CS 580 Client-Server Programming
Fall Semester, 2002
Doc 19 Thread Pools
Contents

References

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

Past CS580 lecture notes

Thread Pools
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

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

Thread consume resources

- Memory
- CPU cycles

A machine has a limit of threads it can productively support

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

Thread Creation Time - VisualWorks

Time in milliseconds

	Iterations or Number Created (n)			
	100	1,000	10,000	100,000
Loop	0	0	0	4
Integer add	0	0	1	4
Object create	0	0	1	17
Collection create	0	0	7	51
Thread create	1	8	70	695
Thread create & run	1	12	304	2893

Statements Timed

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

Run on

- Macintosh PowerBook with 400MHz PowerPC processor
- OS 10.2
- VW 7 with beta version of VM for OS 10

Garbage collection was run between timing of each statement

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

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

VisualWorks Example

```
Smalltalk defineClass: #ThreadedDateServer
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'serverSocket workers workQueue '
   classInstanceVariableNames: ''
   imports: ''
   category: 'SimpleServer'
```

ThreadedDateServer class methods

```
port: anInteger
   ^super new setPort: anInteger
```

ThreadedDateServer instance methods

```
setPort: anInteger
   serverSocket := SocketAccessor newTCPserverAtPort: anInteger.
   serverSocket
      listenFor: 4;
      soReuseaddr: true.
   workQueue := SharedQueue new.
   workers := OrderedCollection new.
   5 timesRepeat: [workers add: self createWorker]

createWorker
   ^[[self processRequestOn: workQueue next] repeat] fork
```

Example Continued

```
processRequestOn: aSocket
  | clientRequest clientIOStream |

  [(aSocket readWaitWithTimeoutMs: 10000) ifTrue: [^nil].
  clientIOStream := aSocket readAppendStream.
  clientIOStream lineEndTransparent.
  clientRequest := clientIOStream through: Character cr.
  (clientRequest startsWith: 'date')
    ifTrue:
      [clientIOStream
        nextPutAll: Time dateAndTimeNow printString;
        cr;
        nextPut: Character lf;
        commit].
  clientIOStream close]
      ensure: [aSocket close]

start
  [serverSocket isActive] whileTrue:
      [workQueue nextPut: serverSocket accept.
      workQueue size > 2 ifTrue: [workers add: self createWorker]]

shutdown
  serverSocket close
```

Issues not addressed

How big to make the thread pool
Setting priority of threads
Using a thread for the server
Maximum number of threads to allow
Making values configuration settings
How to reduce the number of threads
Separating the Server from parsing the protocol

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*/
        }
    }
```

DateHandler Continued

```
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();
      }
```

DateServer Continued

```
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
    }
  }
```

DateServer Continued

```
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
         }
      }
   }
}
```

How to Remove Extra Workers

Let each worker above the minimum end after K requests

When system is idle have reaper thread remove some workers

Have get() on shared queue time out

When workers get time out on the queue, they exit