Contents

[Java Thread Status 2](#_heading=h.sc1desi2cs68)

[Thread Dump Patterns by Type 3](#_heading=h.bd2rddj6lmf4)

[When Unable to Obtain a Lock (BLOCKED) 3](#_heading=h.dbbn8njzclv0)

[When in Deadlock Status 4](#_heading=h.1fvzj6n77juo)

[When Continuously Waiting to Receive Messages from a Remote Server 6](#_heading=h.pphmnu6kmxyi)

[How to take Java Thread dumps 6](#_heading=h.uw28xjgy2kzf)

[How to Solve Problems by Using Thread Dump 7](#_heading=h.v35p95dbr34t)

[Problem – High CPU Usage 7](#_heading=h.yn0bb6o51wej)

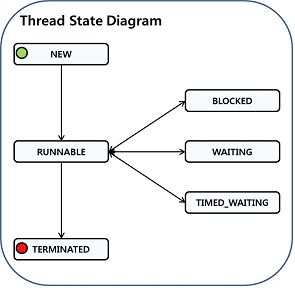
[PROBLEM 2: When the Processing Performance is Abnormally Slow 8](#_heading=h.gar1jlrha736)

[Thread dump Analysis - Tricks 10](#_heading=h.4lzj774titxk)

**JAVA THREAD DUMP ANALYSIS**

Java Thread Status

To analyse a thread dump, you need to know the status of threads. The statuses of threads are stated on java. lang.Thread.State.



**Figure 1: Thread Status.**

NEW: The thread is created but has not been processed yet.

RUNNABLE: The thread is occupying the CPU and processing a task. (It may be in WAITING status due to the OS's resource distribution.)

BLOCKED: The thread is waiting for a different thread to release its lock in order to get the monitor lock.

WAITING: The thread is waiting by using a wait, join or park method.

**TIMED\_WAITING**: The thread is waiting by using a sleep, wait, join or park method. (The difference from WAITING is that the maximum waiting time is specified by the method parameter, and*WAITING* can be relieved by time as well as external changes.)

**Thread Information from the Thread Dump File**

* Thread name: When using Java.lang.Thread class to generate a thread, the thread will be named Thread-(Number), whereas when using java.util.concurrent.ThreadFactory class, it will be named pool-(number)-thread-(number).
* Priority: Represents the priority of the threads.
* CPU time : CPU time consumed by thread.
* Thread ID: Represents the unique ID for the threads. (Some useful information, including the CPU usage or memory usage of the thread, can be obtained by using thread ID.)
* Thread status: Represents the status of the threads.
* Thread callstack: Represents the call stack information of the threads

# Thread Dump Patterns by Type

## When Unable to Obtain a Lock (BLOCKED)

This is when the overall performance of the application slows down because a thread is occupying the lock and prevents other threads from obtaining it. In the following example, BLOCKED\_TEST pool-1-thread-1 thread is running with <0x0000000780a000b0> lock, while BLOCKED\_TEST pool-1-thread-2 and BLOCKED\_TEST pool-1-thread-3 threads are waiting to obtain <0x0000000780a000b0> lock.

"BLOCKED\_TEST pool-1-thread-1" prio=6 tid=0x0000000006904800 nid=0x28f4 runnable [0x000000000785f000]

java.lang.Thread.State: RUNNABLE

at java.io.FileOutputStream.writeBytes(Native Method)

at java.io.FileOutputStream.write(FileOutputStream.java:282)

at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:65)

at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:123)

- locked <0x0000000780a31778> (a java.io.BufferedOutputStream)

at java.io.PrintStream.write(PrintStream.java:432)

- locked <0x0000000780a04118> (a java.io.PrintStream)

at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:202)

at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:272)

at sun.nio.cs.StreamEncoder.flushBuffer(StreamEncoder.java:85)

- locked <0x0000000780a040c0> (a java.io.OutputStreamWriter)

at java.io.OutputStreamWriter.flushBuffer(OutputStreamWriter.java:168)

at java.io.PrintStream.newLine(PrintStream.java:496)

