References:

<https://www.baeldung.com/java-analyze-thread-dumps>

<https://www.baeldung.com/java-thread-dump>

# Which tool have you used for thread and heap dump analysis?

jps and jstack commands of java

UI paid tool: JProfiler

# When we need java thread dump?

Applications sometimes hang up or run slowly, to identifying the root cause we need thread dump

# What information does thread dump contains?

Thread dump will have “*current state of a running Java process*”

# How will you capture java thread dump?

we'll execute JVM Process Status (jps) command to discover the PID process of our application:

*$ jps*

*80661 NetworkDriver*

*33751 Launcher*

*80665 Jps*

*80664 Launcher*

*57113 Application*

Secondly, we get the PID for our application, in this case, the one next to the NetworkDriver. Then, we'll capture the thread dump using jstack

jstack -l 80661 > sender-receiver-thread-dump.txt

Note: Good practice while coding is to always try to give names to your threads so that you can easily analyze your thread dumps.

# Structure of java thread dump?

Let's have a look at the generated thread dump. The first line displays the timestamp while the second line informs about the JVM:

*2021-01-04 12:59:29*

*Full thread dump OpenJDK 64-Bit Server VM (15.0.1+9-18 mixed mode, sharing):*

Next section shows the Safe Memory Reclamation (SMR) and non-JVM internal threads:

*Threads class SMR info:*

*\_java\_thread\_list=0x00007fd7a7a12cd0, length=13, elements={*

*0x00007fd7aa808200, 0x00007fd7a7012c00, 0x00007fd7aa809800, 0x00007fd7a6009200,*

*0x00007fd7ac008200, 0x00007fd7a6830c00, 0x00007fd7ab00a400, 0x00007fd7aa847800,*

*0x00007fd7a6896200, 0x00007fd7a60c6800, 0x00007fd7a8858c00, 0x00007fd7ad054c00,*

*0x00007fd7a7018800*

*}*

Then, the dump displays the list of threads. Each thread contains the following information:

**Name**: it can provide useful information if developers include a meaningful thread name

**Priority** (prior): the priority of the thread

**Java ID** (tid): the unique ID given by the JVM

**Native ID** (nid): the unique ID given by the OS, useful to extract correlation with CPU or memory processing

**State**: the actual state of the thread

**Stack trace**: the most important source of information to decipher what is happening with our application

We can see from top to bottom what the different threads are doing at the time of the snapshot. Let's focus only on the interesting bits of the stack waiting to consume the message:

*"Monitor Ctrl-Break" #12 daemon prio=5 os\_prio=31 cpu=17.42ms elapsed=11.42s tid=0x00007fd7a6896200 nid=0x6603 runnable [0x000070000dcc5000]*

*java.lang.Thread.State: RUNNABLE*

*at sun.nio.ch.SocketDispatcher.read0(java.base@15.0.1/Native Method)*

*at sun.nio.ch.SocketDispatcher.read(java.base@15.0.1/SocketDispatcher.java:47)*

*at sun.nio.ch.NioSocketImpl.tryRead(java.base@15.0.1/NioSocketImpl.java:261)*

*at sun.nio.ch.NioSocketImpl.implRead(java.base@15.0.1/NioSocketImpl.java:312)*

*at sun.nio.ch.NioSocketImpl.read(java.base@15.0.1/NioSocketImpl.java:350)*

*at sun.nio.ch.NioSocketImpl$1.read(java.base@15.0.1/NioSocketImpl.java:803)*

*at java.net.Socket$SocketInputStream.read(java.base@15.0.1/Socket.java:981)*

*at sun.nio.cs.StreamDecoder.readBytes(java.base@15.0.1/StreamDecoder.java:297)*

*at sun.nio.cs.StreamDecoder.implRead(java.base@15.0.1/StreamDecoder.java:339)*

*at sun.nio.cs.StreamDecoder.read(java.base@15.0.1/StreamDecoder.java:188)*

*- locked <0x000000070fc949b0> (a java.io.InputStreamReader)*

*at java.io.InputStreamReader.read(java.base@15.0.1/InputStreamReader.java:181)*

*at java.io.BufferedReader.fill(java.base@15.0.1/BufferedReader.java:161)*

*at java.io.BufferedReader.readLine(java.base@15.0.1/BufferedReader.java:326)*

*- locked <0x000000070fc949b0> (a java.io.InputStreamReader)*

*at java.io.BufferedReader.readLine(java.base@15.0.1/BufferedReader.java:392)*

*at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:61)*

*Locked ownable synchronizers:*

*- <0x000000070fc8a668> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)*

At a first glance, we see that the main stack trace is executing java.io.BufferedReader.readLine which is the expected behavior. If we look further down we'll see all the JVM methods executed by our application behind the scenes. Therefore, we are able to identify the root of the problem by looking at the source code or other internal JVM processing.

