Java Thread Dumps

Part 2

Part II

Understanding Thread States

Before starting on dissecting thread dumps it is very important to get a good understanding of the common thread states. The documentation for java.lang.management.ThreadState clearly states the thread can be in any one of the following states

- NEW : A thread is in a new state when it has just been created and the start() method hasn't been invoked
- RUNNING : A thread is in a runnable state when the start method has been invoked and the JVM actively schedules it for execution
- BLOCKED / WAITING FOR MONITOR ENTRY: Thread enters this state when it isn't able to acquire the necessary monitors to enter a synchronized block. A thread waits for other threads to reliquish the monitor. A thread in this state isnt doing any work
- WAITING : A thread enters this state when it relinquishes its control over the monitor by calling Object.wait(). It waits till the time other threads invoke Object.notify() or Object.notifyAll(). In this state too the thread isnt doing any work
- TIMED WAITING : A thread enters this state when it relinquishes its control over the monitor by calling Object.wait(long). It waits till the time other threads invoke Object.notify() / Object.notifyAll() or when the wait time expires. In this state too the thread isnt doing any work
- SLEEPING / WAITING ON CONDITION: A thread sleeps when it calls Thread.sleep(long). As the name suggests the thread isnt doing in this state either
- Terminated
- Other

As you can see from the above list a thread is only doing active work when it is in the RUNABBLE state. In all other states the thread is just wasting CPU cycles. For maximum throughput from an application it is necessary to ensure that the maximum number of threads are in the RUNNABLE state and fewer threads in WAITING FOR MONITOR ENTRY, SLEEPING, WAITING or TIMED WAITING state.

Having said this it is necessary to understand threads can be in WAITING FOR MONITOR ENTRY, SLEEPING, WAITING or in the TIME WAITING states for valid reasons. Consider thre following thread of the resin application server which is simply waiting for incoming connections from a client. While dissecting thread dumps it will be essential that we evaluate each thread individually and then come to the conclusion whether it is wasting CPU cycles.

```
"tcpConnection-9011-7736" daemon prio=1 tid=0x351750b8 nid=0x7eab in Object.wait()
[0x2fdf4000..0x2fdf50b0]
   at java.lang.Object.wait(Native Method)
   - waiting on <0x43eef7c8> (a java.lang.Object)
   at com.caucho.server.TcpServer.accept(TcpServer.java:648)
   - locked <0x43eef7c8> (a java.lang.Object)
   at com.caucho.server.TcpConnection.accept(TcpConnection.java:211)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:132)
   at java.lang.Thread.run(Thread.java:595)
```

This is the key concept in understanding thread dumps. In further sections we will expand on this topic to understand how to dissect huge thread dumps, draw conclusions and zero down on the offending thread quickly.

Analysing Thread Dumps

Now that we have a good understanding of thread states, it will be easier to dissect thread dumps. Thread dumps for production class applications can be huge. A typical web application has anywhere between 25 - 50 worker threads, add to it threads from caches and system threads and we are looking at a large number. In this section we will identify guidelines which will help us to quickly isolate threads which are cause of concerns. Lets start

1. **Understand the application**: While thread dumps can be understood by anyone who knows the fundamental concepts, it adds immense value if you also understand the application, its technology components and way things have been implemented. You will see that this informaton will come handy in the following steps. If you are a consultant or someone who hasnt been associated closely with the development team you can look at the thread dump and ask releveant questions to the team.

2. **Eliminate Runnable threads**: Runnable threads are already being actively scheduled and there is no need to extensively focus on them in the first pass. We will return to Runnable threads when we look at BLOCKED or WAITING threads.