- locked <0x0000000780a04118> (a java.io.PrintStream)

at java.io.PrintStream.println(PrintStream.java:687)

- locked <0x0000000780a04118> (a java.io.PrintStream)

at com.nbp.theplatform.threaddump.ThreadBlockedState.monitorLock(ThreadBlockedState.java:44)

- locked <0x0000000780a000b0> (a com.nbp.theplatform.threaddump.ThreadBlockedState)

at com.nbp.theplatform.threaddump.ThreadBlockedState$1.run(ThreadBlockedState.java:7)

at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)

at java.lang.Thread.run(Thread.java:662)

Locked ownable synchronizers:

- <0x0000000780a31758> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)

"BLOCKED\_TEST pool-1-thread-2" prio=6 tid=0x0000000007673800 nid=0x260c waiting for monitor entry [0x0000000008abf000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.nbp.theplatform.threaddump.ThreadBlockedState.monitorLock(ThreadBlockedState.java:43)

- waiting to lock <0x0000000780a000b0> (a com.nbp.theplatform.threaddump.ThreadBlockedState)

at com.nbp.theplatform.threaddump.ThreadBlockedState$2.run(ThreadBlockedState.java:26)

at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)

at java.lang.Thread.run(Thread.java:662)

Locked ownable synchronizers:

- <0x0000000780b0c6a0> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)

"BLOCKED\_TEST pool-1-thread-3" prio=6 tid=0x00000000074f5800 nid=0x1994 waiting for monitor entry [0x0000000008bbf000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.nbp.theplatform.threaddump.ThreadBlockedState.monitorLock(ThreadBlockedState.java:42)

- waiting to lock <0x0000000780a000b0> (a com.nbp.theplatform.threaddump.ThreadBlockedState)

at com.nbp.theplatform.threaddump.ThreadBlockedState$3.run(ThreadBlockedState.java:34)

at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)

at java.lang.Thread.run(Thread.java:662)

Locked ownable synchronizers:

- <0x0000000780b0e1b8> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)

## When in Deadlock Status

This is when *thread A* needs to obtain *thread B*'s lock to continue its task, while *thread B* needs to obtain *thread A*'s lock to continue its task. In the thread dump, you can see that DEADLOCK\_TEST-1 thread has 0x00000007d58f5e48 lock, and is trying to obtain 0x00000007d58f5e60 lock. You can also see that DEADLOCK\_TEST-2 thread has 0x00000007d58f5e60 lock, and is trying to obtain 0x00000007d58f5e78 lock. Also, DEADLOCK\_TEST-3 thread has 0x00000007d58f5e78 lock, and is trying to obtain 0x00000007d58f5e48 lock. As you can see, each thread is waiting to obtain another thread's lock, and this status will not change until one thread discards its lock.

## When Continuously Waiting to Receive Messages from a Remote Server

The thread appears to be normal since its state keeps showing as RUNNABLE. However, when you align the thread dumps chronologically, you can see that socketReadThread thread is waiting infinitely to read the socket.

"socketReadThread" prio=6 tid=0x0000000006a0d800 nid=0x1b40 runnable [0x00000000089ef000]

java.lang.Thread.State: RUNNABLE

at java.net.SocketInputStream.socketRead0(Native Method)

at java.net.SocketInputStream.read(SocketInputStream.java:129)

at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:264)

at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:306)

at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:158)

- locked <0x00000007d78a2230> (a java.io.InputStreamReader)

at sun.nio.cs.StreamDecoder.read0(StreamDecoder.java:107)

- locked <0x00000007d78a2230> (a java.io.InputStreamReader)

at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:93)

at java.io.InputStreamReader.read(InputStreamReader.java:151)

at com.nbp.theplatform.threaddump.ThreadSocketReadState$1.run(ThreadSocketReadState.java:27)

at java.lang.Thread.run(Thread.java:662)

# How to take Java Thread dumps

**From the VM machine:**

* Find the Java process ID using below command

JPS

* Run Jstack command to collect thread dump

Jstack <pid>

**How you can capture a Thread dump from a Java application running in a Container**.

Firstly, you need to fetch the java process of your application running in a Kubernetes pod:

kubectl exec -it <pod name> bash – jps

Secondly, collect the thread dump of Kubernetes pod:

kubectl exec -it <pod name> bash – jstack <pid>

# How to Solve Problems by Using Thread Dump

### Problem – High CPU Usage

Example **1** – When High CPU Usage on Java process running on Container/instance.