At the end of the dump, we'll notice there are several additional threads performing background operations such as Garbage Collection (GC) or object termination:

*"VM Thread" os\_prio=31 cpu=1.85ms elapsed=11.50s tid=0x00007fd7a7a0c170 nid=0x3603 runnable*

*"GC Thread#0" os\_prio=31 cpu=0.21ms elapsed=11.51s tid=0x00007fd7a5d12990 nid=0x4d03 runnable*

*"G1 Main Marker" os\_prio=31 cpu=0.06ms elapsed=11.51s tid=0x00007fd7a7a04a90 nid=0x3103 runnable*

*"G1 Conc#0" os\_prio=31 cpu=0.05ms elapsed=11.51s tid=0x00007fd7a5c10040 nid=0x3303 runnable*

*"G1 Refine#0" os\_prio=31 cpu=0.06ms elapsed=11.50s tid=0x00007fd7a5c2d080 nid=0x3403 runnable*

*"G1 Young RemSet Sampling" os\_prio=31 cpu=1.23ms elapsed=11.50s tid=0x00007fd7a9804220 nid=0x4603 runnable*

*"VM Periodic Task Thread" os\_prio=31 cpu=5.82ms elapsed=11.42s tid=0x00007fd7a5c35fd0 nid=0x9903 waiting on condition*

Finally, the dump displays the Java Native Interface (JNI) references. We should pay special attention to this when memory leak occurs because they aren't automatically garbage collected:

*JNI global refs: 15, weak refs: 0*

Thread dumps are similar in their structure, but we'll want to get rid of the non-important data generated for our use case. On the other hand, we'll need to keep and group the important information from the tons of logs produced by the stack trace. Let's see how to do it!

# Recommendations to analyze thread dumps ?

1. **Synchronization Issues**

To identify synchronization issues, we need to focus on **RUNNABLE** or **BLOCKED** threads and eventually **TIMED\_WAITING** threads. These states will help us to identify conflict between 2 or more threads.

In a deadlock situation in which several threads running hold a synchronized block on a shared object

In thread contention when a thread is blocked waiting for others to finish. For example, the dump generated in the previous section.

1. **Execution Issues**

**for abnormally high CPU usage we only need to look at RUNNABLE threads**.

To analyze execution issues along with taking thread dump we need to use other commands to extract extra information. For example, top command

**top -H -p PID**: which displays what threads are consuming the OS resources within that process. We also need to look at the internal JVM threads such as GC just in case.

**When the processing performance is abnormally low, we’ll look at BLOCKED threads.**

In those cases, a single dump will most surely not be enough to understand what is happening. We'll need several dumps at close intervals in order to compare the stacks of the same threads at different times.

To understand the threads' evolution over time, a recommended best practice is to take at least 3 dumps, one at every 10 seconds. Another useful tip is to split the dumps into small chunks to avoid crashes loading the files.

1. **Some other recommendations:**

In execution issues, **capture several snapshots with an interval of 10 seconds** will help to focus on the actual problems. It is also recommended to split the files if needed to avoid loading crashes.

Use **naming when creating new threads** to better identify your source code

Depending on the issue**, ignore internal JVM processing** (for instance GC).

**Focus on long running or blocked threads** when issuing abnormal CPU or memory usage.

**Correlate the thread's stack with CPU processing** by using top -H -p PID

And most importantly, use Analyzer tools

Analyzing the Java thread dumps manually could be a tedious activity. For simple applications, it is possible to identify the threads generating the problem. On the other hand, for complex situations, we'll need tools to ease this task. We'll showcase how to use the tools in the next sections, using the dump generated for the sample thread contention.

# Online tools?

There are several online tools available. When using this kind of software **we need to take into account security issues**. Remember that **we could be sharing the logs with third-party entities**.

1. Fast thread

<https://fastthread.io/>

FastThread is probably the best online tool to analyze thread dumps for production environments. It provides a very nice graphical user interface. It also includes multiple functionalities such as CPU usage by threads, stack length, and most used and complex methods:

FastThread incorporates a REST API feature to automate the analysis of the thread dumps. With a simple cURL command, it's possible to instantly send the results. The main drawback is security because **it stores the stack trace in the cloud**.

1. JStack Review

<https://jstack.review/#tda_1_dump>

JStack Review is an online tool that analyzes the dumps within the browser. It is client-side only, thus no data is stored outside your computer. From the security perspective, this is a major advantage to use it. It provides a graphical overview of all the threads, displaying the running methods but also grouping them by status. JStack Review separates threads producing stack from the rest which is very important to ignore, for instance, internal processes. Finally, it also includes the synchronizers and the ignored lines:

1. Spotify Online Java Thread Dump Analyzer

<https://spotify.github.io/threaddump-analyzer/>

Spotify Online Java Thread Dump Analyser is an online open-source tool written in JavaScript. It shows the results in plain text separating the threads with and without the stack. It also displays the top methods from the running threads:

# Standalone applications?

1. **Jprofiler:**   
   <https://www.ej-technologies.com/products/jprofiler/overview.html>

JProfiler is the most powerful tool in the market, and well-known among the Java developer community. It is possible to test the functionality with a 10-day trial license. JProfiler allows the creation of profiles and attaches running applications to them. It includes multiple functionalities to identify problems on the spot, such as CPU and memory usage and database analysis. It supports also integration with IDEs:

1. **IBM Thread Monitor and Dump Analyzer for Java**

IBM TMDA can be used to identify thread contention, deadlocks, and bottlenecks. It is freely distributed and maintained but it does not offer any guarantee or support from IBM:

1. **Irockel Thread Dump Analyser (TDA)**

Irockel TDA is a standalone open-source tool licensed with LGPL v2.1. The last version (v2.4) was released in August 2020 so it is well maintained. It displays the thread dump as a tree providing also some statistics to ease the navigation: