Identifying which Java Thread is consuming most CPU

*I didn’t come up with this. I was shown how to do this by an esteemed college at work.*

Introduction

Most (if not all) productive systems doing anything important will use more than 1 java thread. And when something goes crazy and your cpu usage is on 100%, it is hard to identify which thread(s) is/are causing this. Or so I thought. Until someone smarter than me showed me how it can be done. And here I will show you how to do it and you too can amaze your family and friends with your geek skillz.

A Test Application

In order to test this, we need a test application. So I will give you one. It consists of 3 classes:  
A HeavyThread class that does something CPU intensive (computing [MD5](http://en.wikipedia.org/wiki/MD5) hashes), a LightThread class that does something not-so-cpu-intensive ([counting](http://www.youtube.com/watch?v=f2xjnXvXkak) and sleeping). And a final class to start 1 cpu intensive and several light threads. Here is code for these classes:

import java.security.MessageDigest;  
import java.security.NoSuchAlgorithmException;  
import java.util.UUID;  
  
/\*\*  
 \* thread that does some heavy lifting  
 \*  
 \* @author srasul  
 \*  
 \*/  
public class HeavyThread implements Runnable {  
  
        private long length;  
  
        public HeavyThread(long length) {  
                this.length = length;  
                new Thread(this).start();  
        }  
  
        @Override  
        public void run() {  
                while (true) {  
                        String data = "";  
  
                        // make some shit up  
                        for (int i = 0; i < length; i++) {  
                                data += UUID.randomUUID().toString();  
                        }  
  
                        MessageDigest digest;  
                        try {  
                                digest = MessageDigest.getInstance("MD5");  
                        } catch (NoSuchAlgorithmException e) {  
                                throw new RuntimeException(e);  
                        }  
  
                        // hash that shit  
                        digest.update(data.getBytes());  
                }  
        }  
}

import java.util.Random;  
  
/\*\*  
 \* thread that does little work. just count & sleep  
 \*  
 \* @author srasul  
 \*  
 \*/  
public class LightThread implements Runnable {  
  
        public LightThread() {  
                new Thread(this).start();  
        }  
  
        @Override  
        public void run() {  
                Long l = 0l;  
                while(true) {  
                        l++;  
                        try {  
                                Thread.sleep(new Random().nextInt(10));  
                        } catch (InterruptedException e) {  
                                e.printStackTrace();  
                        }  
                        if(l == Long.MAX\_VALUE) {  
                                l = 0l;  
                        }  
                }  
        }  
}

/\*\*  
 \* start it all  
 \*  
 \* @author srasul  
 \*  
 \*/  
public class StartThreads {  
  
        public static void main(String[] args) {  
                // lets start 1 heavy ...  
                new HeavyThread(1000);  
  
                // ... and 3 light threads  
                new LightThread();  
                new LightThread();  
                new LightThread();  
        }  
}

And Finally...

Assuming that you have never seen this code, and all you have a PID of a runaway java process that is running these classes and is consuming 100% CPU.

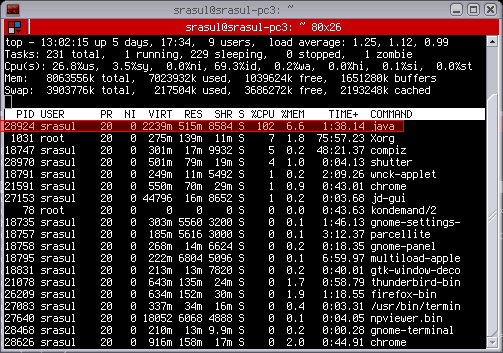
First lets start the StartThreads class.

$ ls

HeavyThread.java LightThread.java StartThreads.java

$ javac \*

$ java StartThreads &

At this stage a java process is running should be taking up 100 cpu. In my top i see:  


In top press Shift-H which turns on Threads. The [man page for top](http://linux.die.net/man/1/top) says:

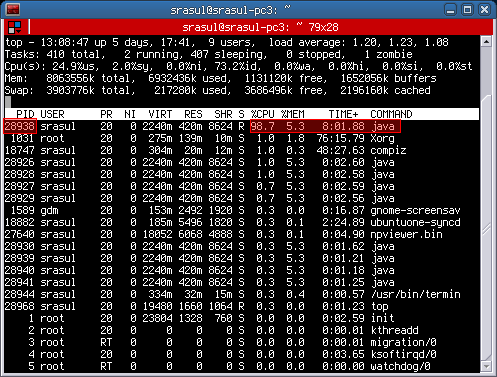
-H : Threads toggle

Starts top with the last remembered 'H' state reversed. When

this toggle is On, all individual threads will be displayed.