3. **Eliminate Idle threads**: It is natural that some threads in the application are in the idle state. They would either be waiting for an incoming connection or sleeping for a period of time before they wake. Here your application context will come handy. Eliminate such threads from your analysis which would be ideally found idling cycles. In the following snippet you will find three threads which are WAITING or SLEEPING for valid reasons. "tcpConnection-9011-7736" is waiting for incoming connections, "Store org.hibernate.cache.StandardQueryCache Expiry Thread" and "Store org.hibernate.cache.UpdateTimestampsCache Expiry Thread" are sleeping for some time before they wake up again. Since this is the expected nature of application they can be safely ignored.

```
"tcpConnection-9011-7736" daemon prio=1 tid=0x351750b8 nid=0x7eab in
Object.wait() [0x2fdf4000..0x2fdf50b0]
   at java.lang.Object.wait(Native Method)
   - waiting on <0x43eef7c8> (a java.lang.Object)
   at com.caucho.server.TcpServer.accept(TcpServer.java:648)
   - locked <0x43eef7c8> (a java.lang.Object)
   at com.caucho.server.TcpConnection.accept(TcpConnection.java:211)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:132)
   at java.lang.Thread.run(Thread.java:595)

"Store org.hibernate.cache.StandardQueryCache Expiry Thread" daemon prio=1
tid=0x3becd4e0 nid=0x1c6e waiting on condition [0x35389000..0x353891b0]
   at java.lang.Thread.sleep(Native Method)
   at net.sf.ehcache.store.DiskStore.expiryThreadMain(DiskStore.java:725)
   at net.sf.ehcache.store.DiskStore.access$700(DiskStore.java:98)
   at net.sf.ehcache.store.DiskStore$ExpiryThread.run(DiskStore.java:882)

"Store org.hibernate.cache.UpdateTimestampsCache Expiry Thread" daemon
prio=1 tid=0x397868a8 nid=0x1c6c waiting on condition
[0x3548a000..0x3548b0b0]
   at java.lang.Thread.sleep(Native Method)
   at net.sf.ehcache.store.DiskStore.expiryThreadMain(DiskStore.java:725)
   at net.sf.ehcache.store.DiskStore.access$700(DiskStore.java:98)
   at net.sf.ehcache.store.DiskStore$ExpiryThread.run(DiskStore.java:882)
```

4. **Analyse BLOCKED and WAITING threads**: This is perhaps the most important part of analysis. It is important to ask the question why are these threads in either the BLOCKED or WAITING state when they should have been in the RUNNABLE state. These threads are waiting for monitors which have been acquired by other threads. The other threads could be either be in the RUNNABLE state or worse be in the BLOCKED state waiting for other monitors. Using thread dumps it is easy to scan which thread owns the required monitor and understand the situation better. In this thread dump you will find thread "tcpConnection-9011-7709" in the BLOCKED state waiting to lock a monitor. If you scan the thread dump you will see that "tcpConnection-9011-7640" has obtained the monitor. Further thread "tcpConnection-9011-7640" is in a runnable state.

```
"tcpConnection-9011-7709" daemon prio=1 tid=0x35029e40 nid=0x7e61 waiting
for monitor entry [0x2fe74000..0x2fe75e30]
   at java.beans.Introspector.getPublicDeclaredMethods(Introspector.java:1249)
   - waiting to lock <0x3d9bb6b0> (a java.lang.Class)
   at java.beans.Introspector.getTargetEventInfo(Introspector.java:934)
   at java.beans.Introspector.getBeanInfo(Introspector.java:388)
   at java.beans.Introspector.getBeanInfo(Introspector.java:222)
   ..............
   ..............
   ..............
   ..............
   ..............
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)


"tcpConnection-9011-7640" daemon prio=1 tid=0x35177a40 nid=0x7dc7 runnable
[0x305fa000..0x305fcf30]
   at java.lang.Throwable.fillInStackTrace(Native Method)
   at java.lang.Throwable.<init>(Throwable.java:218)
   at java.lang.Exception.<init>(Exception.java:59)
   ..............
   ..............
   at java.beans.Introspector.instantiate(Introspector.java:1453)
   at java.beans.Introspector.findExplicitBeanInfo(Introspector.java:410)
   - locked <0x3d9bb6b0> (a java.lang.Class)
   ..............
   ..............
   ..............
   at com.caucho.server.http.CacheInvocation.service(CacheInvocation.java:135)
   at com.caucho.server.http.RunnerRequest.handleRequest(RunnerRequest.java:346)
   at
com.caucho.server.http.RunnerRequest.handleConnection(RunnerRequest.java:274)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)
```

