



TROUBLESHOOTING JAVA ON LINUX

Sumit Nigam
Consultant Engineer
EMC Corporation
Sumit.Nigam@EMC.com

EMC²

Table of Contents

Why is effective and efficient troubleshooting important?	4
Why is troubleshooting difficult?	5
Troubleshooting	6
Load and CPU Usage.....	6
Java Thread dumps.....	8
High CPU usage.....	9
Garbage Collection issues.....	11
Heap dump.....	17
Deadlock detection and lock contention.....	18
Virtual Memory	20
Connection problems.....	22
Context Switching.....	26
CPU affinity	28
I/O Subsystem.....	29
Thread Names.....	32
File descriptors	32
Tempfs and ramfs	33
Application logging	33
Virtualization.....	34
System level tracing	36
Crash analysis.....	39
HotSpot Compilation.....	41
Linux Kernel Profiling.....	42
Monitoring	44
OS specific	44
Application specific.....	44

Troubleshooting Checklist.....	49
Conclusion	51
References	52

Disclaimer: The views, processes, or methodologies published in this article are those of the author. They do not necessarily reflect EMC Corporation's views, processes, or methodologies.

Why is effective and efficient troubleshooting important?

Seasoned programmers will agree with Murphy's Law; when left to themselves, things have a tendency to go from bad to worse. This Law has a direct bearing upon software application troubleshooting. According to recent Cambridge University research, the global cost of debugging software has risen to \$312 billion annually¹.

A problem may manifest in many forms. It could be a suddenly unresponsive system to a total application crash. It could be a system running in degraded manner to a system which simply errors out on every flow. What are the effective means to troubleshoot Java application on Linux? While, the topic at hand is enormously vast, there exist a lot of useful Linux commands and tips which every developer must learn to effectively diagnose a problem quickly. This Knowledge Sharing article will discuss at length commands and tips which can help developers appreciate the science behind troubleshooting. It is not lack of knowledge that limits us but inability to apply the same knowledge effectively and correlate application behavior with operating system (OS) bookkeeping data. We will look at some interesting examples and use command outputs to indicate how root cause analysis could have been done immediately.

Today, we have software many times more complex than what we wrote a few years back. Today, we also have computers capable of performing billions of instructions per second. Troubleshooting is an area which has not got its due. One cannot find a chapter dedicated to troubleshooting in many of the popular Java books.

No troubleshooting discussion is adequate if it cannot identify ways to effectively monitor for such occurrences going into the future. An age-old adage fits well here; prevention is better than cure. Thus, the article concludes with some good remedial measures to address the problems and good application monitoring strategies to help troubleshoot better and faster.

Why is troubleshooting difficult?

This is a good question to ask at the beginning of every software development especially in the context of applications being developed. Asking such a question allows one to identify key areas where a given application may pose challenges when deployed in production.

What makes troubleshooting difficult is partly because of little thought is given to it during the construction of software. Another aspect which makes it difficult is the unknown. This implies all such cases which do not directly impact the application but can still cause considerable deterioration to the application at runtime. These would include aspects such as underlying OS library issues, network issues leading to stream corruption, third party application update, or even unresolved bugs in Java virtual machine (JVM) itself.

What also makes the subject difficult is its vastness. Applications may be running on a virtualized machine (VM). They may have been on an older version of Java or OS. There are a good number of Linux flavors available and each of them may differ in some aspect or another. This article will focus on Oracle Java version 1.6 and 1.7 and Red Hat Enterprise Linux (RHEL) version 2.6. Some of the commands we discuss may require modifications when executed on other Linux versions and readers are directed to consult appropriate manual (man) pages.

Troubleshooting is also difficult because one can never plan enough. These problems fall under the same category as those in Mathematics which illustrate that a problem worth attacking proves it's worth by attacking back². Troubleshooting is also a skill which is tougher to acquire than programming alone. It is similar to detective work³ and that requires both knowledge and an analytical mind. Coupled with the fact that most production setups will not allow developers to log in into the system, it makes troubleshooting seemingly a daunting task altogether. Even in such cases, developers should be able to guide privileged user(s) to collect relevant information.

By no means can we completely cover the entire troubleshooting landscape for Java applications deployed on Linux. However, we have aimed to cover the most important and common aspects.

Troubleshooting

Let us gain some insights into what Java platform offers in troubleshooting space. We will begin with some tools and they take a deep dive into understanding thread dumps, heap dumps and garbage collection statistics.

Load and CPU Usage

First of all, how would we know whether our system is overloaded? Linux provides a metric—called load average⁴—which measures the amount of load on the system. Load average is such a misunderstood concept that is worth getting to know better.

A command known as **uptime** can be executed to find the load average values for the last 1, 5, and 15 minutes. Load average should always be equated to the number of CPU cores. If the ratio of load average to the number of cores > 1 , then the system is under load and requires attention.

```
$ uptime
10:03:28 up 10 days, 1:41, 1 user, load average: 7.04, 11.01, 17.90
```

The ratio for 15 minutes is 17.90/24, which is within reasonable limits

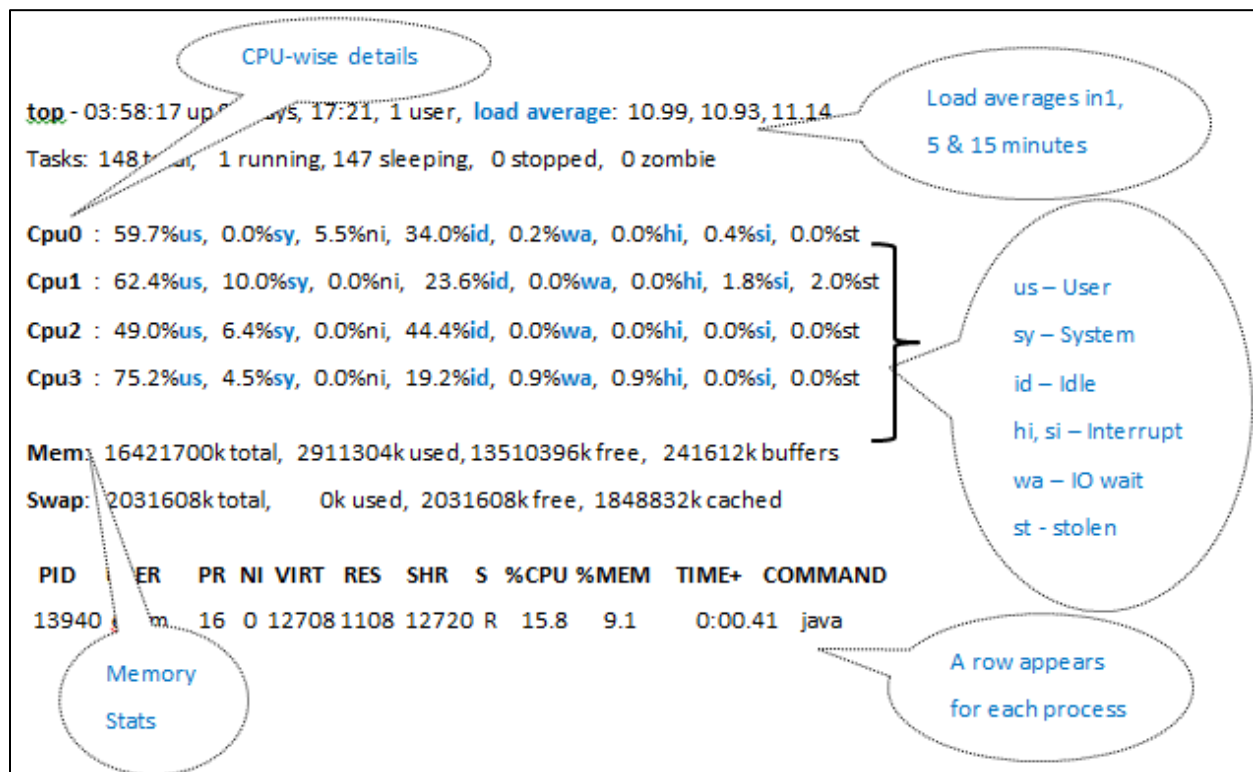
To know the number of cores we can use /proc file system:

```
$ cat /proc/cpuinfo | grep processor | wc -l
24
```

Do note that a few occasional flare-ups in load average are not really alarming. A good way to monitor load averages would be to keep an eye on the last 5 and 15 minute averages and taking their ratio with the number of CPU cores. If those values show a high value, there is reason to be alarmed. It is a good practice to first execute the **uptime** command before issuing any other diagnostic command. That would provide us with a good idea of how loaded the system is and we can avoid issuing commands which can further stress the system.

Linux also provides **top** command which provides a good indication of CPU utilization. It is a good idea to consider CPU utilization along with load average values when troubleshooting load related problems. A usual **top** command output provides load averages, CPUs, and their percentage utilization breakup in terms of usage with respect to user processes, system processes (executing kernel code), waiting for IO operations to complete, interrupt handling, and CPU time stolen from a virtual machine. Further details contain memory statistics in terms

of total RAM usage, allocation done to buffers, memory allocated to cache, and swap memory statistics.



The bottom part of **top** command emits details for each process. These include; its process ID (PID), user who created it, its priority (PR), its virtual memory (VIRT) image in kb which includes code, data, shared libraries as well as pages which are swapped out, its resident size in KB (RES) which is obtained through summation of sizes taken by code and data, its shared memory size (SHR) which is the amount of memory potentially shared with other processes, its current state (S), percentage of CPU utilization (%CPU), percentage memory usage (%MEM), total CPU time consumed since process started (TIME+), and command used to execute it. There are many more useful fields⁵ which can be turned ON in **top**.

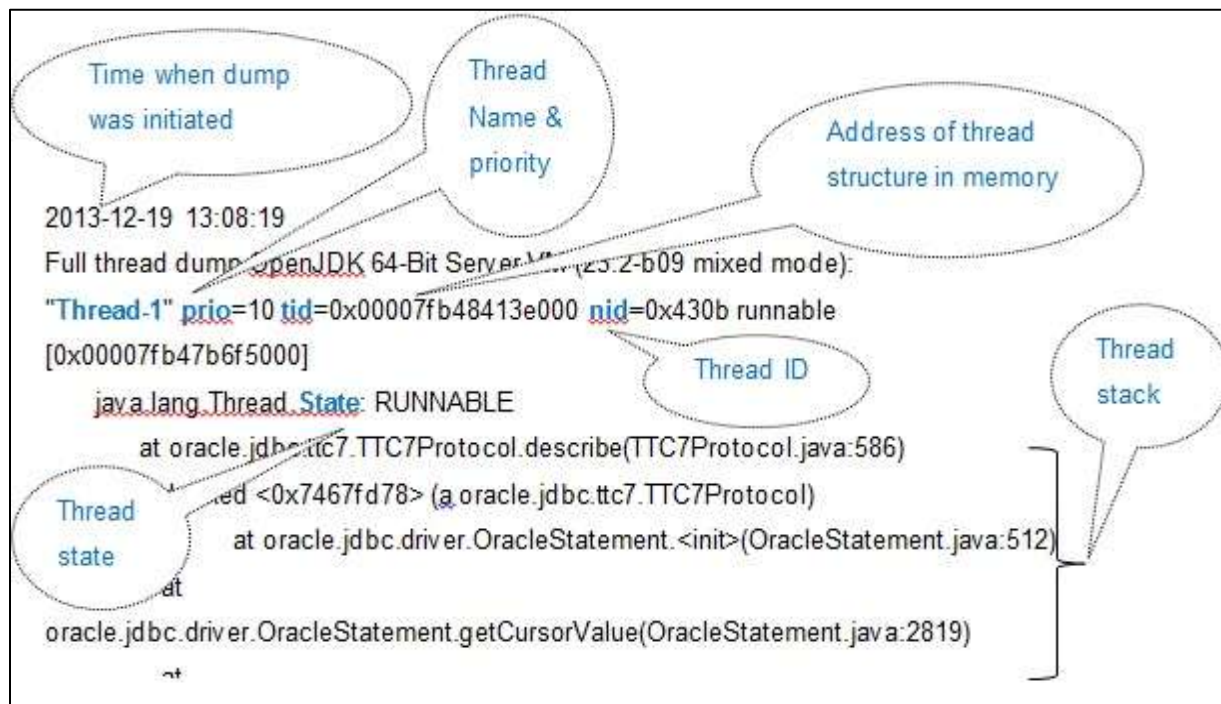
Output from top above shows that there are 4 CPUs (numbered Cpu0 to Cpu3). Around 60-70% overall CPU is used by user (**us**) processes, which is good because our applications would usually fall in this category. It is also derivable from this output that our application is running on a virtualized machine. We will discuss this later in this article when we talk about virtualization. For now, it will suffice to state that the last column (**st**) indicates a non-zero value which is relevant in virtual environments only.

Java Thread dumps

To know the process ID of our Java process, we can either use Linux **ps** (process status) command or JVM process status tool **jps**⁵ which comes bundled with it. Typing **jps** at the command prompt will list process IDs of those Java processes for which the tool has access permissions.

Issuing a SIGQUIT signal to a Java process whose PID (process identifier) is known will generate its thread dump. This is done by typing **kill -QUIT <PID>** at the command prompt.

Alternately, Java platform provides **jstack**⁶ command-line utility which attaches to the specified process and prints the stack traces of all threads that are attached to the JVM. These include application threads, VM internal threads, and optionally native stack frames. Using **jstack**, it is also possible to force (using **-F** flag) a thread dump of a hung process or when its output has been redirected such that the dumps are not available through **kill -QUIT** command.



Thread dumps contain a great amount of information about the current state of the application. These can be extremely useful in ascertaining a range of possible issues such as high CPU consumption, thread deadlocks, application slowdown, etc. Thread dump provides information about a thread state, priority, identifier (**nid** field), and name besides providing whole stack trace

of its execution path. There is other information as well, such as whether thread is a daemon thread or not.

Table 1 summarizes the states that a thread can be in:

Thread State	Description
NEW	The thread has been created but not yet started
RUNNABLE	The thread is executing in the JVM
BLOCKED	The thread is blocked waiting for a monitor lock
WAITING	The thread is waiting indefinitely for another thread to perform some action. Some good examples would be those threads which indefinitely execute wait or join indefinitely.
TIMED_WAITING	The thread is waiting for another thread to perform a specific action up to a specified waiting time. Examples would be threads which execute sleep, or wait and join with time outs, etc.
TERMINATING	The thread has completed execution

Table 1: Thread States

Tip: It is a good idea to generate multiple thread dumps separated by some delay. This is helpful in observing patterns and confirming application behavior across thread dumps. Tools such as Thread Dump Analyzer⁸ make it easier to compare thread dumps.