Otherwise, top displays a summation of all threads in a

process.

And now in my top with Threads display turned ON i see:  


And I have a java process with PID 28294. Lets get the stack dump of this process using jstack:

$ jstack **28924**

2010-11-18 13:05:41

Full thread dump Java HotSpot(TM) 64-Bit Server VM (17.0-b16 mixed mode):

"Attach Listener" daemon prio=10 tid=0x0000000040ecb000 nid=0x7150 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"DestroyJavaVM" prio=10 tid=0x00007f9a98027800 nid=0x70fd waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Thread-3" prio=10 tid=0x00007f9a98025800 nid=0x710d waiting on condition [0x00007f9a9d543000]

java.lang.Thread.State: TIMED\_WAITING (sleeping)

at java.lang.Thread.sleep(Native Method)

at LightThread.run(LightThread.java:21)

at java.lang.Thread.run(Thread.java:619)

"Thread-2" prio=10 tid=0x00007f9a98023800 nid=0x710c waiting on condition [0x00007f9a9d644000]

java.lang.Thread.State: TIMED\_WAITING (sleeping)

at java.lang.Thread.sleep(Native Method)

at LightThread.run(LightThread.java:21)

at java.lang.Thread.run(Thread.java:619)

"Thread-1" prio=10 tid=0x00007f9a98021800 nid=0x710b waiting on condition [0x00007f9a9d745000]

java.lang.Thread.State: TIMED\_WAITING (sleeping)

at java.lang.Thread.sleep(Native Method)

at LightThread.run(LightThread.java:21)

at java.lang.Thread.run(Thread.java:619)

"Thread-0" prio=10 tid=0x00007f9a98020000 nid=0x710a runnable [0x00007f9a9d846000]

java.lang.Thread.State: RUNNABLE

at sun.security.provider.DigestBase.engineReset(DigestBase.java:139)

at sun.security.provider.DigestBase.engineUpdate(DigestBase.java:104)

at java.security.MessageDigest$Delegate.engineUpdate(MessageDigest.java:538)

at java.security.MessageDigest.update(MessageDigest.java:293)

at sun.security.provider.SecureRandom.engineNextBytes(SecureRandom.java:197)

- locked <0x00007f9aa457e400> (a sun.security.provider.SecureRandom)

at sun.security.provider.NativePRNG$RandomIO.implNextBytes(NativePRNG.java:257)

- locked <0x00007f9aa457e708> (a java.lang.Object)

at sun.security.provider.NativePRNG$RandomIO.access$200(NativePRNG.java:108)

at sun.security.provider.NativePRNG.engineNextBytes(NativePRNG.java:97)

at java.security.SecureRandom.nextBytes(SecureRandom.java:433)

- locked <0x00007f9aa4582fc8> (a java.security.SecureRandom)

at java.util.UUID.randomUUID(UUID.java:162)

at HeavyThread.run(HeavyThread.java:27)

at java.lang.Thread.run(Thread.java:619)

"Low Memory Detector" daemon prio=10 tid=0x00007f9a98006800 nid=0x7108 runnable [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"CompilerThread1" daemon prio=10 tid=0x00007f9a98004000 nid=0x7107 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x00007f9a98001000 nid=0x7106 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x0000000040de4000 nid=0x7105 runnable [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=10 tid=0x0000000040dc4800 nid=0x7104 in Object.wait() [0x00007f9a97ffe000]

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

at java.lang.Object.wait(Native Method)

- waiting on <0x00007f9aa45506b0> (a java.lang.ref.ReferenceQueue$Lock)

at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)

- locked <0x00007f9aa45506b0> (a java.lang.ref.ReferenceQueue$Lock)

at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:134)

at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=10 tid=0x0000000040dbd000 nid=0x7103 in Object.wait() [0x00007f9a9de92000]

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

at java.lang.Object.wait(Native Method)

- waiting on <0x00007f9aa4550318> (a java.lang.ref.Reference$Lock)

at java.lang.Object.wait(Object.java:485)

at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)

- locked <0x00007f9aa4550318> (a java.lang.ref.Reference$Lock)

"VM Thread" prio=10 tid=0x0000000040db8800 nid=0x7102 runnable

"GC task thread#0 (ParallelGC)" prio=10 tid=0x0000000040d6e800 nid=0x70fe runnable

