CS 580 Client-Server Programming
Spring Semester, 2006
Doc 18 Some Thread Issues
Apr 6, 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

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

Java Performance and Scalability Vol. 1, Dov Bulka, 2000

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 – Possible Solutions

Pass copies

Returning data

```
pubic foo getFoo() {
    return foo.clone();
}
```

Parameters

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

Passing Data – Possible Solutions

Immutable Objects

Pass objects that cannot change

Java's base types (Integer) and Strings are immutable

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

```
public 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

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();
    }
}
```

java.util.concurrent.FutureTask<V> (JDK 1.5)

```
import java.util.concurrent.*;

public class SimpleRunable<V> implements Callable<V> {
        V fakeResult;

        public SimpleRunable(V x) {
                fakeResult = x;
        }

        public V call() throws Exception {
                Thread.sleep((long)1000);
                return fakeResult;
         }
}

        public void testFuture() throws InterruptedException, ExecutionException {
                Callable<String> faked = new SimpleRunable<String>("go");
                FutureTask<String> example = new FutureTask(faked);
                example.run();
                assertTrue( "go" == example.get());
        }
```

Callbacks

Have the background thread call a method when it is done

```
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 );
    }
}
```

Thread Pools - Some Background

Iterative Server

```
while (true)
        {
        Socket client = serverSocket.accept();
        Sequential code to handle request
        }
```

When usable

TP = Time to process a request

A = arrival time between two consecutive requests

Then we need TP << A

Thread Pools - Some Background

Basic Concurrent Server

```
while (true)
        {
        Socket client = serverSocket.accept();
        Create a new thread to handle request
        }
```

When usable

Let TC = time to create a thread

Let A = arrival time between two consecutive requests

We need TC << A

Often this is good enough

Problem with Threads

Thread consume resources

- Memory
- CPU cycles

A program has a limit of

- Threads it can productively support
- Sockets it can have open

We need to insure we don’t create too many threads

Some Timing Results

VisualWorks Smalltalk				
	Iterations or Number Created (n)			
	100	1,000	10,000	100,000
Empty Loop	0	0	0	1
Integer add	0	0	0	3
Collection create	0	0	4	34
Thread create	1	5	45	380
Thread create & run	1	5	152	1268
Thread create & run	0	3	82	1048

Ruby				
	Iterations or Number Created (n)			
	100	1,000	10,000	100,000
Empty Loop	0	1	10	83
Integer add	0	3	44	248
Collection create	0	4	53	356
Thread create & run	38	289	2153	21526
Thread create & run	37	268	2116	22298

Java				
	Iterations or Number Created (n)			
	100	1,000	10,000	100,000
Empty Loop	0	0	1	7
Integer add	0	0	1	7
Vector create	0	5	10	42
Thread create	2	62	185	1771
Thread create & run	40	402	2626	24909
Thread create & run	51	351	2507	24767

Macintosh PowerBook with 1.25GHz PowerPC processor
OS 10.3.3 (OS 10.4.3 for Ruby)
VW 7.2nc
Java 1.4.2_03 with HotSpot Client
Ruby 1.8

VisualWorks Details

Loop	n timesRepeat:[ ]
Integer add	n timesRepeat:[x := 3 +4]
Collection create	n timesRepeat:[x := OrderedCollection new]
Thread create	n timesRepeat:[[x := 3 +4] newProcess]
Thread create & run	n timesRepeat:[[x := 3 +4] fork]

Code used

```
Transcript clear.
#(100 1000 10000 100000) do:
    [:n |
    | x |
ObjectMemory garbageCollect.
time := Time millisecondsToRun:
  [n timesRepeat: [[x := 3 + 4] fork]].
Transcript
  print: n;
  tab;
  print: time;
  tab;
  print: x;
  cr;
  flush]
```

Java Code Timed

Loop	for (int k = 0; k < n; k++){
	}

Integer add	for (int k = 0; k < n; k++){
	x = 3 + 4
	}

Collection create	for (int k = 0; k < n; k++){
	x = new Vector();
	}

Thread create	for (int k = 0; k < n; k++){
	x = new SampleThread();
	}

Thread create & run	for (int k = 0; k < n; k++){
	x = new SampleThread();
	x.start();
	}

Java Program Used

```
import java.util.*;
import sdsu.util.Timer;

public class TimeTests {
  public static void main (String args[]) {
    Timer clock = new Timer();
    SampleThread x = new SampleThread();

    for (int n = 100; n < 200000; n = n * 10) {
      System.gc();
      clock.start();
      for (int k = 0; k < n; k++){
        x = new SampleThread();
        x.start();
      }
      long time = clock.stop();
      x.x();
      System.out.println("" +  n + "\t" + time);
      clock.reset();
    }
  }
}
```

```
class SampleThread extends Thread {
      int x;
      public void run() {
            x = 3 + 4;
      }

      public int x() {
            return x;
      }
}
```

