Hung Thread Detection

What this exercise is about

The objective of this lab is to introduce you to the hung thread detection features of WebSphere Application Server v6.0. It will also introduce you JMX notifications and startup beans, and how they can be used to aid in problem determination.

Lab Requirements

- WebSphere Application Server v6.0 should be installed
- The lab source files (LabFiles60.zip) must be extracted to the root directory (i.e., C:\) if you want to look at the source
- An application server configured named **C6Server01**

What you should be able to do

At the end of this lab you should be able to:

- Configure WebSphere Application Server's hung thread detection feature
- Interpret the log messages that relate to hung threads
- Enable startup beans, and understand the basics of writing a startup bean

- Find a simple deadlock in a java thread dump

Introduction

In this lab you will learn to configure WebSphere Application Server's hang detection policy by setting the ThreadMonitor interval (how often to search for hung threads), the ThreadMonitor threshold (how long a thread can run before being marked "hung"), and the false alarm threshold (how many threads can be mistakenly marked as hung before increasing the ThreadMonitor threshold).You will then run an example Servlet that will exhibit a hang, so that you can see the output that is generated when a hang is detected.

After walking through this simple example, you will then see how the flexibility of this feature can be exploited to create a more robust problem determination mechanism, using JMX and startup beans.

Exercise Instructions

Part 1: Installing the test application

____ 1. First, start the application server, so you can install the application

__ a. Open a command prompt on your thinkpad and telnet into your system:

```
telnet mvs22x.rtp.raleigh.ibm.com 1023

userid:  wsuser, pw:  mo1nday
```

__ b. Navigate to the appropriate directory by typing :

```
cd /etc/c6cellB/AppServer/profiles/default/bin
```

__ c. Then issue the command

```
./startServer.sh C6Server01
```

____ 2. Install the HangLabApp enterprise application by issuing a Jython command with wsadmin

__ a. Type the command below in your command prompt (all on one line)

```
./wsadmin.sh -lang jython -conntype NONE -c
"AdminApp.install('/etc/LabFiles60/HungThreadLab/HangLabApp.ear','[-
server C6Server01]')"
```

____ 3. Leave your telnet session open.

Part 2: Configuring the Hang Detection Policy

Hung thread detection is enabled by default in WebSphere Application Server. In the default configuration, a thread must be active for at least ten minutes before it is marked as potentially hung. In the interest of time, you will lower that threshold to two minutes by setting a custom property. You will also set another property to change the interval at which the ThreadMonitor polls for hung threads from three minutes to twenty seconds.

____ 4. Log into the Administrative Console.

__ a. Open a web browser and navigate to the following URL:

http://mvs22x.rtp.raleigh.ibm.com:9080/ibm/console

__ b. In the box marked “User ID” type **wsdemo** and click **Log In**.

____ 5. Open the Custom Properties panel for your application server in the Administrative Console.

__ a. Using the Console’s left-side navigation menu, expand **Servers**, and click **Application Servers**.

__ b. In the resulting table, click **C6Server01**.

__ c. On the right side of the panel, expand **Administration**, which you will find listed under the Server Infrastructure heading. You may have to scroll down to see this.

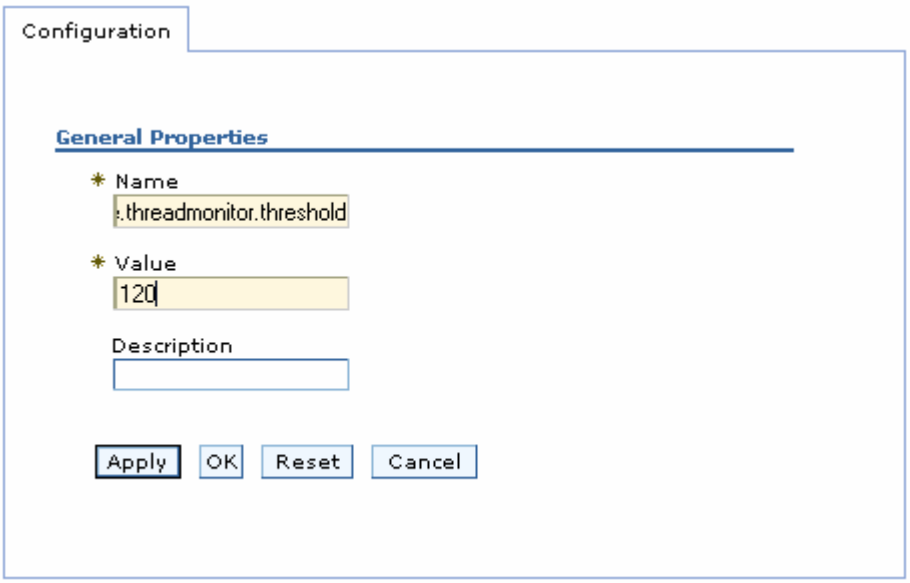
__ d. Under Administration, click **Custom Properties**.