"GC task thread#1 (ParallelGC)" prio=10 tid=0x0000000040d70800 nid=0x70ff runnable

"GC task thread#2 (ParallelGC)" prio=10 tid=0x0000000040d72000 nid=0x7100 runnable

"GC task thread#3 (ParallelGC)" prio=10 tid=0x0000000040d74000 nid=0x7101 runnable

"VM Periodic Task Thread" prio=10 tid=0x00007f9a98011800 nid=0x7109 waiting on condition

JNI global references: 910

From my top I see that the PID of the top thread is: 28938. And 28938 in HEX is 0x710A. Notice that in the stack dump, each thread has an nid which is in HEX. And it just so happens that 0x710A is the id of the thread:

"Thread-0" prio=10 tid=0x00007f9a98020000 **nid=0x710a** runnable [0x00007f9a9d846000]

java.lang.Thread.State: RUNNABLE

at sun.security.provider.DigestBase.engineReset(DigestBase.java:139)

at sun.security.provider.DigestBase.engineUpdate(DigestBase.java:104)

at java.security.MessageDigest$Delegate.engineUpdate(MessageDigest.java:538)

at java.security.MessageDigest.update(MessageDigest.java:293)

at sun.security.provider.SecureRandom.engineNextBytes(SecureRandom.java:197)

- locked <0x00007f9aa457e400> (a sun.security.provider.SecureRandom)

at sun.security.provider.NativePRNG$RandomIO.implNextBytes(NativePRNG.java:257)

- locked <0x00007f9aa457e708> (a java.lang.Object)

at sun.security.provider.NativePRNG$RandomIO.access$200(NativePRNG.java:108)

at sun.security.provider.NativePRNG.engineNextBytes(NativePRNG.java:97)

at java.security.SecureRandom.nextBytes(SecureRandom.java:433)

- locked <0x00007f9aa4582fc8> (a java.security.SecureRandom)

at java.util.UUID.randomUUID(UUID.java:162)

at HeavyThread.run(HeavyThread.java:27)

at java.lang.Thread.run(Thread.java:619)

And so you can confirm that the Thread which is running the HeavyThread class is consuming most cpu.

In read world situations, it will probably be a bunch of threads that consume some portion of CPU and these threads put together will lead to the java process using 100% CPU.

What are the benefits of knowing how garbage collection (GC) works in [Java](http://www.cubrid.org/blog/tags/Java/)? Satisfying the intellectual curiosity as a software engineer would be a valid cause, but also, understanding how GC works can help you write much better Java applications.

This is a very personal and subjective opinion of mine, but I believe that a person well versed in GC tends to be a better Java developer. If you are interested in the GC process, that means you have experience in developing applications of certain size. If you have thought carefully about choosing the right GC algorithm, that means you completely understand the features of the application you have developed. Of course, this may not be common standards for a good developer. However, few would object when I say that understanding GC is a requirement for being a great Java developer.

This is the first of a series of "[*Become a Java GC Expert*](http://www.cubrid.org/blog/tags/Garbage%20Collection/)" articles. I will cover the *GC introduction* this time, and in the next article, I will talk about analyzing GC status and GC tuning examples from [NHN](http://www.cubrid.org/blog/tags/NHN/).

The purpose of this article is to introduce GC to you in an easy way. I hope this article proves to be very helpful. Actually, my colleagues have already published [a few great articles on Java Internals](http://www.cubrid.org/blog/tags/Java/) which became quite popular on Twitter. You may refer to them as well.

Returning back to Garbage Collection, there is a term that you should know before learning about GC. The term is "**stop-the-world**." Stop-the-world will occur no matter which GC algorithm you choose. *Stop-the-world* means that the [JVM](http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/) is stopping the application from running to execute a GC. When stop-the-world occurs, every thread except for the threads needed for the GC will stop their tasks. The interrupted tasks will resume only after the GC task has completed. GC tuning often means reducing this stop-the-world time.

## Generational Garbage Collection

Java does not explicitly specify a memory and remove it in the program code. Some people sets the relevant object to null or use System.gc() method to remove the memory explicitly. Setting it to null is not a big deal, but calling System.gc() method will affect the system performance drastically, and must not be carried out. (Thankfully, I have not yet seen any developer in NHN calling this method.)

In Java, as the developer does not explicitly remove the memory in the program code, the garbage collector finds the unnecessary (garbage) objects and removes them. This garbage collector was created based on the following two hypotheses. (It is more correct to call them suppositions or preconditions, rather than hypotheses.)