5. **Analyse thread dumps over a duration**: It is important to understand that the thread dumps are just a snapshot of the current execution context. Threads are rapidly and actively scheduled by the java virtual machine. Just one snapshot wouldnt be enough to reach the conclusion that threads are blocking each other. It is important to observe these threads over a small duration. I typically recommend people to take one snapshot every 5 seconds for a couple of minutes. You may like to increase the duration between snapshots based on your application context. In the following snippet you will see the same thread as they appear in 2 consequtive snapshots. In the first snapshot the thread "tcpConnection-9011-7696" is waiting for an incoming request. In the next snapshot the thread is seen in a RUNNABLE state. Similarly in the previous example it will be not be appropriate to conclude that "tcpConnection-9011-7709" in the BLOCKED because of "tcpConnection-9011-7640". If you observe the same pattern over consequtive or multiple thread dumps it is a fair conclusion.

```
"tcpConnection-9011-7696" daemon prio=1 tid=0x3466c258 nid=0x7e4f in
Object.wait() [0x2727b000..0x2727bfb0]
   at java.lang.Object.wait(Native Method)
   - waiting on <0x43eef7c8> (a java.lang.Object)
   at com.caucho.server.TcpServer.accept(TcpServer.java:648)
   - locked <0x43eef7c8> (a java.lang.Object)
   at com.caucho.server.TcpConnection.accept(TcpConnection.java:211)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:132)
   at java.lang.Thread.run(Thread.java:595)

In the next snapshot taken 5 seconds after the first snapshot

"tcpConnection-9011-7696" daemon prio=1 tid=0x3466c258 nid=0x7e4f runnable
[0x2727b000..0x2727bfb0]
   at java.net.SocketInputStream.socketRead0(Native Method)
   at java.net.SocketInputStream.read(SocketInputStream.java:129)
   at com.caucho.vfs.SocketStream.read(SocketStream.java:159)
   at com.caucho.vfs.ReadStream.readBuffer(ReadStream.java:790)
   at com.caucho.vfs.ReadStream.read(ReadStream.java:343)
   at com.caucho.vfs.ReadStream.readAll(ReadStream.java:373)
   at com.caucho.server.http.RunnerRequest.scanHeaders(RunnerRequest.java:493)
   at com.caucho.server.http.RunnerRequest.handleRequest(RunnerRequest.java:317)
   at
com.caucho.server.http.RunnerRequest.handleConnection(RunnerRequest.java:274)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)
```

Analyse a few race conditions

Lets use our knowledge to analyse a few race conditions

1. “classic” deadlock
2. "out-of-threads"
3. “resource-contention”

Classic Deadlocks

Deadlock occurs when some threads are blocked to acquire resources held by other blocked threads. A deadlock may arise due to a dependence between two or more threads that request resources and two or more threads that hold those resources. Lets examine a simple thread dump snippet showing a deadlock

```
Full thread dump Java HotSpot(TM) Client VM (1.5.0_04-b05 mixed mode, sharing):

" bar" prio=5 tid=0x00aa3010 nid=0xd0c waiting for monitor entry
[0x02d0f000..0x02d0fb68]
      at org.tw.testyard.thread.DeadLock.run(DeadLock.java:19)
      - waiting to lock <0x22aa9928> (a java.lang.Object)
      - locked <0x22aa9940> (a java.lang.Object)
      at java.lang.Thread.run(Unknown Source)

" foo" prio=5 tid=0x00aa2e00 nid=0x834 waiting for monitor entry
[0x02ccf000..0x02ccfbe8]
      at org.tw.testyard.thread.DeadLock.run(DeadLock.java:19)
      - waiting to lock <0x22aa9940> (a java.lang.Object)
      - locked <0x22aa9928> (a java.lang.Object)
      at java.lang.Thread.run(Unknown Source)

"main" prio=5 tid=0x000361b0 nid=0x1188 in Object.wait() [0x0007f000..0x0007fc3c]
      at java.lang.Object.wait(Native Method)
      - waiting on <0x22aa9a68> (a java.lang.Thread)
      at java.lang.Thread.join(Unknown Source)
      - locked <0x22aa9a68> (a java.lang.Thread)
      at java.lang.Thread.join(Unknown Source)
      at org.tw.testyard.thread.DeadLock.main(DeadLock.java:61)
```

In this snippet there are 2 threads "foo" and "bar" both waiting for monitor entry. These threads are waiting for monitors 0x22aa9940 and 0x22aa9928 respectively. However the same objects have been locked by the other threads. i.e. 0x22aa9940 has been locked by "bar" and 0x22aa9928 has been locked by "foo". Hence this is a classic deadlock case.

In addition to our thread dump analysis, the java virtual machine has made it really easy to spot deadlock issues. The actual thread dump is also accompanied by the following message

```
Found one Java-level deadlock:
=============================
"bar":
  waiting to lock monitor 0x00a680cc (object 0x22aa9928, a java.lang.Object),
  which is held by "foo"
"foo":
  waiting to lock monitor 0x00a680ac (object 0x22aa9940, a java.lang.Object),
  which is held by "bar"

Java stack information for the threads listed above:
===================================================
"bar":
      at org.tw.testyard.thread.DeadLock.run(DeadLock.java:19)
      - waiting to lock <0x22aa9928> (a java.lang.Object)
      - locked <0x22aa9940> (a java.lang.Object)
      at java.lang.Thread.run(Unknown Source)
"foo":
      at org.tw.testyard.thread.DeadLock.run(DeadLock.java:19)
      - waiting to lock <0x22aa9940> (a java.lang.Object)
      - locked <0x22aa9928> (a java.lang.Object)
      at java.lang.Thread.run(Unknown Source)

Found 1 deadlock.
```

As you can see that the java virtual machine has actually analysed the dump and called out the deadlock.

out-of-threads

Consider the following scenario

- Response time of your key pages is rapidly dropping. (pages are taking more time to render)
- The application throughput is more or less constant
- The application server and database servers are running with permissible limits of system parameters
- You have been asked to analyse the situation

The first step in analysing such situations would be to start with thread dumps since they give you an insight into whats happening at the jvm level without having a huge impact on the application as such. The application wouldnt be able to perform satisfactorly when you attach a sophisticated profiling tool. Consider the following thread dump

```
"tcpConnection-9011-8004" daemon prio=1 tid=0x2fed7830 nid=0x49ef runnable
[0x11546000..0x11548130]
   at java.net.SocketInputStream.socketRead0(Native Method)
   at java.net.SocketInputStream.read(SocketInputStream.java:129)
   at com.sun.mail.util.TraceInputStream.read(TraceInputStream.java:79)
   ..........
   ..........
   at com.caucho.server.http.RunnerRequest.handleRequest(RunnerRequest.java:346)
   at com.caucho.server.http.RunnerRequest.handleConnection(RunnerRequest.java:274)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)

"tcpConnection-9011-7994" daemon prio=1 tid=0x300cb8b8 nid=0x49e4 runnable
[0x1594e000..0x1594ffb0]
   at java.net.SocketInputStream.socketRead0(Native Method)
   at java.net.SocketInputStream.read(SocketInputStream.java:129)
   ..........
   ..........
   at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
   at java.io.BufferedInputStream.read(BufferedInputStream.java:235)
   at java.lang.Thread.run(Thread.java:595)

"tcpConnection-9011-7992" daemon prio=1 tid=0x2fefd450 nid=0x49e2 runnable
[0x17c94000..0x17c960b0]
   at java.lang.Class.isArray(Native Method)
   at
org.apache.commons.lang.builder.ToStringStyle.appendInternal(ToStringStyle.java:419)
   at org.apache.commons.lang.builder.ToStringStyle.append(ToStringStyle.java:319)
   ..........
   ..........
   at com.caucho.server.http.FilterChainFilter.doFilter(FilterChainFilter.java:88)
   at com.caucho.server.http.RunnerRequest.handleConnection(RunnerRequest.java:274)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)

"tcpConnection-9011-7967" daemon prio=1 tid=0x309a3620 nid=0x49c5 runnable
[0x1e07c000..0x1e07de30]
   at java.net.SocketInputStream.socketRead0(Native Method)
   at java.net.SocketInputStream.read(SocketInputStream.java:129)
   at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
   ..........
   ..........
   at com.caucho.server.http.RunnerRequest.handleConnection(RunnerRequest.java:274)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)
```