High CPU usage

High CPU usage manifests in many forms. A CPU-bound application may have consumed excessive CPU cycles. Applications which perform a lot of mathematical computations are in this category. It is also possible that an application does not perform heavy-duty computations but has semantics which lead to high-CPU situation. For example, consider a scenario where an application has a spin-loop through which it attempts to obtain some lock. Spin-locks can consume CPU cycles at a very high rate. In such scenarios, it is useful to consider placing breaks in the spin loop. It is also possible that an application suffers from a livelock⁹ condition.

To understand that our application is suffering from high CPU usage, we first need to take **top** output. We must focus on the **idle %** column to note whether CPU is constantly above 85-90% CPU utilization or not. A few intermittent spikes are usually not bad and can sometime be ignored. If CPU utilization is pegged above 85-90%, we need to isolate the root cause(s). Next step would be to initiate a few thread dumps. In thread dumps, we are only interested in those

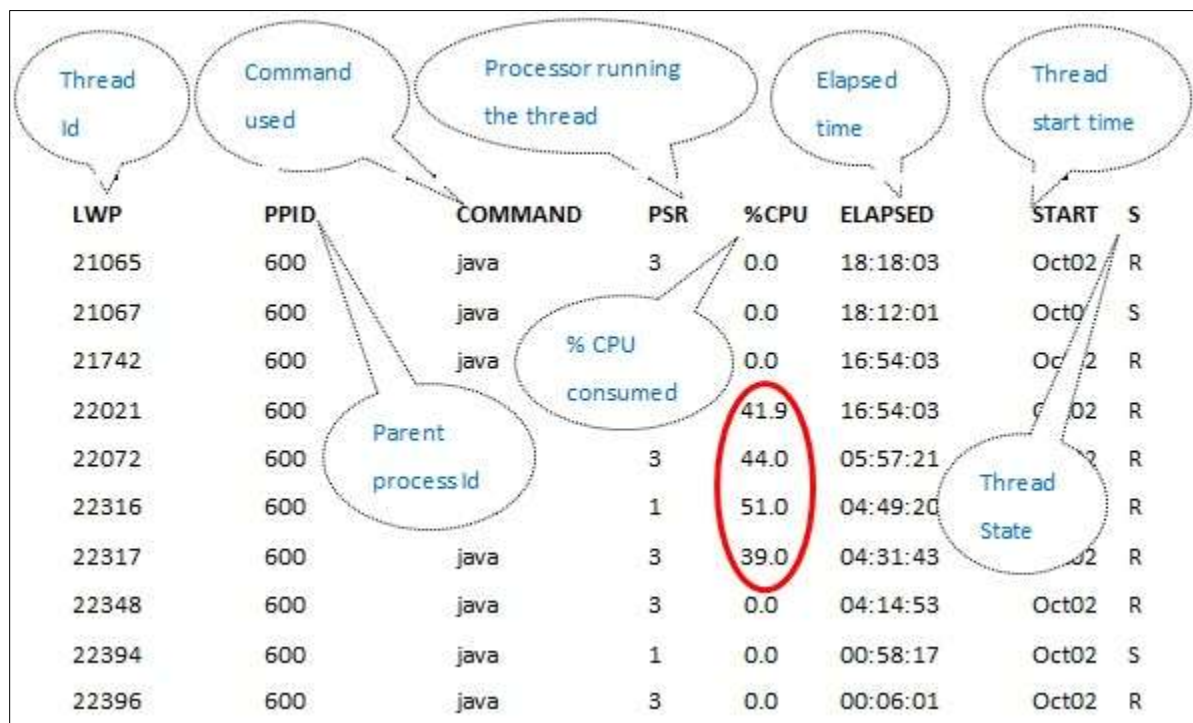
threads which are keeping the CPU busy. They can easily be found by looking for those threads which are in RUNNABLE state. Once we have isolated those threads in thread dump, we can analyze their stack dumps and focus on those code paths.

It may not be enough to just know all RUNNABLE threads. There may be many threads in RUNNABLE state. In most cases, it will be important to further isolate those threads which are indeed consuming a lot of CPU. The **top** command by default does not display a thread view. It is possible to get information at thread level by typing capital letter **H**.

Another command known as **ps** (process status) provides useful information too. This command offers a variety of useful information and can be used to find details of each thread to indicate their process ID (LWP), parent process ID (PPID), elapsed time since the thread was started, exact start time of thread, percentage CPU utilization, exactly which processor is the thread executing on, state information, etc.

Let us look at one **ps** output taken by executing the following command. Older Linux versions may need **ps -emo** and **pid** in place of **lwp** field.

```
ps -eLo 'lwp,ppid,cmd,psr,pcpu,etime,start_time,state' | grep java
```



The diagram shows the output of the command `ps -eLo 'lwp,ppid,cmd,psr,pcpu,etime,start_time,state' | grep java`. The output is a table with columns: LWP, PPID, COMMAND, PSR, %CPU, ELAPSED, START, and S. Annotations include: 'Thread Id' pointing to LWP, 'Command used' pointing to COMMAND, 'Processor running the thread' pointing to PSR, 'Elapsed time' pointing to ELAPSED, 'Thread start time' pointing to START, 'Parent process Id' pointing to PPID, '% CPU consumed' pointing to %CPU, and 'Thread State' pointing to S. The %CPU column has values 0.0, 0.0, 0.0, 41.9, 44.0, 51.0, 39.0, 0.0, 0.0, and 0.0. The 51.0 value is circled in red.

LWP	PPID	COMMAND	PSR	%CPU	ELAPSED	START	S
21065	600	java	3	0.0	18:18:03	Oct02	R
21067	600	java		0.0	18:12:01	Oct0	S
21742	600	java		0.0	16:54:03	Oct 2	R
22021	600			41.9	16:54:03	Oct 02	R
22072	600		3	44.0	05:57:21		R
22316	600		1	51.0	04:49:20		R
22317	600	java	3	39.0	04:31:43		R
22348	600	java	3	0.0	04:14:53	Oct02	R
22394	600	java	1	0.0	00:58:17	Oct02	S
22396	600	java	3	0.0	00:06:01	Oct02	R

First, notice the state of each thread as reported in the last column. Since we are interested only in runnable threads; we focus only on those whose state shows up as 'R'. Next, we notice that for some of these running threads, the %CPU column reports a high value (as highlighted in red). This column is indicative of CPU utilization. We, therefore, need to divert our attention to only these threads and not all. In effect, this output has helped reduce the scope of our troubleshooting considerably.

To, summarize, we took thread dumps and made a note of RUNNABLE threads. In parallel, we also took output from **ps** command. We filtered out a great number of RUNNABLE threads from thread dump for our analysis because we were able to find those threads which are consuming maximum CPU.

Next, we simply correlate this information back with thread dumps. Notice that the first column in the output above is indicative of light-weight process (LWP) ID and is a numeric field. We may remember that thread IDs are reported back in hexadecimal as **nid** field. So, let us take, for example, LWP for one of these high CPU consuming threads which is 22021. To convert this into hex should be simple: **printf "%x\n" 22021** yields 0x5605. It should be easy to now search for **nid=0x5605** in thread dumps and locate this offending thread in the application.

One may not fail to notice the information that is present in output above. For example, it is easy to even locate the exact CPU that the offending threads were running on. One may be able to also correlate this with output of **top** command which shows CPU-wise utilization. Another interesting bit above is that of start time which indicates when a given thread was started. It may be useful information when debugging issues because many threads can surprise by showing up in dumps when we do not expect them. The output above can clearly point out when a thread started. This kind of information is not present in thread dumps and correlation with Linux command becomes essential in such cases. Of course, it is important to have a good grip of application flows. For example, a long running thread may also be because of a thread pool that your application chose to use or implement. However, we cannot find how long a thread has been in present state. A good use in such cases would be application logs, which we cover later in a section dedicated to them.

Garbage Collection issues

One of the issues that applications usually face is excessive garbage collection (GC). Java objects are created on heap. Heap memory is divided into young generation and tenured

generation (and two survivor spaces). Newly allocated objects are created in young generation and objects get promoted to tenured (aka old generation) when garbage collector determines that they are candidates for long term retention and will not likely be referenced in near future.

When a garbage collector cleans the young generation, it leads to application threads getting paused. Such a pause is known as partial GC and amounts to a minor pause. When the garbage collector is unable to clean enough space in young generation, it will be forced to reclaim space from tenured generation. This amounts to full GC and is also known as major pause. Applications witness minor pauses and most of the times, they are harmless. A few major pauses may not be too bad either. However, excessive minor pauses can degrade the application substantially and we need to ascertain root cause for them as well.

Excessive garbage collection problems manifest in many forms. Some of them would be degraded application performance, an out of memory error being reported by the application which indicates that garbage collector was unable to reclaim enough space even after considerable effort. This “considerable effort” would show up in the form of very high CPU utilization. Applications may face severe shortage of CPU cycles because most of them are being used up by garbage collector threads. The out of memory error would be accompanied with a message that GC overhead limit has been exceeded.

To know that our application is facing garbage collection pause issues, we can again correlate thread dumps with **ps** command output. Thread dump also contain details about garbage collector threads as shown below.

```
"VM Thread" prio=10 tid=0x0000000000523000 nid=0x78c runnable
"GC task thread#0 (ParallelGC)" prio=6 tid=0x0000000000476800 nid=0x1650 runnable
"GC task thread#1 (ParallelGC)" prio=6 tid=0x0000000000479000 nid=0x2354 runnable
"GC task thread#2 (ParallelGC)" prio=6 tid=0x000000000047b000 nid=0x2250 runnable
"GC task thread#3 (ParallelGC)" prio=6 tid=0x000000000047d000 nid=0xbd8 runnable
```

Notice from the dump above that the “**VM Thread**” is in runnable state and so are GC task threads. VM Thread performs many operations internal to VM and one of them is garbage collection. From **top** (with thread view) or from **ps** command output we can find whether the CPU cycles being consumed are used up by GC threads or not.

Once we determine that the excessive GC activity is the root cause of CPU consumption, we can enable detailed GC output by passing in some flags to the application at its startup.

```
-J-verbosegc -XX:+PrintGCDetails -J -Xloggc:/path/to/gcdata/garbageCollData.txt -  
XX:+PrintGCTimeStamps -XX:+PrintGCApplicationConcurrentTime -  
XX:+PrintGCApplicationStoppedTime
```

Here is an explanation of these flags:

verbosegc	Enable detailed garbage collection information
-XX:+PrintGCDetails	Provides information specific to sizes of young and old generations before and after GC collections, size of total heap, time taken to perform GC in both the generations and size of objects being promoted to old generation
-Xloggc	Specify location of garbage collection data log file
XX:+PrintGCApplicationConcurrentTime	Amount of time application ran between GC pauses
-XX:+PrintGCApplicationStoppedTime	Length of collection pauses
XX:+PrintGCTimeStamps	Print GC activity time relative to start of application

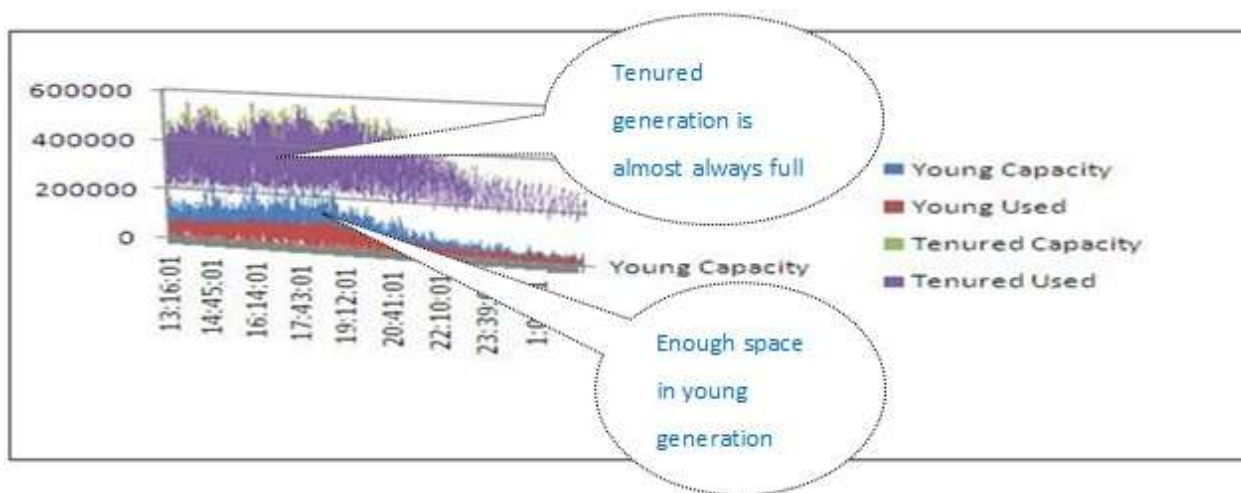
GC output obtained in log provided with **-Xloggc** option can be extremely useful in assessing frequency of GC and total percentage of time the application was paused. This is useful when we want to assure that no more than a certain percentage of GC pause time is tolerable. It is usually a good practice to not have more than 5% of application time being consumed by GC. The output would also let us know the frequency at which major collections may be occurring. The output also provides information about each generation size before and after the garbage collection. This can be extremely useful to see whether collection cycles are really removing enough garbage or not and if we need to tune our generation sizes.

It may be a good idea to specify pause time and throughput goals for the GC. To specify a pause time goal, the VM flag **-XX:MaxGCPauseMillis** can be used. GC will try to keep application pauses below this threshold. Throughput goal for GC is provided through **-XX:GCTimeRatio** flag which specifies ratio of time spent in GC activities to time dedicated to running the application. It is important to benchmark the application with these options before deciding upon a value.

Another way of causing garbage collection output during the runtime of an application would be to use JVM tool **jstat**⁶. The advantage of using **jstat** is that it can be used to collect GC data of a running application without having to pass on any flags to application at its startup. To collect detailed GC data from **jstat**, issue a command like **jstat -gc 1 50**, which is a way of specifying that we want to print 50 GC outputs, each sampled 1 second apart. Here is an explanation of columns we would see in such an output:

S0C, S1C	Capacities of survivor spaces S0 and S1 respectively (KBs)
S0U, S1U	Utilization of survivor spaces S0 and S1 respectively (KBs)
EC, EU	Young generation (aka Eden space) capacity and utilization respectively (KBs)
OC, OU	Old generation capacity and utilization respectively (KBs)
PC, PU	Permanent generation capacity and utilization respectively (KBs)
YGC, YGCT	Total young generation collections and total time spent in this collection respectively
FGC, FGCT	Total old generation collections and total time spent in this collection respectively
GCT	Total time taken by garbage collection activity

One of the interesting things that can be done with output is to plot it for analysis. Here is one plot created with outputs for YGC, YGU, OGC, and OGU columns.