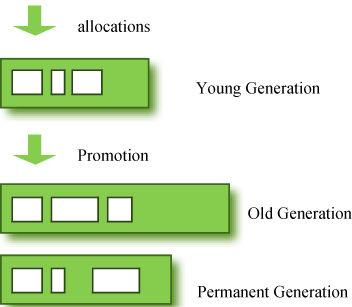
* Most objects soon become unreachable.
* References from old objects to young objects only exist in small numbers.

These hypotheses are called the **weak generational hypothesis**. So in order to preserve the strengths of this hypothesis, it is physically divided into two - **young generation** and **old generation** - in HotSpot VM.

**Young generation**: Most of the newly created objects are located here. Since most objects soon become unreachable, many objects are created in the young generation, then disappear. When objects disappear from this area, we say a "**minor GC**" has occurred.

**Old generation**: The objects that did not become unreachable and survived from the young generation are copied here. It is generally larger than the young generation. As it is bigger in size, the GC occurs less frequently than in the young generation. When objects disappear from the old generation, we say a "**major GC**" (or a "**full GC**") has occurred.

Let's look at this in a chart.



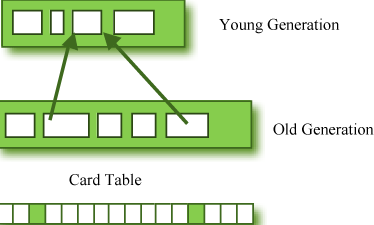
**Figure 1: GC Area & Data Flow.**

The **permanent generation** from the chart above is also called the "**method area,**" and it stores classes or interned character strings. So, this area is definitely not for objects that survived from the old generation to stay permanently. A GC may occur in this area. The GC that took place here is still counted as a major GC.

Some people may wonder:

**What if an object in the old generation need to reference an object in the young generation?**

To handle these cases, there is something called the a "**card table**" in the old generation, which is a *512 byte chunk*. Whenever an object in the old generation references an object in the young generation, it is recorded in this table. When a GC is executed for the young generation, only this card table is searched to determine whether or not it is subject for GC, instead of checking the reference of all the objects in the old generation. This card table is managed with **write barrier**. This *write barrier* is a device that allows a faster performance for minor GC. Though a bit of overhead occurs because of this, the overall GC time is reduced.



**Figure 2: Card Table Structure.**

## Composition of the Young Generation

In order to understand GC, let's learn about the young generation, where the objects are created for the first time. The young generation is divided into 3 spaces.

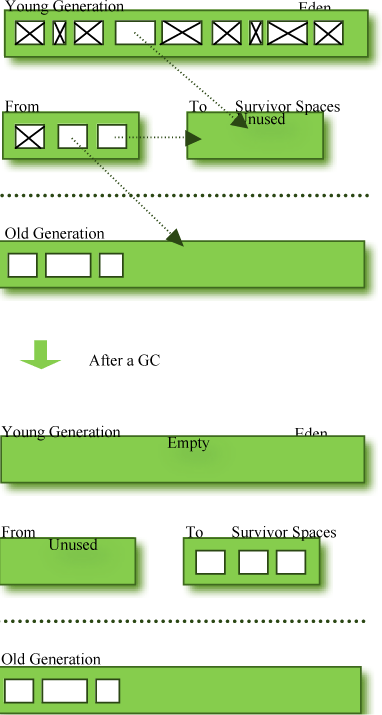
* One **Eden** space
* Two **Survivor** spaces

There are 3 spaces in total, two of which are Survivor spaces. The order of execution process of each space is as below:

1. The majority of newly created objects are located in the Eden space.
2. After one GC in the Eden space, the surviving objects are moved to one of the Survivor spaces.
3. After a GC in the Eden space, the objects are piled up into the Survivor space, where other surviving objects already exist.
4. Once a Survivor space is full, surviving objects are moved to the other Survivor space. Then, the Survivor space that is full will be changed to a state where there is no data at all.
5. The objects that survived these steps that have been repeated a number of times are moved to the old generation.

As you can see by checking these steps, one of the Survivor spaces must remain empty. If *data exists in both Survivor spaces, or the usage is 0 for both spaces*, then take that as a sign that **something is wrong with your system**.

The process of data piling up into the old generation through minor GCs can be shown as in the below chart:



**Figure 3: Before & After a GC.**

Note that in HotSpot VM, two techniques are used for faster memory allocations. One is called "**bump-the-pointer**," and the other is called "**TLABs (Thread-Local Allocation Buffers)**."