____ 6. Create a custom property with the name **com.ibm.websphere.threadmonitor.threshold** and a value of **120**, as shown in the graphic below. This will tell the ThreadMonitor to mark any thread active for longer than two minutes as hung.

__ a. In the Custom Properties panel, click **New**.

__ b. Under Name, enter **com.ibm.websphere.threadmonitor.threshold**

__ c. Under Value, enter **120**.



__ d. Click **OK**.

____ 7. Create a custom property with the name **com.ibm.websphere.threadmonitor.interval** and a value of **20**, as shown in the graphic below. This will tell the ThreadMonitor to look for hung threads every twenty seconds.

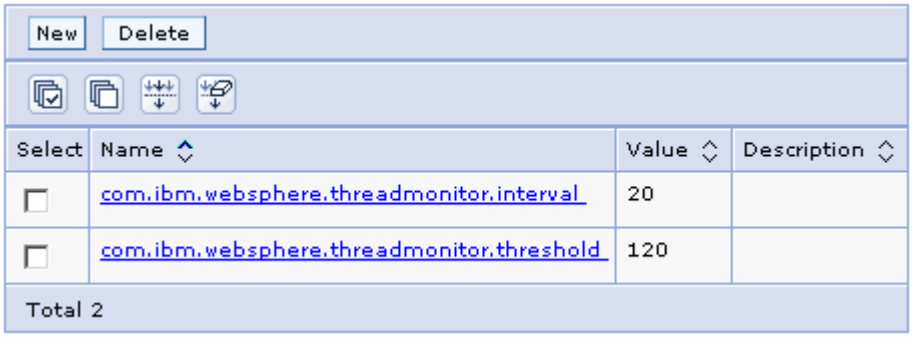
__ a. In the Custom Properties panel, click **New**.

__ b. Under Name, enter **com.ibm.websphere.threadmonitor.interval.**

__ c. Under Value, enter **20**.

__ d. Click **OK**.

____ 8. After clicking OK, your Custom Properties list should look like the graphic shown here.



____ 9. Set the **protocol_http_timeout_output** value to 0 to prevent an EC3 abend from happening first.

__ a. In the Environment->WebSphere Variables panel

__ b. Click on **C6Server01,** and click **Apply**

__ c. Click **New.**

__ d. Under Name, enter **protocol_http_timeout_output.**

__ e. Under Value, enter **0.**

__ f. Click **OK**.

____ 10. Set the **JAVA_DUMP_TDUMP_PATTERN** value to **WSUSER.JVM.TDUMP.&JOBNAME..T&HHMMSS.**

__ a. In the Environment->WebSphere Variables panel

__ b. Click on **C6Server01,** and click **Apply**

__ c. Click **New.**

__ d. Under Name, enter **JAVA_DUMP_TDUMP_PATTERN.**

__ e. Under Value, enter **WSUSER.JVM.TDUMP.&JOBNAME..T&HHMMSS.**. (include a period at the end!)

__ f. Click **OK**.

____ 11. Save your changes and restart the application server.

__ a. In the Console's left-side navigation menu, expand **System Administration** and click **Save Changes to Master Repository**.

__ b. Click the button labeled **Save**.

__ c. Click **Logout** at the top of the browser window to log yourself out of the Administrative Console.

__ d. Close the browser window.

__ e. From the command prompt enter the command.

```
./stopServer.sh C6Server01
```

__ f. When the stop has completed, start the server using the command.

```
./startServer.sh C6Server01
```

Part 3: Detecting Hung Threads

Now that you have customized the hang detection policy, you will call a Servlet that is intended to hang, so that you can see the ThreadMonitor in action.