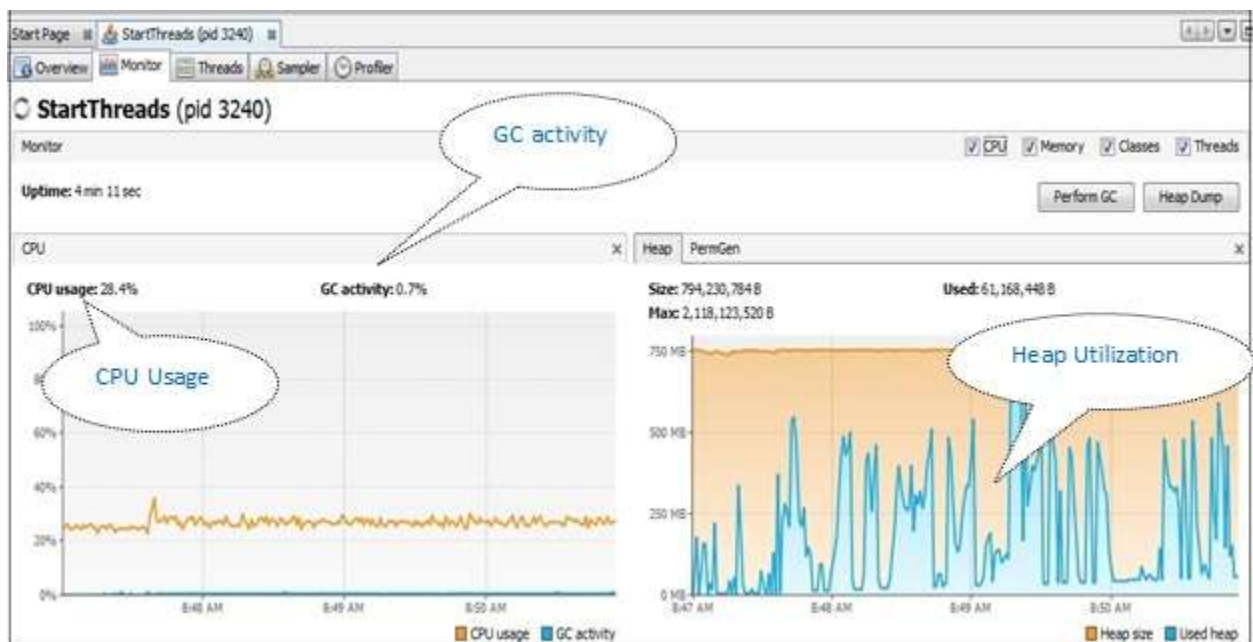


Notice that tenured generation is almost full while the young generation has a lot of free space. Importantly, the trend continues over a period of time. This could mean that we are promoting a

lot of objects from young generation to old generation. We may need to look closely at the code to see why we are having so many long lived objects (if we have too many objects which are surviving GC cycles when they should not). It is also possible that space provided to tenured generation is not enough and may require some tuning¹⁰. Another possibility could be whether some of our heavy duty objects have directly been created in old generation. When can objects be created directly in tenured generation? When GC is unable to reclaim enough space in young generation to allocate the object to it and even after clearing off weak references, the space made available falls short. So, that may be another possibility to consider based on plot above.

One quick hint which can be obtained from **jstat** is the reason for the last GC. This is obtained by passing **-gccause** option to **jstat** along with PID for the Java process. It provides information about last cause of GC as well as current cause of GC (if any).

We can also look at heap activity through **jVisualVM**¹¹ tool by attaching it to a running Java process. The advantage of **jVisualVM** is ease of use and graphical representation of information. Once it attaches with the process, its “Monitor” tab can be used to display heap activity as shown next.



There exists another heap generation which is known as permanent generation. It is used for storing field, method, class-related reflection information. Space for this generation does not

come from Java heap memory settings (as given by **-Xms** and **-Xmx** flags to application at its startup). If an application experiences out of memory issues due to shortage in this generation, the most common cause is loading too many classes dynamically (such as Java Server Pages). A way to track loading and unloading of classes is by passing the flag **-verbose:class** at startup of the Java application. Another reason for permanent generation going out of memory is when an application uses a lot of **intern()**'ed String objects for better re-use. JVM tool **jmap**⁶ can be used to know details of heap generations. To know permanent generation details, execute it as **jmap -permstat <PID>**. It will provide details of how many Strings have been interned and their total size as well as details of each class loader, the number of classes loaded by them, bytes allocated, etc.

Note: With Java 1.7, the interned Strings reside in young/ old generation¹² and JVM parameter **-XX:StringTableSize** can be used to specify the string pool map size.

Another possible cause of Java applications going out of memory is when application reports back a message that a requested array size exceeded VM limit. This is usually due to miscalculation on the part of application logic and indicates an attempt to allocate an array whose size exceeds maximum heap size configured. Some applications tend to explicitly perform garbage collection by invoking **System.gc()** method. Use of this method is discouraged. It not only impedes functioning of the garbage collection process but also forces a major GC event causing garbage collection of both young and tenured generations. It is appropriate to disable such method calls through following flag to application at its startup - **XX:+DisableExplicitGC**.

It may also be important to note that distributed garbage collection invoked as a part of remote method invocation (RMI) also makes explicit calls to **System.gc()** to collect remote objects periodically. The following two arguments can be used to increase this collection interval - **Dsun.rmi.dgc.server.gcInterval** and **-Dsun.rmi.dgc.client.gcInterval**.

While we are on the topic of memory, it will be good to note that **top** command may show memory footprint as more than what is set through **-Xmx** flag. This flag decides the maximum size that a Java heap can grow to. The footprint as reported by **top** command includes heap size (young + old + survivor spaces), permanent generation as well as stack memory allocated to Java threads, and memory consumed by native calls such as through Java native interface (JNI). Hence, it should be a surprise that the **top** output shows value more than **-Xmx**.

Heap dump

While GC output points to a variety of issues that we may face due to GC pauses, it is still imperative to not produce so much garbage in the first place. Heap dumps are sometime necessary to know which objects are consuming a lot of space and their object graphs. Another reason for a Java process to consume a lot of memory could be memory leaks. A lot of applications use **Hashtable** (or equivalent) data structure. It is not uncommon for a leak to occur when developers add key-value pairs and do not make sure that the key objects are made immutable. Many hash code computations are done using the state variables of the object. A leak is now possible whenever developers change the state of the object from outside of the hash structure leading to change in its hash code implicitly. Such an object is now “lost” and is a dangling reference.

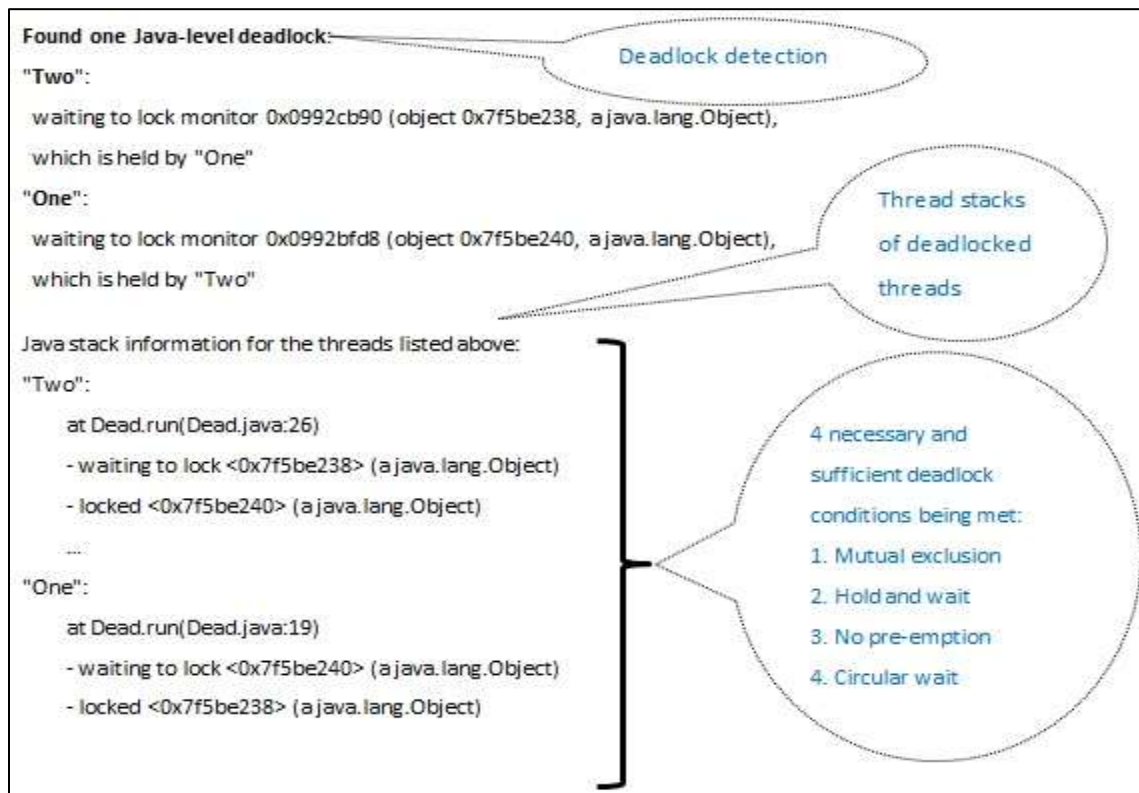
Heap dumps help point out memory leaks as well. There are many ways to force a heap dump. An appropriate time to generate heap dump may be when the application experiences an out of memory error condition. In such cases, if the application were started with the flag - **XX:+HeapDumpOnOutOfMemoryError**, then upon experiencing an out of memory error, the application will generate a heap dump. The heap dump is contained within a file **java_pid<PID>.hprof**, where PID is Java application’s process ID. In case we would like to have heap dumps in a specific location or change the name of the file, then - **XX:HeapDumpPath** flag can be used.

We can also use Heap Profiling (HPROF) tool to generate heap dumps. This can be used at application startup as **java -agentlib:hprof=file=heapdump.hprof,format=b**. This will generate a file by the name **heapdump.hprof**. The format specified here is binary.

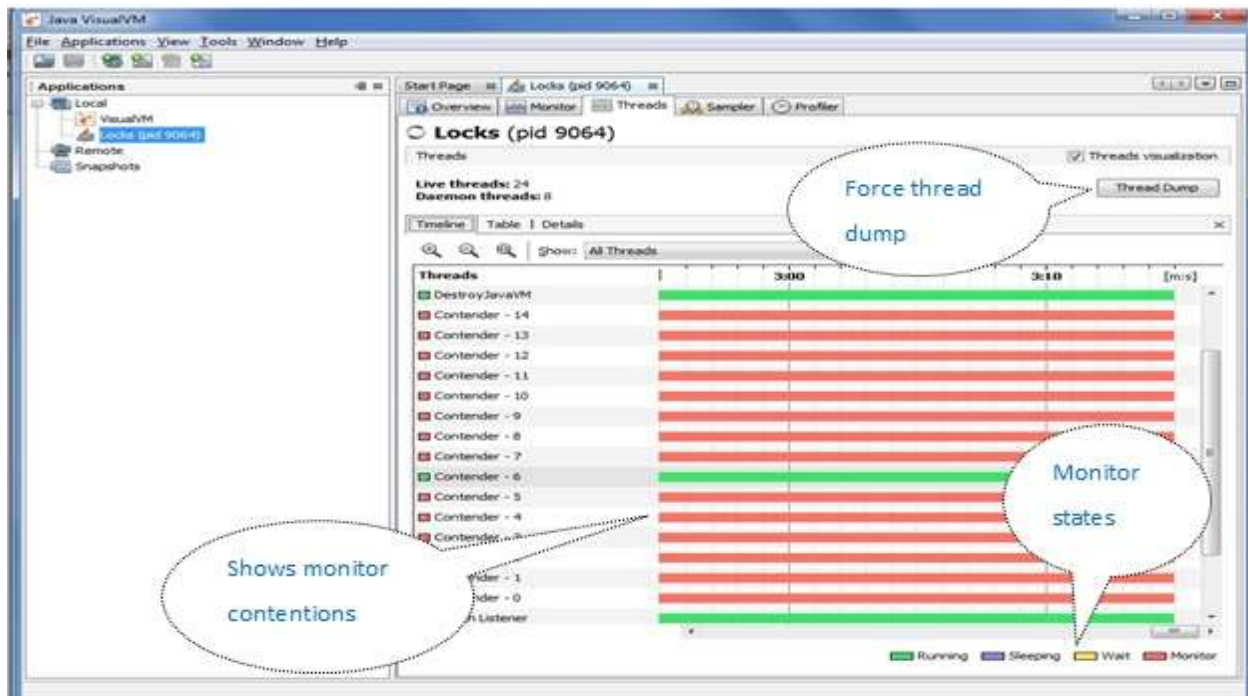
A heap dump so taken can be analyzed with **jHAT**⁶ (Java Heap Analysis Tool). This can be very useful to locate all the objects on heap and their references as well as understand why certain objects are not candidates for garbage collection as yet. Heap dumps should be generated in binary format for **jHAT** to process them. The tool can be used to view heap histograms which provide counts and sizes of each object on the heap when dump was taken. It is easy to browse each object graph. The graph is intuitive as it points to all objects which are reachable from any given object and summarizes total retention in terms of bytes and number of objects. This is usually very helpful in debugging object retention related issues as well as memory leaks.

Deadlock detection and lock contention

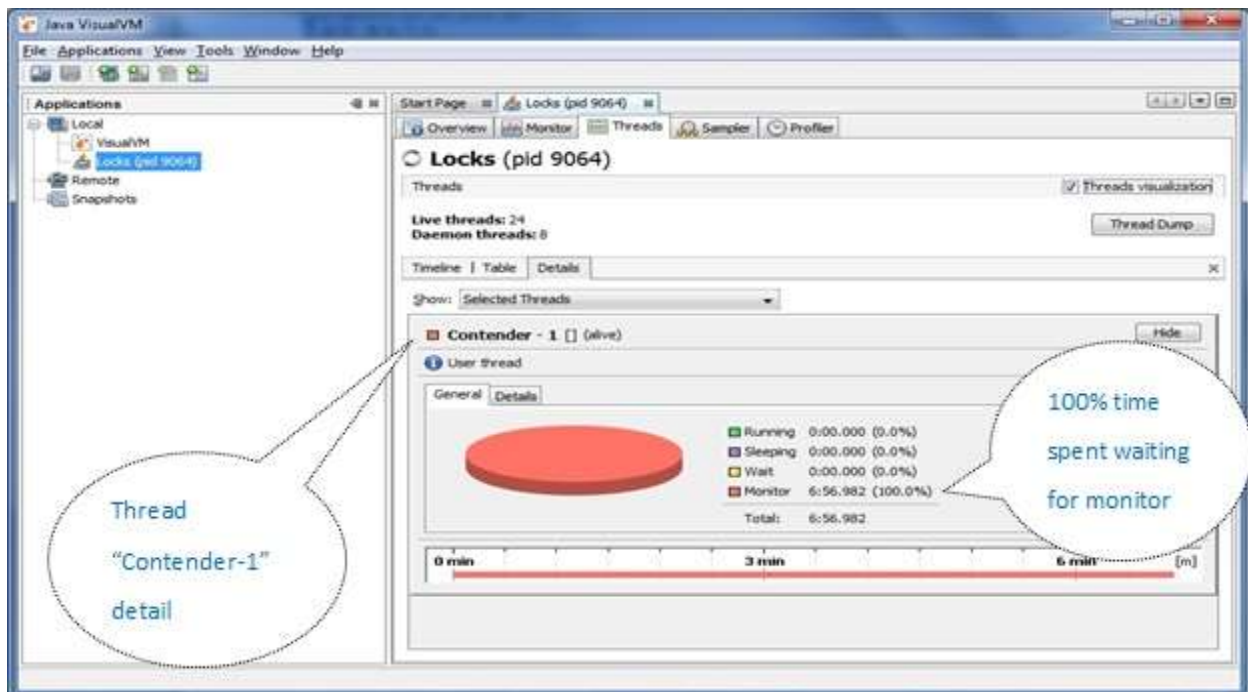
Java thread dumps also provide information about thread level deadlocks¹³. An example is shown below.



Another useful tool to use would be **jVisualVM**. As an example, the output below shows real time thread activities. The tool is not just limited to thread related information but can be used to profile heap memory as well as CPU utilization.



One can further browse individual thread level details by clicking on one of them.



As evident, clicking on one of the above threads shows that this particular thread is starving for a lock and has spent 100% of its time waiting to acquire this monitor. Generating a thread dump will show all of them in BLOCKED state.

Virtual Memory

Virtual memory is a scheme through which OS are able to provide a layer of separation between actual physical memory and logical memory that an application uses. This is achieved by combining random access memory (RAM) with a swap memory space, which may be carved out from hard disk itself. Executing **free** command provides information about memory usage. Here is one output obtained from command **free -m**. All values are in Mb (due to **-m** flag).

	total	used	free	shared	buffers	cached
Mem:	2023	1897	125	0	572	592
-/+ buffers/cache:		733	1289			
Swap:	2047	0	2047			

One mistake that people tend to make with this output is to interpret free memory as 125Mb. It must be noted that total free memory is obtained by summing entries under free, buffers, and cached columns and that is 1289Mb. This is shown in third row from the top. It may be noted that while memory allocated to buffers and cache is being used, it will be reclaimed and made available to applications should the system run low on memory.


Memory is laid out in smaller units, called pages, and applications use these pages to perform their functions. Page faults are a result of memory resident pages of an application being swapped out (to swap memory) and being required back in main memory. Some amount of swapping is not harmful and is acceptable. Excessive swapping is termed as memory thrashing and is detrimental to application performance. Thrashing happens when those pages are swapped out which are still in use and hence very soon, access to those pages causes swapping in again and this action may further send out other pages which are in use. Consequently, the system ends up swapping in and out pages and manages very little useful tasks. Main memory shortage is one of the root causes of thrashing to set in.

To know that an application suffers from thrashing, one can look at output from **sar**¹⁴ (System Activity Reporting) command. The **sar** output contains useful information (both present and historical) about system statistics. Most of the commands such as vmstat, iostat, ps, netstat, etc. provide a current snapshot of the system. Many times, we need to investigate issues which happened in the past. In such cases, **sar** provides a comprehensive record of system data. It

keeps track of memory, CPU usage, IO subsystem usage, interrupts, context switches, connection counts, and many others. As stated, it is very useful for observing historical trends.

Here is an output taken with **sar -W** command. This command reports page swapping statistics.

	pswpin/s	pswpout/s
08:10:01	4.28	1.72
08:20:01	1.19	1.09
08:30:01	11.78	3.09
08:40:01	5.17	6.81
08:50:01	126.36	704.25
09:00:01	0.65	1.00



The output shows that we had excessive swapping between 08:40 and 08:50. It must be noted that the application will appear to have come to a standstill, so **user (us)** CPU in **top** output would be low. However, the system is still pretty busy due to a majority of its time being spent in rectifying page faults and dealing with excessive swapping. Hence, **system (sy)** CPU component would show high utilization. Another thing to check at this time would be major faults that different threads are experiencing. We revisit **ps** command and request information on major faults experienced by each thread as given below (notice the last value within quotes as **majflt**):

```
ps -eLo 'lwp,ppid,cmd,psr,pcpu,etime,start_time,state,majflt' | grep java
```

The last column in output would now contain the number of major faults each thread has had. Some threads would definitely have non-zero entries in their last columns. Plus, many of them would show an increase in this number as you take multiple samples of the above command. These would be threads which are facing maximum penalty due to thrashing. The bad thing about thrashing is that the system may continue to thrash long after the condition which caused it is no longer present. This makes identifying root cause of thrashing difficult.

One more problem that memory thrashing can lead to is GC slowdowns. This can especially happen if portions of heap happen to overlay on swap memory. Memory thrashing does not necessarily mean RAM shortage. eXtensible markup language (XML) document object model (DOM) trees are also usual suspects as are **ResultSet** objects created out of database queries. As a good practice, it is advisable to keep track of sizes of all such objects.

It could also mean inefficient data structure usage as well as in-memory cache going out of control. Certain data structures also provide better spatial locality of reference (such as arrays or array-backed Hashtables or array-based ArrayList versus linked lists) and help minimize page faults. We will discuss more on how to monitor such intricate details when we discuss about Linux kernel profiling.

While on the topic of virtual memory, it may be noted that sometimes Java applications indicate an out of memory error with a detailed message indicating shortage in swap space. In such cases, JVM will create a detailed fatal error log. This log provides useful information about the system memory and thread which failed due to the error. Such an error may indicate (but not necessarily) shortage of overall swap space for the OS or could also mean a leak in native memory.

Connection problems

Sometimes the problem could be due to network level issues. In such cases, it is important to understand how our application fits in the entire ecosystem. For instance, what are all the important network elements that a request traverses through before it reaches our application and network elements encountered when our application makes requests to other applications?

Transport Control Protocol (TCP) knowledge is extremely useful in understanding issues of this nature. One of the commands that helps provide information about network level details is **netstat**. Executing **netstat** command with **-n** switch lists details about protocol being used by socket, receive and send buffers, local address, foreign address, and state. We can always filter TCP-specific sockets from such a command.

In the sample output shown below, the first column is the protocol (TCP in this case) being used. **Recv-Q** column shows the count of bytes that have been received by TCP stack on this host but not yet read off by Java application. **Send-Q** is number of bytes that are held up in TCP send buffer and yet to be relayed to target. **Send-Q** is directly impacted by network latencies and TCP sliding window sizes of the recipient TCP stack. Watching these two columns with multiple **netstat** output is useful to know whether we are experiencing issues. It is entirely possible that the TCP stack on target application is experiencing conditions such as zero window size.

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	Application initiated	68.221.252.3:52338	TIME_WAIT
tcp	0	0	socket shutdown and	68.221.252.7:5239	ESTABLISHED
tcp	0	12420	is now in final 2MSL	68.221.252.3:5033	ESTABLISHED
tcp	1	0	state.	68.221.252.2:5083	CLOSE_WAIT
tcp	0	0	128.221.252.112:5033	128.221.252.2:111	FIN_WAIT2
tcp	0	0	128.221.252.100:60232	128.221.252.112:500	SYN_RECV

Remote host initiated socket shutdown

We received ACK for our socket shutdown request

3-way handshake has been initiated by another host

One way to probe more would be to initiate a TCP dump using **tcpdump** command and taking a packet capture (pcap) in some file. A pcap file can be analyzed with tools such as Wireshark¹⁵. Here is an example of zero window condition as indicated by Wireshark analysis of pcap file.

Filter: tcp.stream eq 2019

Time	Source	Destination	Protocol	Length	Info
201121	277.187.187.104	10.0.2.196	TCP	8814	[TCP segment of a reassembled PDU]
201218	277.187.187.104	68.28.179.112	TCP	60	32656 > 10401 [ACK] Seq=510 Ack=244554 win=7300 Len=0
201219	277.187.187.104	68.28.179.112	TCP	7354	[TCP Window Full] [TCP segment of a reassembled PDU]
201322	277.271810.0.2.196	10.0.2.196	TCP	60	32656 > 10401 [ACK] Seq=510 Ack=251854 win=1460 Len=0
201323	277.271810.0.2.196	68.28.179.112	TCP	1514	[TCP window Full] [TCP segment of a reassembled PDU]
201413	277.271810.0.2.196	10.0.2.196	TCP	60	[TCP ZeroWindow] 32656 > 10401 [ACK] Seq=510 Ack=253314 win=0 Len=0
201740	277.631024	10.0.2.196	TCP	54	[TCP Keep-Alive] 10401 > 32656 [ACK] Seq=253313 Ack=510 win=6432 Len=0
201804	277.673381	68.28.179.112	TCP	60	[TCP ZeroWindow] 32656 > 10401 [ACK] Seq=510 Ack=253314 win=0 Len=0
202607	278.224917	10.0.2.196	TCP	54	[TCP Keep-Alive] 10401 > 32656 [ACK] Seq=253313 Ack=510 win=6432 Len=0
202663	278.267183	68.28.179.112	TCP	60	[TCP ZeroWindow] 32656 > 10401 [ACK] Seq=510 Ack=253314 win=0 Len=0
203380	278.271810.0.2.196	68.28.179.112	TCP	54	[TCP Keep-Alive] 10401 > 32656 [ACK] Seq=253313 Ack=510 win=6432 Len=0
203381	278.271810.0.2.196	68.28.179.112	TCP	60	[TCP ZeroWindow] 32656 > 10401 [ACK] Seq=510 Ack=253314 win=0 Len=0
205235	278.271810.0.2.196	68.28.179.112	TCP	54	[TCP Keep-Alive] 10401 > 32656 [ACK] Seq=253313 Ack=510 win=6432 Len=0

Time display

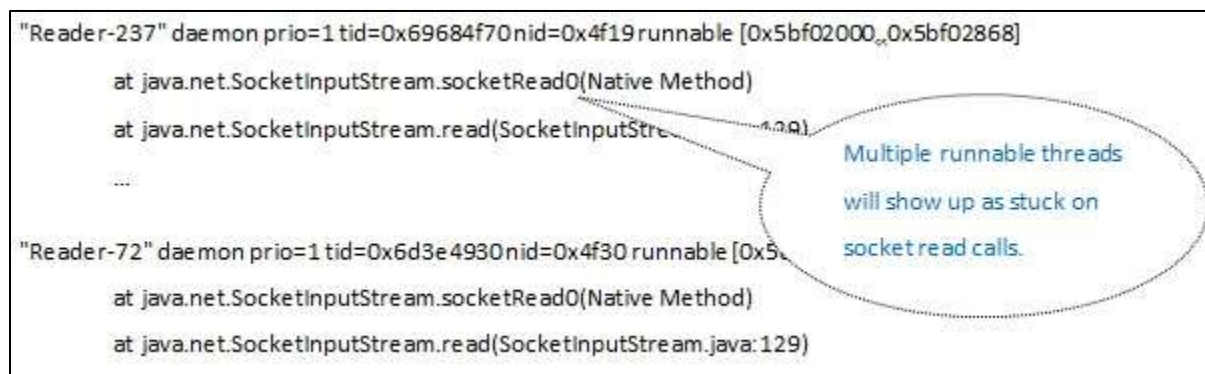
TCP window full

TCP zero window

Indicates bytes which are over the network

Header length: 20 bytes
 Flags: 0x010 (ACK)
 window size value: 6432
 [calculated window size: 6432]
 [window size scaling factor: -2 (no window scaling)]
 Checksum: 0x20ef [validation disabled]
 [SEQ/ACK analysis]
 [bytes in flight: 7300]
 [TCP Analysis Flags]
 [The transmission window is now completely full]

TCP zero window condition means that the target is overwhelmed or has just become slow. Slowdown could be due to internal application issues and can be looked at separately. TCP dump also provides information about time. One can change the display format of time to show actual system time against each line in the TCP dump. This is extremely useful in correlating TCP level information with application log time stamps as well as Java thread dumps. TCP zero window condition also proves that the network is not causing the slowdown in all likelihood. In zero window conditions, the sender side TCP dump will also show a higher value for 'bytes in flight' because that data is not yet acknowledged by the target application. If the Java application is the cause of slowdown, taking thread dump at this time might show multiple threads stuck at socket operations.



TCP allows certain flags to be passed between sender and recipient. One such flag Reset (RST) flag. A Java application that experiences a lot of **java.net.SocketException** with exception message as "Connection reset by the peer" is a clear indication of RST flag been set by the receiver TCP stack. One reason for such a behavior could be sudden shutting down of the target application or could also be due to target TCP stack being overwhelmed with incoming TCP connections leading to its listen backlog buffer being full. Listen backlog buffer is used by TCP stacks to queue incoming TCP connections as it may not have any dedicated connection to spawn.

Note: RST flag was mostly used to denote an abnormal session termination. These days, RST flag is also being used to shut down normal sessions. This is more of an optimization over the usual FIN-WAIT1→FIN-WAIT2 (client side) and CLOSE-WAIT→LAST-ACK (server side) state transitions. The usual FIN-ACK approach leads to a TIME-WAIT state while RST achieves instantaneous cleanup, wiping out the connection. Bottom line, RST flag alone does not indicate a problem. We must be noticing issues in Java application to investigate if it is a problem.

Many times, the slowdown may not necessarily be due to zero window conditions, network latencies, or application level slowdowns. One of the issues that an application can also run into is due to how TCP functions. TCP uses Nagle's algorithm¹⁶ and delayed acknowledgements. These address two different kinds of situations.