**Bump-the-pointer** technique tracks the last object allocated to the Eden space. That object will be located on top of the Eden space. And if there is an object created afterwards, it checks only if the size of the object is suitable for the Eden space. If the said object seems right, it will be placed in the Eden space, and the new object goes on top. So, when new objects are created, only the lastly added object needs to be checked, which allows much faster memory allocations. However, it is a different story if we consider a multithreaded environment. To save objects used by multiple threads in the Eden space for Thread-Safe, an inevitable lock will occur and the performance will drop due to the lock-contention. **TLABs** is the solution to this problem in HotSpot VM. This allows each thread to have a small portion of its Eden space that corresponds to its own share. As each thread can only access to their own TLAB, even the bump-the-pointer technique will allow memory allocations without a lock.

This has been a quick overview of the GC in the young generation. You do not necessarily have to remember the two techniques that I have just mentioned. You will not go to jail for not knowing them. But please remember that after the objects are first created in the Eden space, and the long-surviving objects are moved to the old generation through the Survivor space.

## GC for the Old Generation

The old generation basically performs a GC when the data is full. The execution procedure varies by the GC type, so it would be easier to understand if you know different types of GC.

According to JDK 7, there are 5 GC types.

1. Serial GC
2. Parallel GC
3. Parallel Old GC (Parallel Compacting GC)
4. Concurrent Mark & Sweep GC  (or "CMS")
5. Garbage First (G1) GC

Among these, the **serial GC must not be used on an operating server**. This GC type was created when there was only one CPU core on desktop computers. Using this serial GC will drop the application performance significantly.

Now let's learn about each GC type.

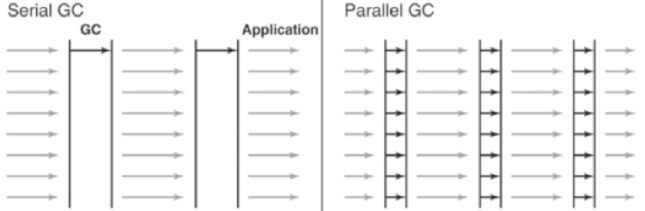
### Serial GC (-XX:+UseSerialGC)

The GC in the young generation uses the type we explained in the previous paragraph. The GC in the old generation uses an algorithm called "**mark-sweep-compact**."

1. The first step of this algorithm is to mark the surviving objects in the old generation.
2. Then, it checks the heap from the front and leaves only the surviving ones behind (sweep).
3. In the last step, it fills up the heap from the front with the objects so that the objects are piled up consecutively, and divides the heap into two parts: one with objects and one without objects (compact).

The serial GC is suitable for a small memory and a small number of CPU cores.

### Parallel GC (-XX:+UseParallelGC)



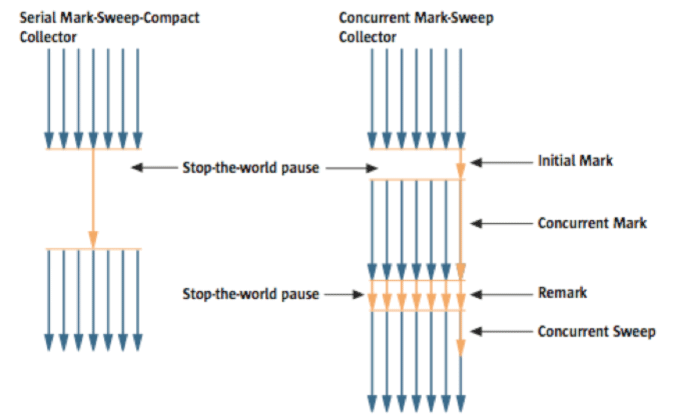
**Figure 4: Difference between the Serial GC and Parallel GC.**

From the picture, you can easily see the difference between the serial GC and parallel GC. While the serial GC uses only one thread to process a GC, the parallel GC uses several threads to process a GC, and therefore, faster. This GC is useful when there is enough memory and a large number of cores. It is also called the "**throughput GC**."

### Parallel Old GC(-XX:+UseParallelOldGC)

Parallel Old GC was supported since JDK 5 update. Compared to the parallel GC, the only difference is the GC algorithm for the old generation. It goes through three steps: *mark – summary – compaction*. The summary step identifies the surviving objects separately for the areas that the GC have previously performed, and thus different from the sweep step of the mark-sweep-compact algorithm. It goes through a little more complicated steps.