**Troubleshooting Guide:**

1. Check the process level CPU usage using TOP command on Linux machine and understand the high CPU usage is caused by us or sys time.
2. If it is due to US time, then the application is causing the High CPU usage, you can identify the process is consuming High CPU.
3. Collect Java thread dump on frequent interval (5 seconds). Take at least 5-10 thread dumps.

**How to Analyse the Thread dumps**

**Find out which thread is using the CPU time the most.**

"NioProcessor-2" prio=10 **cpu=16796.88ms** elapsed=17.35s tid=0x000001f9e589e000 tid=0x0a8d2800 nid=0x2737 runnable [0x49aa5000]

java.lang.Thread.State: RUNNABLE

at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)

at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:210)

at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:65)

at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:69)

- locked <0x74c52678> (a sun.nio.ch.Util$1)

- locked <0x74c52668> (a java.util.Collections$UnmodifiableSet)

- locked <0x74c501b0> (a sun.nio.ch.EPollSelectorImpl)

at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:80)

at external.org.apache.mina.transport.socket.nio.NioProcessor.select(NioProcessor.java:65)

at external.org.apache.mina.common.AbstractPollingIoProcessor$Worker.run(AbstractPollingIoProcessor.java:708)

at external.org.apache.mina.util.NamePreservingRunnable.run(NamePreservingRunnable.java:51)

at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886

Once root cause is identified from the thread stack trace , understand the stack trace from bottom to top and identify the application package which is calling the java native code.

**Example 2** –

This Thread dump tells us a different story. The Thread “App-Thread1” is in RUNNABLE state and it’s **overusing the CPU** (cpu=22484.99ms) by writing streams of data. The method *connect* from the *acme.com.Connector* class is actively writing and flushing on the filesystem.

"App-Thread1" #1 prio=5 os\_prio=0 cpu=22484.99ms elapsed=77.44s tid=0x00007fcbfc00f800 nid=0x16a4 runnable [0x00007fcc01437000]

java.lang.Thread.State: RUNNABLE

at java.io.FileOutputStream.writeBytes(java.base@11/Native Method)

at java.io.FileOutputStream.write(java.base@11/FileOutputStream.java:354)

at java.io.BufferedOutputStream.flushBuffer(java.base@11/BufferedOutputStream.java:81)

at java.io.BufferedOutputStream.flush(java.base@11/BufferedOutputStream.java:142)

- locked <0x0000000430062c00> (a java.io.BufferedOutputStream)

at java.io.PrintStream.write(java.base@11/PrintStream.java:561)

- locked <0x0000000430062bd8> (a java.io.PrintStream)

at sun.nio.cs.StreamEncoder.writeBytes(java.base@11/StreamEncoder.java:233)

at sun.nio.cs.StreamEncoder.implFlushBuffer(java.base@11/StreamEncoder.java:312)

at sun.nio.cs.StreamEncoder.flushBuffer(java.base@11/StreamEncoder.java:104)

- locked <0x000000043007a7d0> (a java.io.OutputStreamWriter)

at java.io.OutputStreamWriter.flushBuffer(java.base@11/OutputStreamWriter.java:184)

at java.io.PrintStream.write(java.base@11/PrintStream.java:606)

- eliminated <0x0000000430062bd8> (a java.io.PrintStream)

at java.io.PrintStream.print(java.base@11/PrintStream.java:745)

at java.io.PrintStream.println(java.base@11/PrintStream.java:882)

- locked <0x0000000430062bd8> (a java.io.PrintStream)

at acme.com.Connector.connect(Connector.java:104)

at acme.com.Connector.init(Connector.java:54)



### PROBLEM 2: When the Processing Performance is Abnormally Slow

The following example will cover another use case. Overall, your CPU/Memory is under control however the application is hanging or performing badly. Let’s look at this stack trace:

When the application is slow, collect the thread dumps and application logs. Check for unusual application errors in logs and collect the thread dumps

After acquiring thread dumps several times, find the list of threads with BLOCKED status.