Nagle's algorithm tries to improve performance by combining many small packets before sending them out on network unless an acknowledgement for previously relayed packets is received and buffered data in sender's TCP stack has a 'PUSH' bit set (no matter whether maximum segment size (MSS) worth of data has been accumulated or not). The algorithm also allows sender to relay data if send timer expires, or more than half the send window size worth of data is ready to be sent.

Delayed acknowledgement applies more to the receiver end wherein acknowledgements for received packets are delayed until either enough packets have been received to acknowledge them all, or receiver has some data to send back along with the ACK, or acknowledgement timer has expired.

In this situation, both ends will delay the communication unless their timers expire and this will show up as latency in the application. One way to handle this would be disable Nagle's algorithm on the sender's side by invoking **java.net.Socket** class method **setTcpNoDelay(boolean on)** and passing it a Boolean false value.

We must also note that every TCP connection set up requires a finite amount of memory to be allocated and this is owned by kernel itself. Memory owned by kernel is non-swappable and can impact Java application performance leading to a lot of memory being held up due to a large number of incoming connections or even due to connections which are in the process of terminating. As an example, if we see a lot of threads in 2MSL (Maximum segment lifetime) state then it may not be a bad idea to reduce this value (after thorough load testing).

Besides socket level information, **netstat** can also be used to generate a lot of network level statistics. These can be stored and plotted to draw inferences. To generate statistics, we can pass **-s** switch as **netstat -s**. We will not go into full details of the output, but some of the information that is very useful here includes counts of total packets received, active and passive connections opened, connection resets received, delayed acknowledgements sent, TCP segments retransmitted, TCP data loss events, resets received for embryonic SYN_RECV sockets, etc. These can be very informative when we trend them over a period of time. For

example, consider resets received for embryonic SYN_RECV sockets count. This counter indicates that the client initiated a 3-way handshake but never completed the handshake with us and sent us back a RST flag after receiving SYN-ACK from our side. Sometimes this can also be indicative of SYN flood attacks.

While we are discussing slow application problems, it is important to acknowledge an important edge case which may make TCP appear unreliable. This usually happens when the host on which a Java application is running is very slow or its TCP stack is overwhelmed. What can happen is that clients may send in data and close the connection without our application having received this data. This amounts to data loss and ends up being a very difficult bug to troubleshoot. It can only be located with TCP dump analysis with a tool such as Wireshark. Here is what could happen.

The client sends data to our Java application. The TCP stack on our server acknowledges that data but has not yet delivered it to our Java application. This could be the case when our application is overloaded or experiencing other issues such as GC. When all data has been acknowledged by our server's TCP stack, the client is free to close the connection. At this point, our application has still not read the data off from TCP stack. It is likely that the TCP receiver eventually times out because our Java application was too busy to accept the incoming accept request. Once the timer expires, the TCP stack issues a RST flag and wipes out the data as well as the connection, which of course, does not alert the sender because it has already closed the connection amounting to data loss.

Context Switching

Context switching is the process by which an operating system pauses the execution of a thread or a process by saving its state (program counter, variables, etc.) so that it can be resumed from the same point later. Context switching allows for multiple threads to share CPU cycles.

Possible places in the Java application where a thread can get context switched include waiting for a lock on some object, explicitly giving up CPU by calling **sleep()** or **yield()** method on the thread, invoking a **join()** on another thread, waiting for I/O subsystem to fulfill a read/ write request, higher priority thread entering a RUNNABLE state, page faults experienced by thread due to its memory resident pages not being available in main memory, etc.

Excessive context switching is bad and causes applications to degrade. It is important to note that one of the reasons could also be spawning of many threads. When CPU has many threads

to deal with, it may context switch threads even before they are able to do anything useful. Usually, context switches in the range up to 4000 may not be that bad. However, if an application is performing slowly, those few 1000s can also be a problem. In general, a sudden huge jump in context switches is not a good sign.

A way to determine context switching values is through a command called **vmstat** which is useful in printing virtual memory statistics. One way to use **vmstat** would be take multiple samples over a period of time as **vmstat 2 15**, which instructs OS to emit information every 2 seconds and provide 15 samples.

procs			memory			swap		io		system		cpu			
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
2	0	4444	17776	60336	3335292	1	0	0	0	0	2	2	0	1	
2	0	4444	17660	60368	3335280	0	0	0	0	1781	1132	33	14	53	
2	0	4444	17828	60368	33				0	1843	1817	33	7	61	
110	0	4444	17628	60384	3335180	0	0	0	3466	1984	3267	63	22	15	0
387	0	4444	17500	60384	3335352	0	0	0	0	1979	6232	66	26	8	0
276	0	4444	17360	60412	3335296	0	0	0	2424	1800	4371	46	13	42	0
122	0	4444	18032	60412	3334688	0	0	0	0	1234	2491	18	9	73	0
113	0	4444	17220	60412	3335380	0	0	0	0	1582	2512	31	12	57	0

Output from **vmstat** shown above indicates a steady rise in context switches across the samples. If this is bad or not would depend on whether the application experienced any slowdown at the same time or not. If yes, did the slowdown progressively deteriorate application performance? If yes, then we may have a reason to worry. Also, most likely the application must have experienced slowdown because the first column indicates that it has a lot of processes waiting to run but were probably experiencing context switches making them slow.

At this point, an output of **ps** (as we discussed earlier) that lists only Java threads will be extremely useful. Another useful output at this time would be thread dumps. There may be more threads than what we assume to be present. If we have data as shown by **vmstat** above, it is usually easy to infer whether our Java threads are experiencing issues due to context switching

or not by looking at application logs. If the application prints logs for every important step that a thread performs, it should be easy to see long gaps in between their timestamps.

Other useful information in **vmstat** is about memory, CPU utilization, and bytes swapped in/ out. One more inference from this output is that because swap memory has stayed constant over the sampling duration, it must not have caused context switching to kick in. Another point worth noting in the output above is that, under the CPU utilization column, system CPU % (**sy** column) has shown an increase around the same time when context switches increased. This also points to considerable effort on the part of kernel as it is just busy context switching threads.

CPU affinity

Some Java applications can be more CPU-bound than being input-output (I/O) bound. CPU affinity¹⁷ is a way to tie processes to particular CPUs. This has some advantages. One of them is that when process gets context switched and rescheduled to run on the same CPU, chances are high that its state information will be still present in CPU cache. If it were allotted a totally different CPU, it will experience a series of cache misses and its state will either have to be retrieved from another processor or from RAM. Another advantage is that it prevents processes from being too greedy with respect to CPU cycles. So, if the application spawns multiple threads, those are now restricted to run on only those CPUs to which process has been bound.

Linux has **taskdef** command to tie the process to given CPU(s). CPU affinity ensures that a definite set of processors are always dedicated to our Java process. If we choose to implement CPU affinity for our Java process, we must make ourselves aware of a potential pitfall which surfaces due to interrupt handling in Linux.

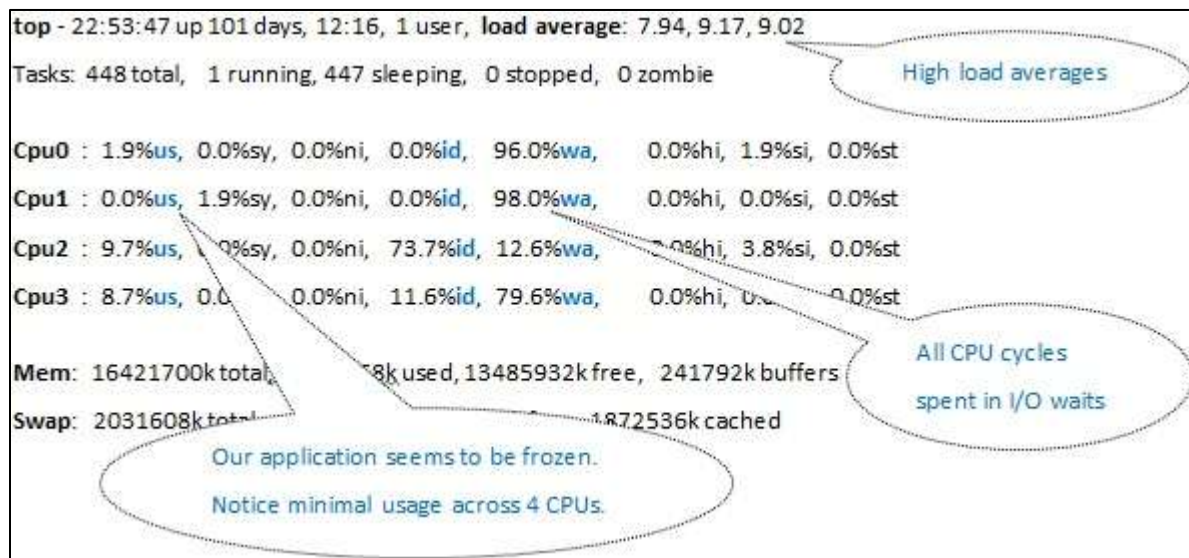
For many years, x86-based systems have had Advanced Programmable Interrupt Controller (APIC). The job of APIC is to deliver interrupts to various CPUs in the system. The pitfall here is that APIC delivers all interrupts to CPU 0 by default. Running a command like **cat /proc/interrupts** will indicate on most Linux systems that by default CPU 0 is handling all interrupts. This puts extra load on CPU 0 and we may well avoid making our Java application affine to this CPU. If we want to distribute interrupt handling amongst other CPU cores (only first 7 cores can participate in this distribution), we have to understand that for every interrupt request (IRQ) number, there exists a directory dedicated to each interrupt number under **/proc/irq/** directory. Each of these directories contains a file by the name **smp_affinity**. We will

have to update this file to change the default CPU to which that particular interrupt will be delivered for handling.

I/O Subsystem

Java applications can stall while performing input/ output to a storage device. It becomes critical to be able to troubleshoot those issues quickly because very soon a majority of threads can get 'frozen' waiting for I/O subsystem to fulfill their requests. This has the potential of the entire application coming to a grinding halt.

We will turn back to **top** command output first to see if we are experiencing I/O subsystem slowdowns. Consider the output below which is taken during the time of application slowdown.



Now that we have determined that our application is experiencing tremendous I/O waits, we next need to determine what is causing the delay. We should try finding whether the issue is due to the storage device or if our application is handling multiple concurrent requests for processing heavy-duty files on disk. To know which storage device may be the root cause of the problem, we can issue **iostat** command. To draw some conclusions, we can take multiple samples. Executing **iostat** with **-x** switch generates detailed I/O statistics.

Linux 2.6.18-128.1.1.6007.EMC (d29502) 12/28/2013						High I/O wait					
avg-cpu: %user %nice %system %iowait %steal %idle						Highly saturated device					
2.32 0.00 1.98 80.45 0.00 15.24											
Device:	rrqm/s	wrqm/s	r/s	w/s	rsec/s	wsec/s	avgrq-sz	avgqu-sz	await	svctm	%util
hda	2.46	46.93	5.80	18.31	66.08	344.04	17.01	0.03	0.16	0.20	0.48
hda1	0.00	0.00	0.00	0.00	0.00	0.00	6.65	0.00	9.55	6.70	0.00
hda2	0.12	1.58	0.21	0.85	2.67	19.44	20.84	0.01	1.13	4.93	0.52
hda3	0.70	35.21	6.10	20.50	222.50	2050.20	85.43	9.50	348.21	75.51	98.91

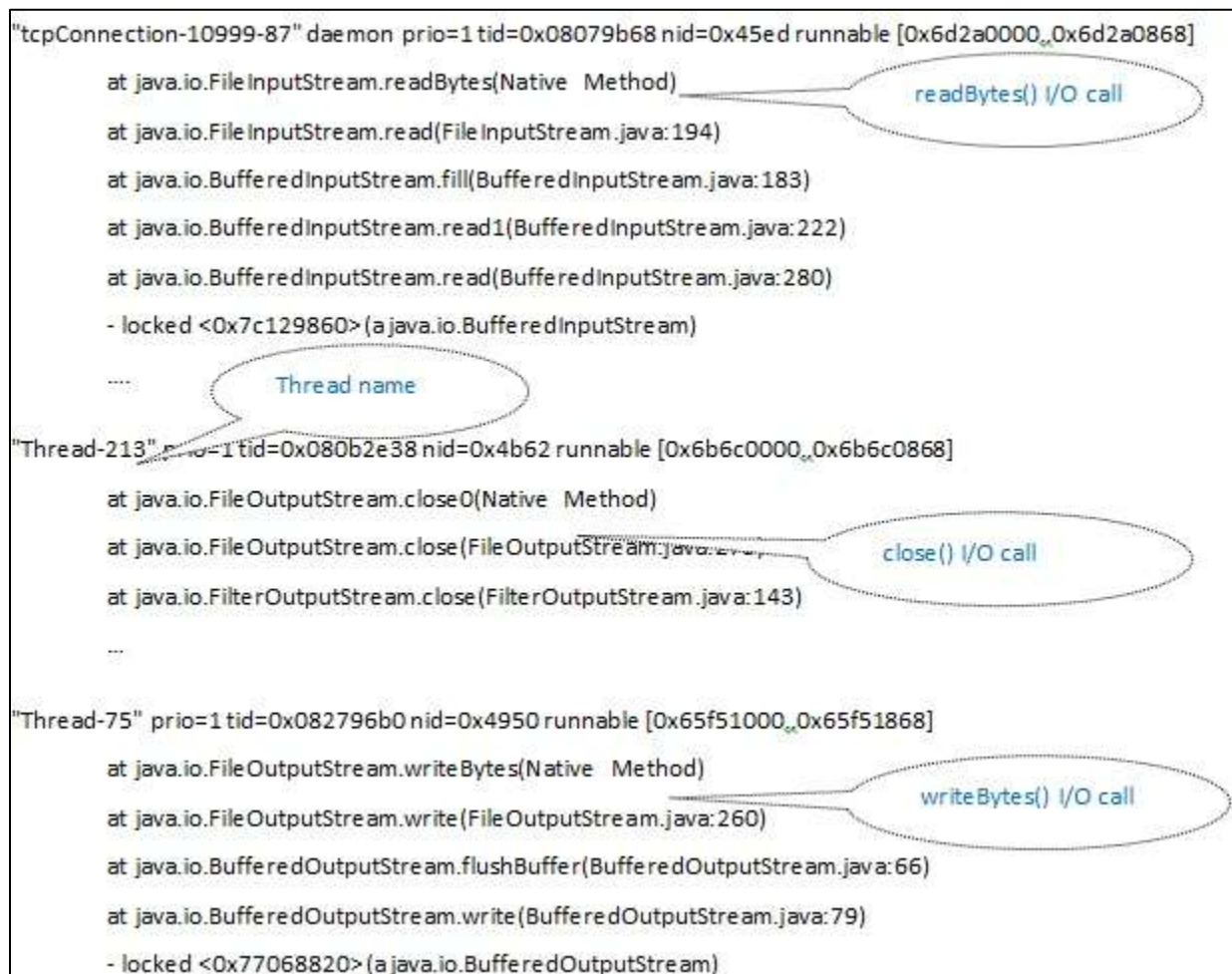
The first part of **iostat** output displays CPU utilization which we have seen in **top** command output as well. The bottom part of the output displays storage partition and device-wise statistics. These include read/write requests issued to the device (**r/s** and **w/s** columns) as well as average queue lengths of requests issued to the device. The most important parameters to analyze would be those under the last three columns. The **await** column indicates average time (in milliseconds) for I/O requests issued to the device to be served. This includes the time spent by the requests in queue and the time spent servicing them. The column **svctm** indicates average service time (in milliseconds) for I/O requests that were issued to the device. The last column, **%util**, indicates device saturation levels. If this value is close to 100%, the device is close to becoming saturated. Values higher than 100% indicate progressively more stressed out devices. A good practice to ensure a response-time sensitive application is to not allow for more than 70% utilization of the storage device. Thus, appropriately plan the capacity in advance.