### CMS GC (-XX:+UseConcMarkSweepGC)



**Figure 5: Serial GC & CMS GC.**

As you can see from the picture, the Concurrent Mark-Sweep GC is much more complicated than any other GC types that I have explained so far. The early *initial mark* step is simple. The surviving objects among the objects the closest to the classloader are searched. So, the pausing time is very short. In the *concurrent mark* step, the objects referenced by the surviving objects that have just been confirmed are tracked and checked. The difference of this step is that it proceeds while other threads are processed at the same time. In the *remark*step, the objects that were newly added or stopped being referenced in the concurrent mark step are checked. Lastly, in the *concurrent sweep* step, the garbage collection procedure takes place. The garbage collection is carried out while other threads are still being processed. Since this GC type is performed in this manner, the pausing time for GC is very short. The CMS GC is also called the low latency GC, and is **used when the response time from all applications is crucial**.

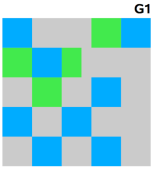
While this GC type has the advantage of short stop-the-world time, it also has the following disadvantages.

* It uses more memory and CPU than other GC types.
* The compaction step is not provided by default.

You need to carefully review before using this type. Also, if the compaction task needs to be carried out because of the many memory fragments, the stop-the-world time can be longer than any other GC types. You need to check how often and how long the compaction task is carried out.

### G1 GC

Finally, let's learn about the garbage first (G1) GC.



**Figure 6: Layout of G1 GC.**

If you want to understand G1 GC, forget everything you know about the young generation and the old generation. As you can see in the picture, one object is allocated to each grid, and then a GC is executed. Then, once one area is full, the objects are allocated to another area, and then a GC is executed. The steps where the data moves from the three spaces of the young generation to the old generation cannot be found in this GC type. This type was created to replace the CMS GC, which has causes a lot of issues and complaints in the long term.

The biggest advantage of the G1 GC is its **performance**. It is faster than any other GC types that we have discussed so far. But in JDK 6, this is called an *early access* and can be used only for a test. It is officially included in JDK 7. In my personal opinion, we need to go through a long test period (at least 1 year) before NHN can use JDK7 in actual services, so you probably should wait a while. Also, I heard a few times that a JVM crash occurred after applying the G1 in JDK 6. Please wait until it is more stable.

I will talk about the **GC tuning** in the next issue, but I would like to ask you one thing in advance. If the size and the type of all objects created in the application are identical, all the GC options for WAS used in our company can be the same. But the size and the lifespan of the objects created by WAS vary depending on the service, and the type of equipment varies as well. In other words, just because a certain service uses the GC option "A," it does not mean that the same option will bring the best results for a different service. It is necessary to find the best values for the WAS threads, WAS instances for each equipment and each GC option by constant tuning and monitoring. This did not come from my personal experience, but from the discussion of the engineers making Oracle JVM for JavaOne 2010.

In this issue, we have only glanced at the GC for Java. Please look forward to our next issue, where I will talk about **how to monitor the Java GC status and tune GC**.

I would like to note that I referred to a new book released in December 2011 called "*Java Performance*" ([Amazon](http://amzn.com/0137142528), it can also be viewed from safari online, if the company provides an account), as well as “*Memory Management in the Java HotSpotTM Virtual Machine*,” a white paper provided by the Oracle website. (The book is different from "*Java Performance Tuning*.")

By Sangmin Lee, Senior Engineer at Performance Engineering Lab, NHN Corporation.

# **Top 10 reasons why your Enterprise Java Application is slow**

*by* KARUN SUBRAMANIAN *on* JULY 27, 2014

Let’s face it. **Nobody likes a slow application**. It is our job to make sure our applications  fulfill the functionality need completely, but at the same time delivers good user experience.

In 16 years of my experience in technical support primarily in the **world of JEE**, while I cannot honestly say ‘I have seen it all’, I can say “I have seen a **lot**“. Here are the top 10 reasons why your Java application may slow.

Note that this list is NOT ordered in anyway, meaning reason 1 is not always the most popular reason for the slowness. These are all equally critical issues that can cause your application to crawl.

**1. There is a sudden spike in the ‘Load’ that you did not notice**

This is especially true when your application processes requests from external entities (for example, a financial institution processing batch updates from a data provider). Since you cannot really control (for the most part) what others might send, you can suffer a surge in incoming requests and hence your application can slow down.