Ruby Code

```
[100, 1000, 10000, 100000].each do |size|
    start = Time.now
    size.times do
        x = Thread.new do
            x = 3 + 4
        end
    end
    endTime = Time.now

    puts ((endTime - start)*1000).round
end
```

Empty Loop

```
size.times do
end
```

Collection Creation

```
start = Time.now
x = 4
size.times do
   x = Array.new
end
endTime = Time.now
x[0] = 4
```

Warning about Micro-benchmarks

Micro-benchmarks are

- Hard to do well
- Misleading

Better to measure the performance of your system

Concurrent Server With Thread Pool

```
Create N worker threads
while (true)
        {
        Socket client = serverSocket.accept();
        Use an existing worker thread to handle request
        }
```

When usable

TP = Time to process a request

A = arrival time between two consecutive requests

N = Thread Pool size

Then we need TP << A * N

Concurrent Server With Thread Pool & Thread Creation

```
Create N worker threads
while (true)
        {
        Socket client = serverSocket.accept();
        if worker thread is idle
                Use an existing worker thread to handle request
        else
                create new worker thread to handle the request
        }
```

When usable

Number of requests we can handle at a unit of time

$$TP / N + 1/TC$$

where N is not constant

What to do with the new Worker Threads?

Client requests are not constant over time

Requests can come in bursts

Threads consume resources

Don’t want a large pool of threads sitting idle

Common strategy

Have a minimum number of threads in a pool

When needed add threads to the pool up to some maximum

When traffic slows down remove idle threads

Threads & Memory Cache

Threads require a fair amount of memory (why?)

Virtual memory divides memory into pages

A page may be in
- Memory
- Memory Cache
- Disk Cache
- Disk

Access to a page is faster if it is in memory

Last thread that completed is likely to be in memory or cache

Reusing last thread that complete can improve performance

Which Should I use?

Which method to use?

Which values (number of threads, etc) to use?

Depends on your

- Application
- Implementation
- Hardware
- Performance requirements

How to reuse a Thread?

Classic idea

Server places client requests in a queue

Worker repeats forever
- Read request from queue
- Process request

Queue

- Block on read if queue is empty
- Signals waiting threads when data is added

Java Example

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);
      }

public int size()
   {
   return elements.size();
   }
}
```

DateHandler

```
import java.net.*;
import java.io.*;
import java.util.Date;

public class DateHandler extends Thread
  {
  SharedQueue workQueue;

  public DateHandler(SharedQueue workSource )
    {
    workQueue = workSource;
    }

public void run()
  {
  while (!isInterrupted() )
    try
      {
      Socket client = (Socket) workQueue.get();
      processRequest(client);
      }
    catch (Exception error )
      {
      /* log error*/
      }
  }

void processRequest(Socket client) throws IOException
    {
    try
      {
      client.setSoTimeout( 10 * 1000 );
      processRequest(
        client.getInputStream(),
        client.getOutputStream());
      }
    finally
      {
      client.close();
      }
    }

void processRequest(InputStream in, OutputStream out)
    throws IOException
  {
  BufferedReader parsedInput =
      new BufferedReader(new InputStreamReader(in));
  PrintWriter parsedOutput = new PrintWriter(out,true);
  String inputLine = parsedInput.readLine();
  if (inputLine.startsWith("date"))
    {
    Date now = new Date();
    parsedOutput.println(now.toString());
    }
  }
}
```

DateServer

```
import java.util.*;
import java.net.*;
import java.io.*;

public class DateServer {
   SharedQueue workQueue;
   ServerSocket listeningSocket;
   ArrayList workers = new ArrayList();

   public static void main( String[] args ) {
      System.out.println( "Starting");
      new DateServer( 33333).run();
   }

public void run() {
      Socket client = null;
      while (true) {
         try {
            client = listeningSocket.accept();
            workQueue.append( client);
         } catch (IOException acceptError){
            // need to log error and make sure client is closed
            }
         }
      }
   }
```

```
public DateServer( int port ) {
      try {
         listeningSocket = new ServerSocket(port);
         workQueue = new SharedQueue();
         for (int k = 0; k < 5; k++) {
            Thread worker = new DateHandler( workQueue);
            worker.start();
            workers.add( worker);
         }
      } catch (IOException socketCreateError) {
         //log and exit here
         }
   }
```