Once we know which device is stressed and causing our application to stall on I/O operations, we need to find two more things. One, details of this device itself and two, Java thread dumps to see which threads are performing I/O operations and possibly which files they are working with. Knowing about the files can help us complete our root cause analysis (RCA) and correlate application behavior with I/O subsystem better.

To know storage device details is easy. In the case above, **hda3** was the offending device and we can find more details about it by just looking up its entry in **/etc/mstab** file. This file contains information about currently mounted file systems and provides details about the device such as mount point, file system details (ext3 or nfs or tempfs or other), along with other details. The mount point will point us to a location (such as **/** or **/proc** or **/var** etc) with which we can easily track back to application flows which may be performing I/O to it. As an example, if we

had all our logs being written to /var location then possibly we have overwhelmed the device and are possibly also running close to it being full.

As a next step, we can generate Java thread dumps and we should be able to locate those threads which are stuck while performing native I/O. We may see multiple threads showing up stack traces similar to those shown below. As evident in stack trace below, many threads will report as busy performing some operation or the other on I/O device. All of them show up as RUNNABLE threads, but as we may have guessed, none of them is making any real progress. We can confirm this by taking a few more thread dumps.



```
"tcpConnection-10999-87" daemon prio=1 tid=0x08079b68 nid=0x45ed runnable [0x6d2a0000,0x6d2a0868]
  at java.io.FileInputStream.readBytes(Native Method)
  at java.io.FileInputStream.read(FileInputStream.java:194)
  at java.io.BufferedInputStream.fill(BufferedInputStream.java:183)
  at java.io.BufferedInputStream.read1(BufferedInputStream.java:222)
  at java.io.BufferedInputStream.read(BufferedInputStream.java:280)
  - locked <0x7c129860> (a java.io.BufferedInputStream)
  ....
  Thread name

"Thread-213" prio=1 tid=0x080b2e38 nid=0x4b62 runnable [0x6b6c0000,0x6b6c0868]
  at java.io.FileOutputStream.close0(Native Method)
  at java.io.FileOutputStream.close(FileOutputStream.java:142)
  at java.io.FilterOutputStream.close(FilterOutputStream.java:143)
  ....
  close() I/O call

"Thread-75" prio=1 tid=0x082796b0 nid=0x4950 runnable [0x65f51000,0x65f51868]
  at java.io.FileOutputStream.writeBytes(Native Method)
  at java.io.FileOutputStream.write(FileOutputStream.java:260)
  at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:66)
  at java.io.BufferedOutputStream.write(BufferedOutputStream.java:79)
  - locked <0x77068820> (a java.io.BufferedOutputStream)
  ....
  writeBytes() I/O call
```

Note: Using **iostat** with a **-n** switch, will also give details of network file system (NFS) mount points. Both **-x** and **-n** switches are mutually exclusive.

Thread Names

One shortcoming in our dumps that we have analyzed thus far is that thread names are not very intuitive. For example, in the dumps above, if we could identify which files these threads were trying to work with, it can provide ultimate clues into troubleshooting the problem. Java thread API provides a very easy way to set names for the threads. This can be done while creating the Thread object or by explicitly calling **setName(String)** method on Thread object. Most times, our Java application threads will be performing operations which are unique in their own way. As an example, we may have an application which assigns a unique ID to every message that gets in to the system. Or, the user on whose behalf an operation is being performed may be unique in terms of user ID. Plus, it is a good idea to have unique IDs being tagged in application logs so that debugging becomes easier. Hence, setting a unique ID as a name for each thread can be extremely useful in troubleshooting applications. As we discussed, we could simply invoke **setName(String transaction_id)** every time we create a new thread dedicated to a given transaction or take one from a pool of threads. If we take one from the pool, it will be important to reset the name in a *finally* block by invoking **setName(String original)** method again before we return the thread back to the pool.

File descriptors

If we are interested in listing all open files at any instant, we can use the **lsuf** command. This command can be used to filter only those files which are opened by Java processes by invoking it as **lsuf -c java**. In the above example, we can run this command and know all the open files at that instant. Many times, a Java application fails to call **close()** method on various File and streams that it works with. This leads to file descriptor leakage which eventually leads to **IOException** with a message indicating “Too many open files”. Linux provides a way to increase these descriptor limits (and other aspects as well, such as number of threads, default thread stack sizes) through the **ulimit** command. However, those usually provide only temporary respite at best.

Such issues can easily be tracked by executing **lsuf** command as listed above. If the application file names are unique (say, those are messages on disk or data pertaining to unique transactions, etc.), it is safe to assume that each thread name can be set to be the same as those unique IDs. This has multiple benefits. First, these thread names will show up in thread dumps which will make it very easy to correlate exact transaction or message that they represent at the time of dump being generated. Second, if we issue an **lsuf** command at the time of taking thread dumps, it will also report the exact path to the disk for those open files.

Correlating thread dump of the thread dealing with this file (as it now has the same name as file name) will be a very easy matter now.

It may be noted that **lsuf** will list all open file descriptors which includes socket connection as well. It provides much more information, such as size of file being read, mode of operation (r, w, etc), node numbers, etc.

Tempfs and ramfs

If our Java application is getting slowed down too often due to disk accesses, it may benefit from directly creating those files on a portion of RAM allocated for them. This would make the I/O process faster. Tempfs and ramfs are two means to dedicate a portion of RAM and be used as a separate partition. A Java application can continue using **java.io.File** application programming interface (API) or New I/O (NIO) package to deal with these files. From an API abstraction point of view, there is no change at all.

Volatility of RAM makes it unsuitable for storing critical data. However, many applications generate a lot of temporary files and those can be stored on Tempfs or ramfs. Both tempfs and ramfs can be mounted with **mount** command. It must be noted that there are some differences between the two and we must be aware of them. Tempfs honors the size restriction put on it when it was mounted. This means that once this limit is reached, it will error out with the infamous “no space left on device” **IOException**. It may also be noted that tempfs can use swap portion as well. However, the size limit specified applies to its overall footprint on RAM + swap. On the other hand, it does not make any difference if we specify size for ramfs as it does not honor those limits and grows dynamically as long as there is free space on RAM. Once RAM is exhausted, it will not use swap memory space at all and report back similar error.

Application logging

Seasoned programmers will agree that application logs are the most important tool for troubleshooting any application. While there may be many guidelines about what to log, we will focus only on those which ease the task of troubleshooting.

First and foremost, every log line must have an ID tagged to it that identifies the transaction it is a part of. In a production system, multiple log statements will interleave and unique ID tag will help segregate application flows easily. Second, every log statement should print the thread's name that emitted the log. This helps correlate application with thread dumps and if our application is setting some intuitive thread names, this becomes a big plus. Next, we must log

time stamps with every log to the granularity of milliseconds. It is also important to print in logs the URLs to every third party application which our application connects to. Logs must also print names of application-specific entities such as files being accessed, messages being handled, transaction being performed, etc.

Developers should be encouraged to write logs which can be easily parsed with scripts. Always put whitespace between printing the log and appending variable's value to it. It is always a bad idea to print logs while holding another lock. It is also a bad idea to print data which requires acquiring another lock (such as printing size of Hashtable object).

It is absolutely critical not to hide any exception. Print all exceptions to logs and values of method parameters around the time exception was generated. If our application has log rollover policies set up, it is important to plan for enough capacity to retain logs long enough to be able to review them when and if the need arises. Even older logs are best stored for some time as they can offer great insights into application behavior from a historical perspective.

Writing to **System.out** is always a bad idea as those may get redirected to **/dev/null**.

It is critical to monitor a newly released feature closely and logs are an important tool in this regard. If we do not introduce logs around new features, we will not be able to investigate whether the issue we notice is due to software changes or not.

In distributed systems, passing along a unique ID (could be transaction ID, message ID, etc.) between applications and printing it in their individual logs can help correlate flows which span multiple systems.

It may also be a good idea to store logs in a common place and index them on important fields. Apache Lucene/ SOLR can be useful tools in indexing the same. Storing logs in a common place has another advantage. Powerful analytics can be built on top of such logs using statistical and mathematic models which can further help with complex troubleshooting analysis and application behavior predictions.

Virtualization

Increasingly, more applications are beginning to take advantage of virtualization. It is possible to run multiple JVMs on the same virtualized machine. All the issues we have discussed so far apply here as well.

However, an additional checklist item gets in place when we size the virtual machine's memory requirements. This should ideally be calculated taking into consideration the memory that would be needed by each guest OS. Failing to do so can result in main memory problems for our Java applications which we discussed above. Consider a case where we have run our Java applications on VMware ESX/ESXi virtualization hypervisor. ESX/ESXi can use swapping or memory ballooning to dynamically control the amount of memory being allocated to virtual machines. Ballooning is the preferred way and requires a balloon driver. Swapping is used when either the driver is not available or is unable to reclaim memory quick enough. Since, swapping is detrimental to performance, it is important to reserve the memory for Java process by taking into account its entire footprint (heap + stack + permanent generation) as well as guest OS memory requirements.

Timekeeping is critical to troubleshooting issues. With virtualization, it becomes tricky¹⁸ as to how the guest OS remains in sync with actual time as tracked by the physical machine. In case CPU utilization is high and the Linux OS under question uses tick counting as a means to keep track of time, it is possible that this guest OS falls behind actual time. This is bad because we may have problems in correlating events with real time and this can impact our ability to troubleshoot effectively.

Tick counting mechanism is used by many OS and involves interrupting the CPU periodically to keep track of time. With Java applications, this can impact its performance and one must look at output of **top** and **vmstat** to keep track of interrupts. A way to address this problem would be to use an external Network Time Protocol (NTP) source.

When working on virtual machine-hosted Java applications, it is usually accurate when it comes to reporting guest OS specific context switches, interrupts, CPU usage, and memory usage.

One more aspect to be aware of when using virtualization is about accessing functionality provided by hypervisor from a guest OS hosting our Java application. This introduces some complex touch points (entry and exit) between the two. Do note that these touch points are not being explicitly called by Java application but are required in any virtualized environment. This can have a direct bearing on the CPU cache contents being flushed. This is more so because the hypervisor code itself now needs its data and code to be present in cache lines. This causes CPU cache to be likely flushed of data that our application would have needed.

One of the most important caches is Translation Look-aside Buffer (TLB) which maintains a mapping of virtual to physical memory addresses. These translations are costly and this is why a fully dedicated cache (TLB) is used to store these. Since cache is affected every time upon entry to the hypervisor, it may be helpful to enable support for large memory pages. Increasing page size¹⁹ will allow for a single TLB entry to represent larger memory range. Keeping these aspects in mind, we can pass the following flag at the startup of our Java application - **XX:+UseLargePages**. We will also have to enable support for usage of large pages in our kernel (Linux kernel 2.6 onwards) by appropriately setting **/proc/sys/kernel/shmmax** and **/proc/sys/vm/nr_hugepages**.

If our application continues to run in a degraded manner then it may be possible that the hypervisor has swapped our application altogether.

An important parameter to also look at would be stolen CPU field in top output. This is usually the last column (we briefly mentioned this when we discussed top output in detail). This column indicates CPU time that our application was ready to run but the hypervisor chose to run other applications. This can also be regarded as virtualization penalty. Here is an example from top which shows stolen CPU time:



Summarily, while virtualization has resulted in great savings in terms of hardware consolidation, its cost²⁰ can have an impact on our application.

System level tracing

Sometimes the issue cannot be debugged within the realm of Java application alone and we may have to enable system level traces. Consider a case where requests are not reaching our Java application at all (at least we do not see traces in our logs). We do not know whether there is an issue with our application not logging enough information, or whether the request never even reached our application. In such cases, **strace** command allows us to enable system level trace and we can see system level calls being made.

Consider another case, where certain requests that were reaching our application were causing it to crash (possibly with a segmentation violation). We would like to find out more about what

these kinds of requests are. Let's troubleshoot this case using **strace** and also hope that we gain some insights into this wonderful utility.

Before we begin troubleshooting this issue, here is something to think about. To know what may be causing the application to crash can be a daunting task. What we can attempt at least is to divide and conquer rather than trying a brute force method to think about possibilities. One way would be to stop all incoming traffic to this application by disabling it on the load balancer. It can then be checked whether the crash is caused due to some internal trigger or not. This way we may be able to eliminate many possibilities and reduce our troubleshooting surface area. Once we know the PID of our Java process (using **jps** or **ps**), we can trace its system calls as **strace -f -t -r -p 21847 -o /path/to/output/21847_strace**

There are many more options available but we will list here details of the ones we have used above. The **-f** flag instructs to trace the forked child processes as they are created. This is important to be able to trace all our Java threads. The flag **-t** is to print time of the day against each entry. The **-r** flag is to print the relative timestamps against each entry. The **-p** flag is to specify the process ID and is followed by the numeric PID value. The **-o** flag is to store the output of **strace** command to some file for later analysis.