____ 1. Make a call to the poorly-coded Servlet, HangServlet.

__ a. Open a web browser window.

__ b. Type in the address
`http://mvs22x.rtp.raleigh.ibm.com:9080/HangLab/HangServlet` and press enter.

__ c. You shouldn't see any output in this browser window, it will just be "waiting".

____ 2. Call HangServlet a second time.

__ a. Open a second web browser window.

__ b. Again, type the address
`http://mvs22x.rtp.raleigh.ibm.com:9080/HangLab/HangServlet` and press enter.

__ c. Like the first window, no output will be produced in this window.

____ 3. Calling HangServlet twice will cause a hang. Since the ThreadMonitor threshold was set to 120 seconds, wait two minutes, and then look for the ThreadMonitor's output at the end of the SYSPRINT for the servant.

__ a. Wait at least two minutes. (more like 11 minutes because the threshold doesn't seem to be working right now)

__ **b.** From a PCOMM session pointing to **RALVMY.RTP.RALEIGH.IBM.COM**, type '**D MVS21x**'

__ **c.** Type **TSO WSUSER** and when prompted for a password, type **mo1nday**

__ **d.** Type **=d.st** to find the SYSPRINT for the job

__ e. Look in the **Sysprint** for **C6SR01S.**

__ f. You should see a line similar to the following near the end of the file. Do a 'Find' to search for **ThreadMonitor** if you are having trouble finding it.

```
Trace: 2005/03/12 17:40:04.930 01 t=8CF658 c=UNK key=P8 (13007002)
  ThreadId: 00000038
  FunctionName: com.ibm.ws.runtime.component.ThreadMonitorImpl
  SourceId: com.ibm.ws.runtime.component.ThreadMonitorImpl
  Category: WARNING
  ExtendedMessage: BBOO0221W: WSVR0605W: Thread
"WebSphere:ORB.thread.pool t=008cca60" (00000028) has been active for
608089 milliseconds and may be hung.  There is/are 1 thread(s) in
total in the server that may be hung.
```

____ 4. There are some important things to note about this message.

__ a. All log messages generated by the ThreadMonitor are printed with the component name "ThreadMonitor", so they are easy to locate.

__ b. The thread name (WebSphere:ORB.thread.pool) and thread ID (008cca60) are printed in the message. These will help you locate the hung thread in a Java dump, which is usually the next step in debugging, after you have discovered that there are hanging threads.

Part 4: Building a Simple PD Tool Using the ThreadMonitor

The ThreadMonitor is a welcome addition to WebSphere Application Server, but its functionality is limited to notifying you when a hang occurs. In most cases, the first action you will take after becoming aware of the hang is forcing a Java dump so that you can further investigate. Because the ThreadMonitor broadcasts a JMX notification when it detects a hang, you can easily build your own tools using JMX to help automate the problem determination process.

In this section, you will see a simple bean that listens for a hung thread, and forces two Java dumps separated by a few minutes in time, so that you can look for the root cause of the hang. A tool like this can be very useful while troubleshooting an application that is known to hang, but at unpredictable times. It frees you from having to constantly watch the server so that you can force a Java dump when it hangs.

The bean is implemented as a Startup Bean. Startup Beans, like the other Programming Model Extensions, are now available in all editions, including Express. They were previously only available as a part of Enterprise Edition. The Startup Bean simply registers a JMX listener, and waits for a notification from the ThreadMonitor.

If you want to learn more about how this utility was implemented, the code is available in the expanded EAR directory, located at c:\LabFiles60\HungThreadLab\HangLabExpanded.

____ 1. The Startup Bean that contains the monitoring code is already deployed on your application server, as a part of the **HangLabApp** application. However, you need to enable the Startup Beans service on the server before you can take advantage of it.

__ a. Open a web browser window.

__ b. Type the location `http://mvs22x.rtp.raleigh.ibm.com:9080/ibm/console` and press enter.

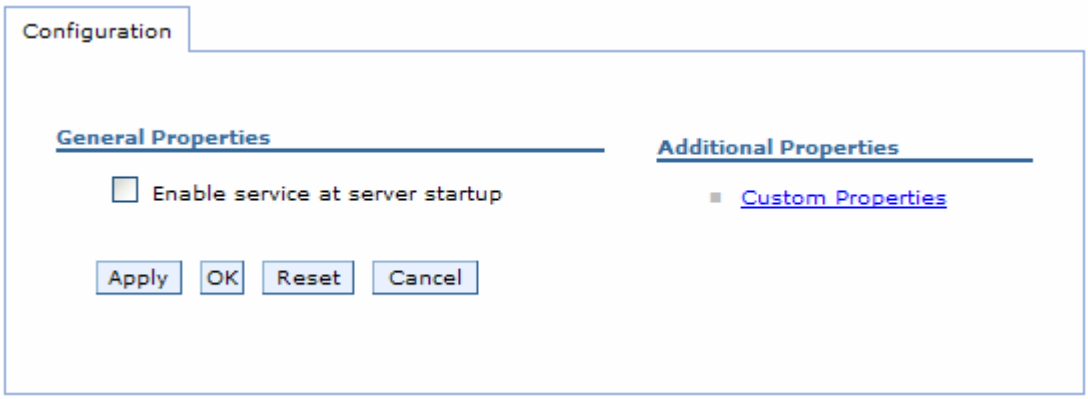
__ c. Enter the User ID **hanglab**, and click the **Log In** button.

__ d. In the left-side navigation menu, expand **Servers**, and click **Application Servers**.

__ e. In the resulting list, click **C6Server01**.

__ f. Under the **Container Settings** heading on the right, expand **Container Services**, and click **Startup beans service**.

__ g. Check the box next to **Enable service at server startup**, as shown below, and click **OK**.



Note: A startup bean is a session bean that implements the AppStartup interface. Implementing this interface requires two additional methods start() and stop(), which get executed when the application starts and stops, respectively. To see this firsthand, you can browse the code from this example, located in C:\LabFiles60\HungThreadLab\HangLabAppExpanded\HangMonitorEJB.jar\ejbs\MonitorBean.java

____ 2. Save your changes and restart the application server.

__ a. In the Console's left-side navigation menu, expand **System Administration** and click **Save Changes to Master Repository**.

__ b. Click the button labeled **Save**.

__ c. Click **Logout** at the top of the browser window to log yourself out of the Administrative Console.

__ d. Close the browser window.

__ e. From the command prompt enter the command.

```
./stopServer C6Server01
```

__ f. When the stop has completed, start the server using the command.

```
./startServer C6Server01
```

__ g. Startup beans are now enabled for this application server. Whenever an application starts up, any deployed startup beans will execute their start() methods.

____ 3. Like you did earlier, call HangServlet twice to generate a hang.

__ a. Open a web browser window.

__ b. Type in the address `http://mvs22x.rtp.raleigh.ibm.com:9080/HangLab/HangServlet` and press enter.

__ c. You shouldn't see any output in this browser window, it will just be "waiting".

__ d. Open a second web browser window.

__ e. Again, type the address `http://mvs22x.rtp.raleigh.ibm.com:9080/HangLab/HangServlet` and press enter.

__ f. Like the first window, no output will be produced in this window.

____ 4. After a few minutes, you will again see notification of hanging threads. This time, however, you will also find that a couple of TDUMPs have been taken (as well as system dumps).

__ **a.** Wait two minutes (more like 11 because the threshold doesn't seem to be working right now).

__ b. From **=d.st**, again look in the **Sysprint** for **C6SR01S.** and notice again that the ThreadMonitor has found hung threads. This time you should also see a message generated by the startup bean that reads "Generating the first Java dump".

__ c. Look for 'WSUSER.JVM.TDUMP. C6SR01S.Txxxxxx' under option 3.4, where xxxxxx is a timestamp.

__ d. Get rid of the system dumps by responding to the WTOR from option **d.log**

Note: To generate the Java dump, the startup bean creates a JMX listener, which waits for a hung thread notification from the ThreadMonitor. JMX is outside the scope of this lab, but to learn more about how to

creating a simple JMX listener, check out the documented code in C:\LabFiles60\HungThreadLab\HangLabAppExpanded\HangMonitorEJB.jar\ejbs\MonitorBean.java

Once the notification is received, a call is made to the **com.ibm.jvm.Dump.SystemDump()** method, which generates a thread dump. You could also make a call to the **com.ibm.jvm.Dump.JavaDump()** method but that doesn't appear to be very useful on z/OS. A simple Servlet that calls this method when invoked can be a handy addition to your toolbox.

Part 5: Inspecting the Java Dump File (z/OS)

A Java tdump file can be useful for diagnosing the cause of a hang. In this section you will use the jformat tool which is shipped with Java in order to the identify the cause of the hang that was produced in the previous section.

In this particular example, the hang was caused by a deadlock. With this information already in hand to simplify the process, see if you can identify the deadlock using the Java dump file.

____ 5. A deadlock is caused when two or more threads are holding locks on objects for which the other threads are waiting. Since each thread will not release its lock until it can acquire the lock from another thread, the threads will "hang" forever. To identify this condition, there is a 'deadlock' command that you can use from the jformat tool

__ a. In the previous section you located the two Java dump files that were created. We will copy the first one to USS to be read with the jformat tool. To so this:

1) Open a command prompt on your thinkpad and telnet into your system:

```
telnet mvs22x.rtp.raleigh.ibm.com 1023
```

2) Navigate to the /tmp directory by typing :

```
cd /tmp
```

3) You'll notice a lot of dumps have been created. Delete them to clear space:

```
rm HEAPDUMP*

rm JAVADUMP*
```

4) Copy the TDUMP to USS

cp "//'WSUSER.JVM.TDUMP.C6SR01S.Txxxxxx'" JVMDUMP.TDUMP (where xxxxxx is the proper timestamp)

Note: In many cases it is useful to compare two Java dump files to see how the thread states have changed, but in this case, since you already know the problem is a deadlock (and the state of a deadlock will not change over time), you will only be using one dump.

__ b. Once copied (this takes awhile), we'll start jformat

```
cd /etc/c6cellB/AppServer/java/bin

jformat
```

__ c. You should be at a 'Ready……..' prompt. The first thing we need to do is tell jformat what dump we want to work with:

```
set dump=/tmp/JVMDUMP.TDUMP
```

__ d. Before we ask jformat about deadlocks, we need to 'initialize' the dump. This will take a little while but you should see a progress indicator as it works.

```
dis os
```

__ e. Finally, we will ask jformat about deadlocks in the dump:

```
deadlock
```

__ f. Notice that the analysis shows that a thread named **WebSphere:ORB.thread.pool t=008cf4a0** is waiting on a monitor held by **WebSphere:ORB.thread.pool t=008cf308**. **WebSphere:ORB.thread.pool t=008cf308**, however, is waiting for a monitor that is being held by **WebSphere:ORB.thread.pool t=008cf4a0**.

```
Deadlock(s) detected !!!

============================

  Thread 0x0x8cf4a0 "WebSphere:ORB.thread.pool t=008cf4a0"

    is waiting to be notified for:

      (0x1c143510) "java/lang/Object"

       which is owned by:

    Thread 0x0x8cf308 "WebSphere:ORB.thread.pool t=008cf308"

      which is waiting to be notified for:

        (0x1c143500) "java/lang/Object"

         which is owned by:

      Thread 0x0x8cf4a0 "WebSphere:ORB.thread.pool t=008cf4a0"

============================
```

__ g. Since neither of these threads will do any work until they can access the aforementioned monitors, they will remain in this state indefinitely.

____ 6. Resolving a deadlock problem such as this requires modifying the program's source code, to ensure that threads cannot get themselves into this situation. This step is not covered in this lab. However, the source code to the example Servlet is available in C:\LabFiles60\HungThreadLab\HangLabAppExpanded\HangLab.war\WEB-INF\source\HangServlet.java if you wish to inspect the code on your own time.

Part 6: Inspecting the Java Dump File (Distributed)

A Java dump file is very useful for diagnosing the cause of a hang. In this section you will use the Monitor Information section of the Java dump file to identify the cause of the hang that was produced in the previous section.

