations in the foreground
Need to get the result when operation is done

Issue

Don’t make the code sequential

Avoid polling

```java
class Poll {
    public static void main( String args[] ) {
        TimeConsumingOperation background = new TimeConsumingOperation();
        background.start();

        while ( !background.isDone() ) {
            performSomethingElse;
        }

        Object neededInfo = background.getResult();
    }
}
```
Futures

A future starts a computation in a thread
When you need the result ask the future
You will block if the result is not ready

Smalltalk

Promise class in VisualWorks

| delayedAnswer realAnswer |
delayedAnswer := [aClient perform: 'computePi'] promise.
Do some other work here
realAnswer := delayedAnswer value
Sample Java Future

class FutureWrapper {
    TimeConsumingOperation myOperation;

    public FutureWrapper() {
        myOperation = new TimeConsumingOperation();
        myOperation.start();
    }

    public Object value() {
        try {
            myOperation.join();
            return myOperation.getResult();
        } catch (InterruptedException trouble) {
            DoWhatIsCorrectForYourApplication;
        }
    }
}

public class FutureExample {
    public static void main( String args[] ) {

        FutureWrapper myWorker = new FutureWrapper();

        DoSomeStuff;
        DoMoreStuff;

        x = myWorker.value();
    }
}
**Callbacks**

Have the background thread call a method when it is done

**Java Outline**

class MasterThread {
    public void normalCallback( Object result ) {
        processResult;
    }

    public void someMethod() {
        compute;
        TimeConsumingOperation backGround =
            new TimeConsumingOperation( this );
        backGround.start();
        moreComputation;
    }
}

class TimeConsumingOperation extends Thread {
    MasterThread master;

    public TimeConsumingOperation( MasterThread aMaster ) {
        master = aMaster;
    }

    public void run() {
        DownloadSomeData;
        PerformSomeComplexStuff;
        master.normalCallback( resultOfMyWork );
    }
}
Smalltalk Outline

ClientCode>>startBackground
   [| result |
   result := anObject performSomeCalculation.
   self processResult: result] fork

ClientCode>>processResult: aResult
   Handle the result here

ClientCode>>aMethod
   Do some work
   self startBackground.
   Do more work