It is very easy to follow the chain of events in **strace** output given above. First column is the PID of the child thread. Next two columns list the timestamp and time taken for a given call. As can be seen, this output shows that an incoming HTTP HEAD request was made for the resource /heartbeat. Immediately after that, the process terminates with a segmentation violation. This output also shows that no other processing seems to have taken place inside our application and hence, it is safe to assume that closer scrutiny of incoming HTTP HEAD request is required and is most likely the root cause of this issue.



Note: As an aside, troubleshooting for this issue was done exactly as explained here. Since, this application crashed intermittently, the first thing to find was which request(s) or internal processing may be causing this to happen. Another aspect about this application was that it is enabled both on an external as well as an internal virtual IP (VIP) and is load balanced. To know what could be the root cause, one node (running this application) was disabled on both internal and external VIPs. This was to ensure that there are no application threads which are causing it to crash. Maybe the application starts daemon threads at its startup which were somehow causing this crash. The application did not crash, so next it was disabled on external VIP but enabled on internal VIP. The application on this node crashed soon. This proved that most likely the issue was internal. Next, it was enabled on external VIP and disabled on the internal VIP. It did not crash again which proved that external requests were not the root cause.

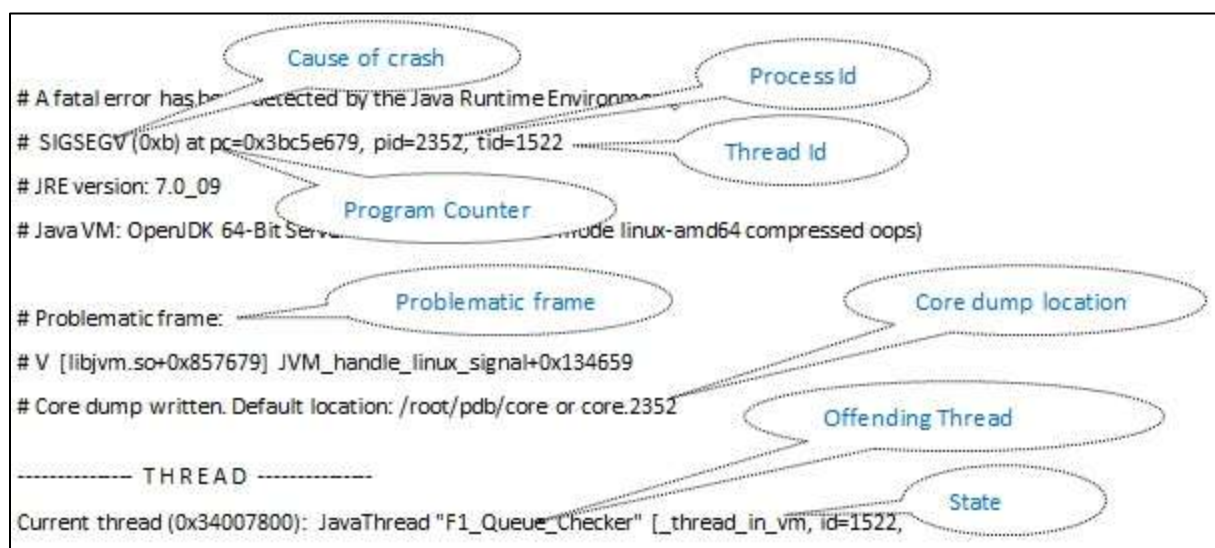
With help of **strace** it was determined that the application crashed whenever a monitoring application sent it heartbeat requests. It turned out that the monitoring application had undergone some upgrades which resulted in corrupting a few HTTP HEAD requests. These requests landed on Apache HTTP server which front-ends the Java application. The requests never made it past Apache to be logged in our Java application logs.

It should also be noted that **strace** output can be used to also figure out whether our application is experiencing deadlocks. If the application threads seem to be stuck on **futex**²¹ (Fast Userspace Locking) system call, then most likely application threads are deadlocked because **futex** is typically used to implement the contended case of a lock in shared memory.

Crash analysis

The topic of application crash is so broad that theorists could write a full book dedicated to it. The possibilities are many more than what we are able to discuss here.

When a Java application crashes, it will produce an error log by the name **hs_err_pidXXXX.log** in the working directory of the application. This file provides a lot of information about cause of crash and problematic application frames. Location as well as name of this file can be changed by passing the **-XX:ErrorFile** flag at startup of a Java application. As an example, this flag can be set as **-XX:ErrorFile=path/to/dump/java_error%p.log**, where **%p** is useful to tag process ID to the file name. It also contains information about the offending thread whose action led to the crash. Some of the details provided here include thread name, its state, thread type, thread ID, etc. Note in the crash log snippet below that that type listed for offending thread is **JavaThread**. Other possibilities are **GCTaskThread**, **VMThread**, **CompilerThread**, **WatcherThread**, and **ConcurrentMarkSweepThread**. Each of those is a special purpose thread within the VM. We have already come across **VMThread** and **GCTaskThread** earlier. **CompilerThread** is dedicated to HotSpot compilation of code. **WatcherThread** performs task of running periodic operations in VM. It is a native watcher thread which simulates a timer interrupt waking up every 50ms.



The problematic frame can be VM frame (prepended with letter V in this case). It could also be a C language native frame (prepended with letter C) among others. The state defines exact operation that the thread was performing. The following table summarizes important states.

_thread_in_vm	Thread executing VM code
_thread_in_native	Thread executing native code
_thread_blocked	Thread in blocked state
_thread_new	Thread created but not yet started
_thread_in_Java	Thread running compiled/ interpreted Java code

Next, the error log contains information about the signal which caused the crash, register context, top of stack information, and following opcodes close to the program counter (PC) when application received faulted and crashed. These are followed by a section on thread stack.

Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)

V [libjvm.so+0x857679] JVM_handle_linux_signal+0x134659

j sun.misc.Unsafe.getAddress(J)J+0

j Direct.main([Ljava/lang/String;)V+50

v ~StubRoutines::call_stub

V [libjvm.so+0x53b0ce] AsyncGetCallTrace+0xd78ae

V [libjvm.so+0x5483e7] JNI_CreateJavaVM+0x12e7

C [libjli.so+0x3a5c] printf+0x3a5c

Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)

j sun.misc.Unsafe.getAddress(J)J+0

j Direct.main([Ljava/lang/String;)V+50

v ~StubRoutines::call_stub

Explanation of frames

Java frame causing the crash

There are details about memory map which follows and can be used to locate where exactly the program counter (PC) points. We may recall that PC is displayed at the beginning of this error file. A memory map of a process contains information about libraries it uses as well as their locations (both code and data segments). Here is one example:

<div data-bbox="397 210 722 283" data-label="Text">Memory range</div> <div data-bbox="219 283 755 367" data-label="Text"> <pre>3b405000-3b406000 r--p 00082000 fc:00 1715012 3b406000-3b407000 rw-p 00083000 fc:00 1715012</pre> </div>	<div data-bbox="1112 231 1404 304" data-label="Text">Library Name</div> <div data-bbox="852 283 1047 367" data-label="Text"> <pre>/lib64/libm-2.12.so /lib64/libm-2.12.so</pre> </div>
---	--

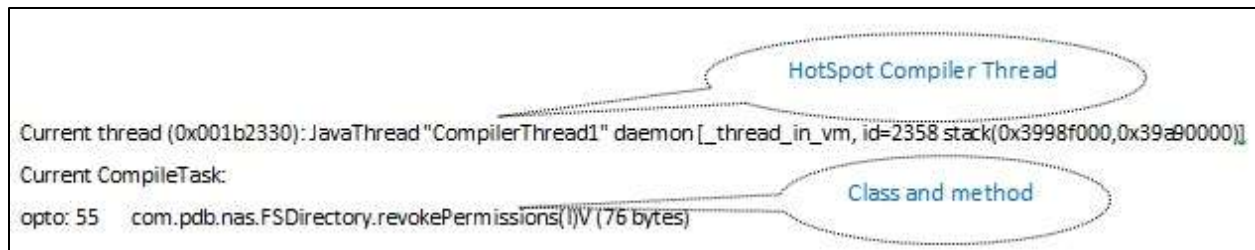
In this process map output shown above, the first column shows the memory span of the library. This can be extremely useful in mapping the program counter to identify the exact offending library. The second column is easy to determine which of them is code versus data segment. The one with 'w' bit set is data segment.

Depending upon the severity of issue, sometimes a full error report may not get generated. It is even possible that process map output is not included. In such cases, it can be useful if we generate process map of our application using **pmmap** command. The same information can also be found in **/proc/<PID>/maps**, where PID is process ID of our Java application. It must be noted that there is no point in taking a process map if our application has already crashed as there won't be one existing.

Note: In case the application uses Apache projects such as Lucene and SOLR for indexing the documents with Java 1.7, a bug²² in the hotspot compilers can cause index corruption²³ and also lead to segmentation violation issues. This has to do with two default VM option flags which are turned ON by default. These are **-XX:+OptimizeStringConcat** and **-XX:+AggressiveOpts**. Those are also present in release 1.6 but are turned OFF by default. The first of these flags tries to find opportunities to optimize Java String concatenation while the other turns ON compiler optimizations that are expected to default in future Java releases. Index corruptions in Lucene happen due to bugs which wrongly compile certain loops. This can be avoided by disabling those optimizations using JVM flag **-XX:-UseLoopPredicate**

HotSpot Compilation

HotSpot²⁴ component in Java Standard Edition (SE) performs multiple functions including adaptive compilation of Java bytecodes into optimized machine instructions. HotSpot keeps track of which portions of a Java application are being used more than the others. These methods are assumed to be in hot spot and the HotSpot compiler compiles them using Just-In-Time (JIT) compilation and further, heavily optimizes them on-the-fly. Let us consider the case of application crash caused by the HotSpot compilation process. Here is an example of a crash encountered due to HotSpot compiler thread:



It is evident from snippet above that the current thread is **CompilerThread0**, which is one of the many compilation threads in the HotSpot Server VM. One way to handle this issue is to start Java application with **-client** flag, which is unacceptable in most production software as it takes away many optimizations that Server HotSpot VMs provide. The crash file output pasted above clearly lists which method compilation led to a crash.

A better solution would be to exclude this method from HotSpot compilation itself. This can be achieved by creating a special file by the name `.hotspot_compiler` in the working directory of the Java application and adding a line in it as given below. Upon restart of the application, HotSpot compiler will consult this file (if it exists) and exclude this method from its compilation process.

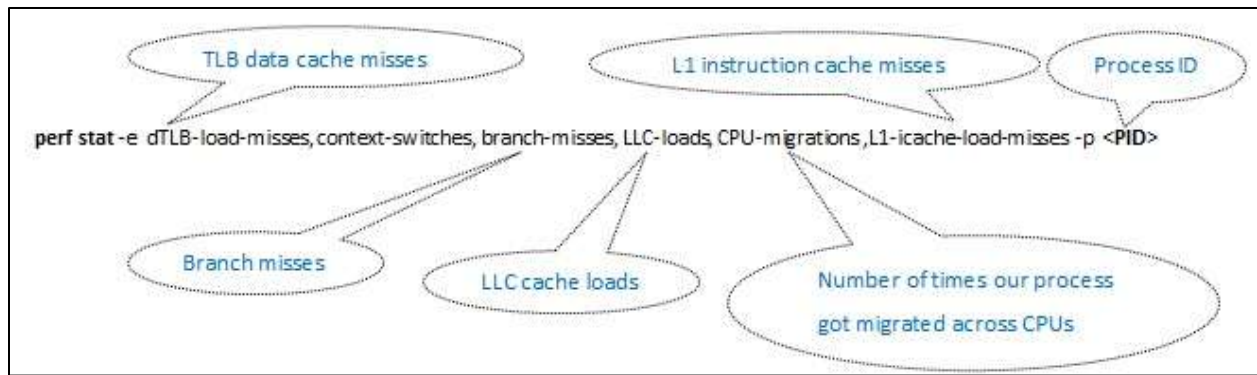
```
exclude com/pdb/nas/FSDirectory revokePermissions
```

It is possible to change the default location and name of this file by using JVM flag **-XX:CompileCommandFile=/path/to/command/file**. If we want to completely restrict compilation to only a handful of methods, those can always be passed through command line argument to the Java application using the flag **-XX:CompileOnly**.

Exclusion instructions can also be given at startup of the Java application using the following flag **-XX:CompileCommand=exclude,com/pdb/nas/FSDirectory,revokePermissions**

Linux Kernel Profiling

Linux kernel 2.6+ provides an easy to use command-line tool called **perf**²⁵ to find many intricate metrics about our running Java process. The tool provides easy means to collect and analyze data such as cache misses, i.e. L1, TLB, Last Level Cache (LLC), cache pre-fetches, branch misses, times our process was migrated to a different CPU, etc. It can be executed as:



Profiling with **perf** tool as shown above can be very useful in tuning the application. By default, **perf** measures these metrics at user level. However, we can append each of the metric above with a modifier to denote the measurement level. For example, cache-misses:**G** would provide metrics for **G**uest machine on a virtualized environment. Similarly appending **H** would do the same at **H**ost machine level; **h** would monitor **h**ypervisor events, etc. Monitoring at a specific level can be very useful to also understand the cost of virtualization. It can also be very useful to understand whether our algorithm or process is causing many cache misses (L1, LLC data and instruction), wrong branch predictions or others events and we can tune it, if needed.

Monitoring

We have discussed troubleshooting at length in preceding sections. Let us turn our focus to monitoring aspects. It is important for applications to provide hooks that enable effective monitoring and early diagnosis. Proactive monitoring forms the core of effective troubleshooting.

As a first step, the key aspects to identify during software construction phase are those which are directly influenced by applications and may require active monitoring. These would include memory, sockets, threads, and storage to name a few. These aspects may be directly influenced by applications and we will discuss how to monitor them in real and historical times. It is important to define thresholds for these and trigger alarms in a timely manner.

OS specific

We have discussed how important monitoring aspects are, such as memory usage, file descriptor counts, CPU usage, load averages, network statistics, thread counts, disk IOPS, disk space, context switches, interrupts, major fault counts, bytes transacted, etc. It is extremely useful if we are able to plot these metrics to spot trends.