In this particular example, the hang was caused by a deadlock. With this information already in hand to simplify the process, see if you can identify the deadlock using the Java dump file.

____ 1. A deadlock is caused when two or more threads are holding locks on objects for which the other threads are waiting. Since each thread will not release its lock until it can acquire the lock from another thread, the threads will "hang" forever. To identify this condition, you should use the **Monitor Pool Dump** section in the Java dump file.

__ a. In the previous section you located the two Java dump files that were created. Open the file with the earliest timestamp using WordPad.

Note: In many cases it is useful to compare two Java dump files to see how the thread states have changed, but in this case, since you already know the problem is a deadlock (and the state of a deadlock will not change over time), you will only be using one dump.

__ b. The Monitor Pool Dump section of the Java dump lists all of the monitors in the JVM. To find this section quickly, use the Find command (crtl+F) to search for **LKMONPOOLDUMP**.

__ c. In this section of the file, try to find the two Servlet threads that are deadlocked with each other. Each will be simultaneously holding a lock that the other is waiting for. Use the Thread Identifiers section (search for **LKFLATMONDUMP**) for a mapping of thread identifiers to thread names. This step is a bit more advanced than the rest of the lab. If you don't know how to do this, please ask an instructor for help, or just continue to the next step.

____ 2. If you found the deadlocked threads, congratulations! If not, don't worry. There is a feature of the IBM JVM that will identify deadlocks automatically in a Java dump file.

__ a. Search for **LKDEADLOCK** to find the deadlock detection section of the dump file.

__ b. The text in this section will look something like this:

```
1LKDEADLOCK    Deadlock detected !!!
NULL           ---------------------
NULL
2LKDEADLOCKTHR  Thread "Web Container : 1" (0x264B3490)
3LKDEADLOCKWTR    is waiting for:
4LKDEADLOCKMON      sys_mon_t:0x26D99328 infl_mon_t: 0x00000000:
4LKDEADLOCKOBJ      java.lang.Object@13FF3C08/13FF3C10:
3LKDEADLOCKOWN    which is owned by:
2LKDEADLOCKTHR  Thread "Web Container : 0" (0x25979DB8)
3LKDEADLOCKWTR    which is waiting for:
4LKDEADLOCKMON      sys_mon_t:0x26CDD7A8 infl_mon_t: 0x00000000:
4LKDEADLOCKOBJ      java.lang.Object@13FF3BF8/13FF3C00:
3LKDEADLOCKOWN    which is owned by:
2LKDEADLOCKTHR  Thread "Web Container : 1" (0x264B3490)
```

__ c. Notice that the analysis shows that a thread named **Web Container: 1** is waiting on a monitor held by **Web Container : 0**. Web Container : 0, however, is waiting for a monitor that is being

held by Web Container : 1. This should be the same conclusion you reached if you completed Step 1 of this section.

__ d. Since neither of these threads will do any work until they can access the aforementioned monitors, they will remain in this state indefinitely.

____ 3. Resolving a deadlock problem such as this requires modifying the program's source code, to ensure that threads cannot get themselves into this situation. This step is not covered in this lab. However, the source code to the example Servlet is available in C:\LabFiles60\HungThreadLab\HangLabAppExpanded\HangLab.war\WEB-INF\source\HangServlet.java if you wish to inspect the code on your own time.

What you did in this exercise

In Part 1, you used wsadmin to install the enterprise application that this lab exercise uses. It contained both the hanging Servlet code and the startup bean code that was used as a diagnostic tool.

In Part 2 you modified the default hang detection policy, creating a more aggressive policy for the purpose of saving time in this lab exercise. You decreased the ThreadMonitor threshold to two minutes, and the ThreadMonitor interval to twenty seconds. You wouldn't typically want to use a policy this aggressive in a real scenario.

In Part 3 you watched the ThreadMonitor catch a pair of hung threads, and interpreted the log output generated by this event.

In Part 4, you learned that the functionality of the ThreadMonitor can be extended by writing custom code that takes advantage of the ThreadMonitor's JMX notifications. You saw how a startup bean could be used as a JMX listener to automatically create Java dump files.

In Part 5 you inspected the Java dump file and found a deadlock between two Servlet threads. You also learned about the new JVM feature that automatically identifies deadlocks.