" DB-Processor-13" daemon prio=5 tid=0x003edf98 nid=0xca waiting for monitor entry [0x000000000825f000]

java.lang.Thread.State: BLOCKED (on object monitor)

at beans.ConnectionPool.getConnection(ConnectionPool.java:102)

- waiting to lock <0xe0375410> (a beans.ConnectionPool)

at beans.cus.ServiceCnt.getTodayCount(ServiceCnt.java:111)

at beans.cus.ServiceCnt.insertCount(ServiceCnt.java:43)

"DB-Processor-14" daemon prio=5 tid=0x003edf98 nid=0xca waiting for monitor entry [0x000000000825f020]

java.lang.Thread.State: BLOCKED (on object monitor)

at beans.ConnectionPool.getConnection(ConnectionPool.java:102)

- waiting to lock <0xe0375410> (a beans.ConnectionPool)

at beans.cus.ServiceCnt.getTodayCount(ServiceCnt.java:111)

at beans.cus.ServiceCnt.insertCount(ServiceCnt.java:43)

" DB-Processor-3" daemon prio=5 tid=0x00928248 nid=0x8b waiting for monitor entry [0x000000000825d080]

java.lang.Thread.State: RUNNABLE

at oracle.jdbc.driver.OracleConnection.isClosed(OracleConnection.java:570)

- waiting to lock <0xe03ba2e0> (a oracle.jdbc.driver.OracleConnection)

at beans.ConnectionPool.getConnection(ConnectionPool.java:112)

- locked <0xe0386580> (a java.util.Vector)

- locked <0xe0375410> (a beans.ConnectionPool)

at beans.cus.Cue\_1700c.GetNationList(Cue\_1700c.java:66)

at org.apache.jsp.cue\_1700c\_jsp.\_jspService(cue\_1700c\_jsp.java:120)

If the threads are BLOCKED, extract the threads related to the lock that the threads are trying to obtain.

Through the thread dump, you can confirm that the thread status stays BLOCKED because <0xe0375410> lock could not be obtained. This problem can be solved by analysing stack trace from the thread currently holding the lock.

There are two reasons why the above pattern frequently appears in applications using DBMS. The first reason is inadequate configurations. Even though the threads are still working, they cannot show their best performance because the configurations for DBCP and the like are not adequate. If you extract thread dumps multiple times and compare them, you will often see that some of the threads that were BLOCKED previously are in a different state.

The second reason is the abnormal connection. When the connection with DBMS stays abnormal, the threads wait until the time is out. In this case, even after extracting the thread dumps several times and comparing them, you will see that the threads related to DBMS are still in a BLOCKED state. By adequately changing the values, such as the timeout value, you can shorten the time in which the problem occurs.



## Thread dump Analysis - Tricks

**Trick #1 Know what to ignore in a Thread Dump**

As there is a lot of information in thread dumps, it might quickly be overwhelming, especially in server systems that manage hundreds or sometimes even thousands of simultaneous threads.

In general, avoid the daemon threads which are generally not to be worried of. Look for application related thread stack traces.

**Trick #2 Look at your code first**

Always check the thread stack trace containing your application package, you should then start looking at the libraries, server and JDK code to understand what could be going on. you will be able to directly look at the code with your application package.

**Trick #3 The longer the thread stack, the more suspicious it is**

This trick might seem strange, but it is based on my personal experience, and has proven to be a reasonably good indicator of potential problems. Basically, the longer the stack trace, the more probable it is that something is misbehaving.

Collect the java thread dump with multiple thread dumps.

With multiple thread dumps close to one another, we can start checking if threads have indeed continued executing or if they are “stuck” on a specific piece of code, for example waiting for a network response or waiting on a monitor.

You do need to be careful with this type of analysis because false negatives are also possible, because if two stack traces are identical between two thread dumps it could mean either:

The thread is stuck at this particular part of the code

The thread has continued executing normally but has again been executing the same part of the code