**How can you monitor?**

You must monitor the **‘throughput’** of the application (specifically, number of requests/minute). You will most probably need a commercial **APM (Application Performance Management)** tool to monitor this reliably. If you don’t have an APM solution implemented (why ?), try to come up with some home grown script/program that can monitor this. **For one or my clients**, I had to write a program that performed a ‘row count’ of a Database table that stored the ‘processed orders’. I had to run this program through a scheduler (cron) periodically and send an alert email if the number of processed records crossed a threshold.

**2. There is a poorly responding backend system**

Your application may talk to one or more backend systems other than a Database. Perhaps**it talks to an external system using a Web Service call**. Perhaps it sends a WebSphere MQ message to a**remote Queue Manager** (that in turn routes the message to an external system to get a response). If any of these external systems respond poorly (or even stop completely), your application will suffer.

**How can you monitor?**

You must monitor the **response time of the external systems experienced by your application.** **A periodic synthetic transaction might help** (if feasible). If you log the response times in a log file, you could monitor the particular entry in the log file. But the best bet would be to use an APM solution and instrument the specific method that makes the external call. Tools such as **AppDynamics and CA Wily Introscope** (among many others) can automatically detect the backends and reveal the response times.

**3. Your database calls are taking for ever**

This is a big one. It is possible that the sql queries (or Stored procedures) that originate from your application run extremely slow at the Database Server. This could be due to the following reasons:

a. Database Server is **running our System resources** such as CPU, Memory etc,

b. A **Block** in the Database

c. **Missing index in the database,**

d. Your query really runs slow (perhaps you are**missing a ‘where’ clause**). In this case, work with a DBA to tune the query/SP.

**How can you monitor ?**

To some extent, monitoring from the DB side should help i.e the DBAs should have monitoring for things like long running queries, blocks, deadlocks etc. From application’s perspective, you must monitor the **JDBC response times experienced by your application**. You can either log this information in log file (expensive and not recommended) or use an APM tool. New generation APM tools like **Dynatrace** (among many others) can reveal slow running JDBC calls including the **actual SQL query or stored procedure**.

**4. You are running out Database connections**

This is another big one. Typically 50 JDBC connections per JVM should be more than enough. But it greatly depends on the Application. They key is to make sure you are not maxing out of DB connections. When you **max out the DB Connection Pool,** your requests are going to **wait** for an available connection which results in painful slow response time. The reason why you are maxing out of the DB connections should be found out. It could be that you are NOT closing the DB connections properly.

**How can you monitor?**

If your application server has **JMX console**, Connection pools can be monitored via the MBeans, but it can be a painful process. You can also write your own Java program (**JMX client**) to connect and retrieve values from the Mbeans. Easy way is to install a commercial APM tool

[](http://karunsubramanian.com/wp-content/uploads/2014/07/jmx.jpg)

Image from wiki

**5. You are running out of Threads**

Work enters to your application via a Thread. You only have limited number of Threads to process incoming requests (limited by the **Thread Pool**). When you exhaust all the available threads in the Thread pool, subsequent requests go into ‘wait’ state, waiting for an available thread – **painful wait time**. The reason why you are running of out thread should be investigated. It could be because of a poorly performing backend or a slow DB response time.

**How can you monitor?**

If your application server has **JMX console**, Thread pools can be monitored via the MBeans, but it can be a painful process. You can also write your own Java program (JMX client) to connect and retrieve values from the Mbeans. Easy way is to install a commercial APM tool. Some Application Server consoles (such **as IBM Websphere application server**) have built in monitoring dashboards as part of the Admin console. But it can be extremely slow and not very flexible.

**6. The Server hosting your application is running slow or running out of resources**

Yes, even with Cloud and dynamically expandable hardware, the fact is hardware is not unlimited. When the CPU hits 100% in your host Server, your application is going to suffer. With shared hardware in most enterprises, it is possible that some other application is chewing up all the CPU and your application just becomes a victim.

**How can you Monitor?**

Operating system level monitoring tools will help. Tools like **Nagios** can help monitor the infrastructure effectively.

**7. Super excessive logging in your Application**

Logging is expensive, in terms of resource usage. While debug logging will help in certain situations, in production environments, keep the logging to a minimum level (perhaps ‘INFO’ or ‘WARN’ log4j levels). An additional side effect of excessive logging is Disk drives filling up (which can have its own consequences). One of my clients was logging every single sql statement generated by the application (Hibernate verbose logging) and the application was generating 500 MB log file every day. While the performance impact was not too bad in this case, it created disk space issue until I disabled the verbose logging.

**How can you monitor ?**

**Periodically check your Application log files** (JVM system out and system error). If you notice unnecessary noise, get rid of those logging entries. Checkout **logstash**.