Application specific

1. **Queue depth** – Applications may provide for various queues to store messages, files, or other information. It is important to monitor queue depths and set alarms on them. It is also a good architectural practice to follow separation of concerns and separate out queues according to functions they perform. As an example, applications may have a specific queue for incoming traffic bifurcated by type (i.e. HTTP GET requests versus HTTP POST requests) or another high level function (i.e. buying an item versus selling an item). Keeping them separate not only allows scaling them separately but paves the way for easier monitoring and troubleshooting. Further, applications may provision a separate queue for messages which failed for temporary reasons and retry them after some time. Monitoring and alarming on such retry queues can be useful to know if a 3rd party application is responding too slow or failing to respond altogether.
2. **Latencies** – Keeping track of internal application latencies is important. It is easy to keep track of running averages using some simple data structures such as a circular queue. It is useful to remember important numbers when it comes to latencies²⁶.
 - a. **Latencies bifurcated with respect to each application flow** – It is important to keep track of application flow latencies. A suggestion would be to separate them with respect to functions they perform. A messaging system may want to

separate them with respect to time it took to decode some incoming message, time it took to execute some specific algorithms, etc. It may also want to keep track of time being taken on an average to write some data to a queue or read from it. Keeping track of latencies has another indirect bearing. It helps make design decisions better. One can easily understand whether adding extra processing to some flow can lead to latency specific service level agreement (SLA) violations.

b. **Latencies bifurcated with respect to every 3rd party application invocation.**

It is important to record time an external application takes to perform its processing. For example, a telecom provider may want to keep track of latencies with respect to time it took to make a call to billing system and relay call data or how much time it took to authenticate some user by invoking a dedicated authentication application. Those add to overall flow latencies and we have to know those upfront. A good architecture may ensure enough insulation against such external factors say, by means of making the flows asynchronous. However, this is not always possible.

c. **Latencies with respect to clients.** This is extremely useful to know whether the slowness in the application as being perceived by clients is also being contributed to by them as well. A typical slow client (say, who comes over a slow connection or itself is slow in pushing data or has buggy TCP implementation) can consume a lot of application time and resources. Monitoring time taken by clients to send in all data across is a good practice. Client side latencies may also show up as “**java.net.SocketTimeoutException: Read timed out**” exceptions. Certain HTTP layer applications such as nginx provide for a dedicated response code (HTTP 499) which can keep track of such incidents.

3. Error counts

a. **With respect to internal errors and exception cases** - Keeping track of errors internal to an application can be a life-saver when troubleshooting issues. An appropriate example would be maintaining a counter to keep track of transactions which were aborted internally for some reason. If possible, the aborted transaction count should be further bifurcated into subcategories. For example, our application may have separate counters for transaction aborted or dropped due to malformed content, or due to authentication failure, or due to insufficient funds. Monitoring and alarming based on such a counter can be

extremely useful in times when this counter shows sudden variation. It can be particularly insightful to monitor this counter before and after a production software is upgraded. Decisions to rollback an upgrade can be easily made when such a counter is enabled and alarms set on it. Such counters can be useful to ascertain quantum of data loss as well.

b. **With respect to 3rd party application returned errors and issues.**

Applications do not work in isolation. They coordinate with other applications to provide services to end users. Many times, an error in a 3rd party application can cause the application to either error out or degrade considerably. A good architecture would prevent entire ecosystems from collapsing and ensure graceful degradation. The architecture may or may not have taken necessary precautions such as allowing for graceful degradation. However, monitoring 3rd party application errors are critical to troubleshoot and isolate issues. Those errors should include (but not be limited to):

- i. **Connection counts and failures** – We must keep track of all connection counts and connection failures with respect to all applications which we transact with in separately dedicated counters for each of those applications.
- ii. **Application specific errors and exceptions** – We must keep counters for specific error responses we receive from other applications. It is always helpful to have separate counters for the most important (and expected) error cases and bucket the remaining in a single counter.
- iii. **Retry later counts** – Many times, applications can respond to us by indicating that we should retry the flow later. We must dedicate a separate counter to keep track of such incidents.

c. **Error counts with respect to clients who invoke the application -**

Applications which receive incoming traffic from client applications (such as browsers, handheld devices, client application software) are best safeguarded by keeping active track of errors which are originating from certain clients. Such monitoring counters can be extremely useful in ruling out whether it is our application at fault or not. When keeping track of errors with respect to browser and mobile clients, it is useful to segregate them into different user agents. Many HTTP servers (nginx or Apache) allow for logging details of incoming requests such as source IP, destination IP, request type, request size, the response being

sent by our application to that request, etc. All of them also have a provision to log time taken by the request to be served. This is an important parameter and can be monitored separately.

4. **Count of each application flow invocation** – Keeping a counter to track application flows is always useful when ascertaining if there is a sudden surge in flow compared to other days. For example, an online retail application can keep track of creation of new shopping cart counts, number of buys, number of payments initiated, etc. If there is a sudden change in this counter, it can be first ascertained whether there has been a change in response from the online retail application to other applications which are submitting those transactions. It is possible that clients are resubmitting the transactions due to some error in response being sent back to them. It is a good practice to keep a separate counter for a given request type as well as its corresponding response. For instance, a difference in count values for message submission requests to message submission responses can point to problems. Failure to send responses in a timely manner may be leading those clients to resubmit the messages over and over again. If it is not a response issue, then those applications can be checked to ascertain if they have been experiencing internal errors which are causing them to re-submit those messages. Again, correlating counters can be extremely useful too. As an example, in the online retail application discussed above, usually the number of buys should match with number of payments initiated. If the number of payments initiated counter shows a much higher value, most likely our billing service is erring out or it is even likely that someone is trying to break in to our payment flow.
5. **Count of internal application resources** – Keeping a separate counter for important resources such as database connections, LDAP (Lightweight Directory Access Protocol) connections, threads pools, etc. can be very useful to determine application health. They can also point to leaks in such important resources.
6. **Transaction sizes** – Usually, transaction size is ignored because it remains within some range and the range is not on the order of multiple kilobytes. However, keeping track of transaction size can be useful for media applications which deal with a lot of heavy duty media being relayed to them. During peak holiday seasons, such as New Year or Christmas, there may be many heavy-duty videos being uploaded and transaction sizes may become extremely large (on the order of higher MBs) which may quickly drain system resources or network bandwidth. Another aspect about heavy-duty media is that at some point, the application may need to represent those as an in-memory data

structure (say, as a byte array) and put tremendous strain on heap memory. It is also possible that there can be a strain on system RAM, causing it to start swapping heavily or even end up thrashing. Keeping the size trends recorded historically enables enough capacity planning to architect systems better. For example, we may even take a call to sandbox transactions bigger than certain sizes and handle them with least priority or sandbox them into a dedicated set of machines to handle them, etc.

Applications should also provide an easy way to fetch these values. There are many options available such as Simple Network Management Protocol (SNMP) or HTTP or Java Management Extensions (JMX). A good architecture will make sure that it is built with loose coupling. Data structures which keep track of various counters, time ticks, or values should be maintained separate from the core application architecture and data structures.

Applications degrade gradually unless it is a sudden crash. Using latency metrics for various important flows or error counts, we can build in simple alarm measures in our application. Again, these alarm triggers should fire independent of the application. This decoupling makes sure that the application does not become taxed further and that critical alarms are triggered independent of state our application may be in.

Most applications use heartbeats to monitor application health. A simple script can be put in place which monitors the application heartbeat and upon successive failures or slow responses, triggers a notification to the engineering team with details of some commands such as vmstat, iostat, ps, top, lsof, etc. depending on the application nature. It must also trigger a few thread dumps. If relaying of notification is not possible (not allowed), the script could save these details in a file which can be separately analyzed.

Troubleshooting Checklist

Despite best efforts at building fail-safe applications, there will be instances of production issues. Again, Murphy's Law²⁷ – “Anything that can go wrong, will go wrong” applies well. A few pointers may be helpful in identifying the root cause faster. Usually, it is about asking the right question knowing very well that every science begins with a good question. A very influential book, *Programming Pearls*²⁸, stated something to the same effect¹. Asking the right set of questions can help filter out ‘noise’ and focus on actual root cause(s). Preliminary investigation should help us form a hypothesis which can then be proved or disproved. Here are some questions which can provide initial guidance:

1. When did the problem begin and what were the exact symptoms?

This is important to not only understand circumstances around the issue time but also understand the impact it is causing.

2. Were there any background jobs executing at the same time of issue?

Fire and forget almost always fails when it comes to executing long running jobs. Always keep a document handy of all such background jobs and their scheduled execution times. Many applications have been hit by some database job which got triggered and exhausted database (DB) resources to the extent that upstream application flows started timing out. New software releases can be negatively impacted if such jobs fire during the same time window.

3. Has this issue been noticed before? If yes, when was it and what was the RCA? Does the issue happen at the same time every day?

The basic idea is to see some pattern and use that as starting point.

4. Does the issue really impact business currently? How quickly can we assess data loss?

This is important to decide whether we are investing too much time in troubleshooting issues. We tend to enjoy problem solving and engineering hours are expensive. Calculating data loss is important because we may have SLAs around aborted transactions, dropped messages, failed flows, delayed transactions, etc.

5. How have metrics changed after our new release?

It is a good idea to keep track of various metrics before and after any release. These could be metrics that we discussed in previous sections. For example, if we were keeping track of latencies or error counts before our production release, the same can be checked for changes to ascertain whether our release has caused degradation.

1. Ch5, Pages 55-56 – “The expert debugger never forgets that there has to be a logical explanation, no matter how mysterious the system's behavior may seem when first observed.”

6. If the application has a common dependency which other applications also have (such as a common database), and the problem is with respect to this common application, are all dependent applications facing a similar issue?

Here the idea is to rule out common application as root cause. If the common application is the bottleneck or has faulted, then not only our Java applications should experience Java database connectivity (JDBC) issues, other applications also should.

7. Has our application changed recently? What else has changed besides our application?

Here, the basic idea is to eliminate possibilities. Many times, considerable effort goes into troubleshooting the issue assuming that it is the application which is at fault. However, knowledge of other aspects which may have changed can be of great help. It is likely that the application has had a new release but at the same time some network element has been re-configured. Maybe the Virtual Private Network (VPN) that our clients or partners use to connect to our application has faulted or our load balancer has hit some internal bug.

8. If there are multiple instances of this application on other nodes, are all of them experiencing the same issue?

This can help us not only decide severity of the issue, but also form different hypothesis. If the issue is not present on other nodes, probably something about the entropy on this particular machine needs diagnosis or, this machine has received corrupt data from client, or this node is misconfigured on load balancer, etc.

9. Is there a quick work-around which we can implement as a short term remedy?

It is important to contain the issue quickly especially if it causes data loss. Maybe we can contain the issue if we restart the application every few hours. If we have multiple instances of the same application, then probably a rolling restart of application on each node can prevent complete outage. Sometimes, the remedy may be as simple as turning ON or OFF some application configuration parameters. It is a good practice to document such 'knobs' and keep them handy.

10. Could the problem possibly be affecting even other applications around the globe?

It may sound strange but sometimes we hit a problem which affects applications globally. A subscription to bug or vulnerability notifications can go a long way in preparing for them and saving on precious troubleshooting hours. An example is that of Leap second bug²⁹ which caused high CPU on many Java applications around the world in 2012.

Conclusion

Troubleshooting Java applications on Linux can be tricky due to a whole range of possibilities. We have tried making the subject more within our reach by looking at specifics of how we can leverage both Linux and Java provided tools and commands to get to root cause in a quick manner.

It is important to understand how to correlate Java application behavior with Linux commands output to know on what aspect(s) of our application we need to focus. Importance of application logs cannot be underestimated and serve as one of the most important correlation tools. We have to exercise restraint in terms of commands we execute on a production system which is already under stress knowing very well that these commands and tools can further degrade the application.

Troubleshooting is a reactive measure. It is important to build in enough hooks during the architecture and design phase of our application to help proactive monitoring and alerting. Monitoring and alerting should be non-intruding when it comes to an application we are architecting.

It is always important to identify these metrics upfront and pay attention to how those will provide deeper insights into a misbehaving application. Creating a strategy around these metrics and their acceptable thresholds can be very important in not only avoiding application problems but can also provide invaluable insights into what to improve in application design. These can be our key performance indicators (KPI) which help us honor our SLAs.

Asking the right questions can make a big difference and help create good hypothesis. The ability to ask the right question has nothing to do with application knowledge in many cases and aids the elimination process.

Finally, troubleshooting requires appropriate knowledge of application flows without which correlation with command output can really not be achieved. Troubleshooting requires planning which should ideally begin in the design and architecture phase.

Happy troubleshooting!

References

1. Annual software debugging costs - <http://www.prweb.com/releases/2013/1/prweb10298185.htm>
2. Musings on Math - <http://musingsonmath.com/problems-that-fight-back-2/>
3. The Medical Detectives 1st Edition by Burton Roueche', Penguin Group
4. Linux load averages - <http://www.linuxjournal.com/article/9001>
5. Linux top command - http://linux.about.com/od/commands/l/blcmdl1_top.htm
6. JVM monitoring tools - <http://docs.oracle.com/javase/7/docs/technotes/tools/#monitor>
7. JVM troubleshooting tools - <http://docs.oracle.com/javase/7/docs/technotes/tools/#troubleshoot>
8. Java thread dump analyzer - <https://java.net/projects/tda>
9. Livelock - <http://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html>
10. Garbage Collector Tuning - <http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>
11. jVisualVM Tool - <http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/>
12. Java 1.7 interned Strings - http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6962931
13. Deadlocks - <http://en.wikipedia.org/wiki/Deadlock>
14. Linux System Activity Reporting tool - <http://linuxboxadmin.com/articles/tools-and-utilities/tracking-system-performance-using-sar.html>
15. Wireshark TCP analysis tool - <http://www.wireshark.com/>
16. Nagle's algorithm - <http://tools.ietf.org/search/rfc896>
17. CPU affinity - <http://www.linuxjournal.com/article/6799>
18. Timekeeping in VMs - <http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf>
19. What every programmer must know about memory - <http://ftp.linux.org.ua/pub/docs/developer/general/cpumemory.pdf>
20. Costs of virtualization - <http://queue.acm.org/detail.cfm?id=1348591>
21. Fast user space mutex - <http://en.wikipedia.org/wiki/Futex>
22. Java 1.7 loop unrolling bug - http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7044738
23. Lucene/ SOLR index corruption bug - <https://issues.apache.org/jira/browse/LUCENE-3349>
24. Java Hotspot VM - <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html>
25. Linux kernel profiling tool - <https://perf.wiki.kernel.org/index.php/Tutorial>
26. Numbers every programmer should know - <http://highscalability.com/numbers-everyone-should-know>
27. Murphy's Law - http://en.wikipedia.org/wiki/Murphy's_law
28. Programming Pearls, 2nd edition, Jon Bentley, Pearson Education, Inc.
29. Leap second bug - http://www.theregister.co.uk/2012/07/02/leap_second_crashes_airlines/

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED “AS IS.” EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.