In the following snippet you see 4 application threads. Additionally assume that these 4 application threads represents the total number of threads available for processing incoming requests. Closer look reveals that the application threads are in the RUNNABLE state. It is pretty evident that there is no room for any additional processing. The application server is pretty much maxed out on the request processing capacity and the new threads are waiting longer before they can be picked up and processed by the application server. It is possible that you may be seeing a sudden surge of users hitting the application. Based on the analysis the only immediate recourse would be to increase the number of threads in the thread pool to cater to this increased demand. However scaling the number of threads in the pool without any capacity consideration is a dangerous thing. You need to consider the impact of such changes before making the changes. (You could very well (and rightfully so) argue that the application hasnt been tuned to handle this kind of a load. Thats a topic of a different discussion)

Resource Contention

Resource contention typically happens when the threads are fighting for the same resources. If a majority of threads in the application are fighting for the same resource this could speak about poor application design. In some cases it is a necesary evil. Consider the scenario where multiple incoming requests are trying to retrieve data from the cache and render a page. The cache may need to retrieve data from the disk and hence this could create a contention issue. The most common symptons of resource contention issues are

- The throughput / responsiveness of the application has fallen considerably and the application is too slow
- The CPU utilization is low
- All the application resources (db connection pools etc) seem to free and available
- In thread dumps most the thread would appear in the BLOCKED / WAITING FOR MONITOR ENTRY state

```
"tcpConnection-9011-8009" daemon prio=1 tid=0x31446a80 nid=0x49f5 waiting for monitor entry [0x2edf5000..0x2edf6e
   at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.put(AbstractConcurrentReadCache.java:164
   - waiting to lock <0x44008de0> (a com.opensymphony.oscache.base.algorithm.LRUCache)
   at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.put(AbstractConcurrentReadCache.java:863)
   at com.opensymphony.oscache.base.Cache.putInCache(Cache.java:624)
   at com.opensymphony.oscache.web.tag.CacheTag.doAfterBody(CacheTag.java:380)
   ...........
   ...........
   ...........
   at com.caucho.server.http.Invocation.service(Invocation.java:315)
   at com.caucho.server.http.CacheInvocation.service(CacheInvocation.java:135)
   at com.caucho.server.http.RunnerRequest.handleRequest(RunnerRequest.java:346)
   at com.caucho.server.http.RunnerRequest.handleConnection(RunnerRequest.java:274)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)

" tcpConnection-9011-8008" daemon prio=1 tid=0x309fe908 nid=0x49f4 waiting for monitor entry [0x16bf3000..0x16bf4f
   at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.put(AbstractConcurrentReadCache.java:158
   - waiting to lock <0x44008de0> (a com.opensymphony.oscache.base.algorithm.LRUCache)
   at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.get(AbstractConcurrentReadCache.java:729)
   at com.opensymphony.oscache.base.Cache.getCacheEntry(Cache.java:666)
   at com.opensymphony.oscache.base.Cache.getFromCache(Cache.java:246)
   ...........
   ...........
   ...........
   at com.opensymphony.oscache.web.tag.CacheTag.doStartTag(CacheTag.java:514)
   at com.caucho.server.http.CacheInvocation.service(CacheInvocation.java:135)
   at com.caucho.server.http.RunnerRequest.handleRequest(RunnerRequest.java:346)
   at com.caucho.server.http.RunnerRequest.handleConnection(RunnerRequest.java:274)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)

" tcpConnection-9011-8007" daemon prio=1 tid=0x2f9f4e60 nid=0x49f2 waiting for monitor entry [0x15240000..0x15242
   at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.put(AbstractConcurrentReadCache.java:164
   - waiting to lock <0x44008de0> (a com.opensymphony.oscache.base.algorithm.LRUCache)
   at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.put(AbstractConcurrentReadCache.java:863)
   at com.opensymphony.oscache.base.Cache.putInCache(Cache.java:624)
   at com.opensymphony.oscache.web.tag.CacheTag.doAfterBody(CacheTag.java:380)
   ...........
   ...........
   ...........
   at com.caucho.server.http.Invocation.service(Invocation.java:315)
   at com.caucho.server.http.CacheInvocation.service(CacheInvocation.java:135)
   at com.caucho.server.http.RunnerRequest.handleRequest(RunnerRequest.java:346)
   at com.caucho.server.http.RunnerRequest.handleConnection(RunnerRequest.java:274)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)

" tcpConnection-9011-7817" daemon prio=1 tid=0x24fee130 nid=0x491c runnable [0x1513e000..0x15140130]
   at java.lang.System.identityHashCode(Native Method)
   at java.io.ObjectOutputStream$HandleTable.hash(ObjectOutputStream.java:2165)
   at java.io.ObjectOutputStream$HandleTable.lookup(ObjectOutputStream.java:2098)
   ..........
   ..........
   at
com.opensymphony.oscache.plugins.diskpersistence.AbstractDiskPersistenceListener.storeGroup(AbstractDiskPersistenceList
   at
com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.persistStoreGroup(AbstractConcurrentReadCache
   at
com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.addGroupMappings(AbstractConcurrentReadCac
   at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.remove(AbstractConcurrentReadCache.java:
   - locked <0x44008de0> (a com.opensymphony.oscache.base.algorithm.LRUCache)
   - locked <0x44008de0> (a com.opensymphony.oscache.base.algorithm.LRUCache)
   at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.put(AbstractConcurrentReadCache.java:159
   - locked <0x44008de0> (a com.opensymphony.oscache.base.algorithm.LRUCache)
   at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.put(AbstractConcurrentReadCache.java:863)
   at com.opensymphony.oscache.base.Cache.putInCache(Cache.java:624)
   ..........
   ..........
   at com.caucho.server.http.Invocation.service(Invocation.java:315)
   at com.caucho.server.http.CacheInvocation.service(CacheInvocation.java:135)
   at com.caucho.server.http.RunnerRequest.handleRequest(RunnerRequest.java:346)
   at com.caucho.server.http.RunnerRequest.handleConnection(RunnerRequest.java:274)
   at com.caucho.server.TcpConnection.run(TcpConnection.java:139)
   at java.lang.Thread.run(Thread.java:595)
```

In the following snippet all the threads are waiting to acquire lock on the monitor <0x44008de0>. The monitor has been acquired by tcpConnection-9011-7817. Unless tcpConnection-9011-7817 releases the lock all the threads would continue to be in the BLOCKED state. It isnt surprising to see that tcpConnection-9011-7817 has acquired lock on the same object thrice. All java locks are re-enterant and hence threads owning the lock can enter the synchronized block without any issues. This is a snapshot from a real application. Additional background investigation revealed that the operating system had run out of file descriptors and hence thread tcpConnection-9011-7817 couldnt really persist the cached object to the disk and release the monitor. Consequtively all the threads in the application entered the BLOCKED state and the application came to a grinding halt. The problem was resolved by increasing the limit on the number of file descriptors (ulimit) and restarting the application.

Resource contention can also happen in other scenarios. In the following thread dump snippet you see that worker threads acquire connections from a ConnectionPool and then retrieve data from the connections. However since the connection pool has been inadqeuately sized (size set to 2 connections) it is not sufficient for all the three worker threads. Two threads "Thread-0" and "Thread-2" manage to get connections and start retrieving the data. However "Thread-1" waits to retrieve connection from the ConnectionPool. If you see such things happening often in your application it is time to revisit the connection pool configuration. In any case the connection pool size should be atleast as big as the number of worker threads in your application. You may see the similar issues when dealing with resources such as db connections, network connections or file handles.

```
Full thread dump Java HotSpot(TM) Client VM (1.5.0_04-b05 mixed mode, sharing):

"Thread-2" prio=5 tid=0x00a87338 nid=0x1254 waiting on condition
[0x02d4f000..0x02d4fae8]
      at java.net.SocketInputStream.socketRead(Native Method)
      at java.net.SocketInputStream.read(SocketInputStream.java:86)
      at java.io.BufferedInputStream.fill(BufferedInputStream.java:186)
      at java.io.BufferedInputStream.read(BufferedInputStream.java:204)
      at org.tw.testyard.thread.WorkerThread.run(WorkerThread.java:9)
      at java.lang.Thread.run(Unknown Source)

"Thread-1" prio=5 tid=0x00a861b8 nid=0xbdc in Object.wait()
[0x02d0f000..0x02d0fb68]
      at java.lang.Object.wait(Native Method)
      - waiting on <0x22aadfc0> (a org.tw.testyard.thread.ConnectionPool)
      at java.lang.Object.wait(Unknown Source)
      at org.tw.testyard.thread.ConnectionPool.getConnection(ConnectionPool.java:39)
      - locked <0x22aadfc0> (a org.tw.testyard.thread.ConnectionPool)
      at org.tw.testyard.thread.WorkerThread.run(WorkerThread.java:7)
      at java.lang.Thread.run(Unknown Source)

"Thread-0" prio=5 tid=0x00a86038 nid=0x988 waiting on condition
[0x02ccf000..0x02ccfbe8]

      at java.net.SocketInputStream.socketRead(Native Method)
      at java.net.SocketInputStream.read(SocketInputStream.java:86)
      at java.io.BufferedInputStream.fill(BufferedInputStream.java:186)
      at java.io.BufferedInputStream.read(BufferedInputStream.java:204)
      at org.tw.testyard.thread.WorkerThread.run(WorkerThread.java:9)
      at java.lang.Thread.run(Unknown Source)
```

Resources

- An Introduction to Java Stack Traces
- Thread Dump Analyser : The **TDA** (Thread Dump Analyzer) for Java is a small Swing GUI for analyzing Thread Dumps and Heap Information generated by the Sun Java VM
- Common WebLogic Server Deadlocks and How to Avoid Them
- View From The Trenches : Looking at Thread Dumps

Summary

Thread dumps are a snapshot of the JVM state. It reveals the state of all the threads and monitors. Thread dumps provide invaluable insight into the state of an application and help in quickly identifying issues. Threads dumps are invaluable tools in production environments since sophisticated tools like profilers and debuggers cant be easily attached. Thread dumps also dont impose a performance penatly on the application while profilers and debuggers are capable of causing an application to grind to a halt. Thread dumps can and should be used for first level analysis of common issues such as resource contention, out of threads or plain simple dead locks.

In this article you understood the importance of thread dumps, how to capture thread dumps, the various thread states and finally how to interpret thread dumps through various snippets captured from real world situations.

Revision History

Date	Comments
15-July-2007	First version published