**8. Garbage collection overhead**

This is a critical performance area for any Java application. If you are running of Java Heap, the JVM will initiate GC to keep up with the memory demand. GC tuning is a separate and exhaustive topic by itself. Short story is GC will consume resources and excessive GC overhead will slow down your application. Pay attention to the Minimum (-Xms) and Maximum (-Xmx) Heap size values in the Java command line options.

**How can you monitor?**

Enable verbose GC logging and watch the log file. You enable verbose GC logging with the options -verbose:gc -XX:+PrintGCDetails XX:+PrintGCTimeStamps.

Once the logs are collected, you can either eyeball it or use a tool like **“IBM Pattern Modeling and Analysis Tool for Java Garbage Collector”**

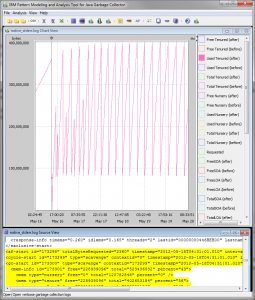
[](http://karunsubramanian.com/wp-content/uploads/2014/07/gc.png)

Image from ibm.com

**9. Third party jar files that you have included in your application is buggy**

It is not uncommon to use third party jar files to use certain functionality for your application. However care must be taken to ensure there are no performance implications. Even if the jar is reliable and proven (such as spring.jar), there may be issues when you upgrade the spring version without analyzing the release notes etc. Perhaps you may be using the API in a certain way that changed with the newer release of spring. I have seen third party jars causing severe memory leak. I have also seen a rules engine (that was integrated into my client’s JEE Application) that ran into ‘infinite loop’ issue causing the entire application to slow down.

**How can you monitor?**

This can be trick to monitor. There is no bullet proof way to monitor just one jar file/plugin unless your application specifically uses the thrid part jar for certain specific transactions, in which case you can use an APM tool to instrument montioring for just that particular method. Even then, you won’t know if it is your code or the third part jar that is creating the issue. Through analysis is required to troubleshoot such issue.

**10. Poor Application Architecture**

No matter how much tuning you put in your code, application server and database servers, if the Architecture is flawed, your application will be slow. For example,

a. Even when not required, using ‘persistent’ messages with JMS.

b. Storing more than necessary data in a HTTP  ‘Session’ increasing its size

c. Not using (or using misconfigured) caching solution

Application architecture is a vast subject and scores of books have been written on this subject. Study carefully.

**How can you monitor?**

There is no specific monitoring to be done to monitor design flaws. Design issues will be revealed when you thoroughly analyze the problem and the available monitoring data.

There you have it. Application performance tuning starts from the design stage. There are various reasons your application could be slow. You must be diligent to monitor as much relevant metrics you can about your application and act promptly to avoid customer impact. Investing in a good quality APM solution adds heaps (no pun intended) of ammunition to your weapons in the battle of performance monitoring.

# **4 things you need to know about CPU utilization of your Java application**

So, when was the last time you monitored the CPU utilized by your Java application ? Perhaps ‘never’ ?. It is actually important to keep an eye on the CPU utilzied by your application. Your Application runs on a host (or hosts) that has one or more CPUs managed by the Operating System. **The CPU resource is not unlimited** (even though some developers and Administrators wish that were true).

Let’s dive into the 4 things you need to know about the CPU Utilization of your Java application.

**1. Why do you care ?**

The CPU cycles are shared among all the processes running on the host. When there is no cpu cycles left, **things start to queue up** and you see ‘slowness’ in your Application.**It is possible that your application is the one that consumes the most CPU.**Or it could be some other process that is taking up all the CPU, but your application will be affected regardless because all the processes on the host share the same CPU resources.

Based on my experience, your application should take no more than **30% of the CPU**, if it is the only ‘major’ application on the host. This is to make sure you have some head room in case of sudden spike in the Load. When you notice higher CPU usage, **it can mean underlying issues with the code and/or the environment.**

Top reasons for higher CPU Utilization:

a. There are excessive GC cycles going on

b. Too many Application threads active

c. Code problems such as ‘infinite loops’ or excessive backend calls.

d. The application is indeed working hard and the host does not have enough CPU (this is actually a good reason)

**2. How do I measure the CPU utilization of my Java Application ?**

Your application is nothing more than a ‘process’ as far as the operating System is concerned. This means you can rely on **Operating System tools** to find out the CPU utilized by your application.

It is important to note that you must be able to identify your Application process in order to see the CPU utilized by your application. To do this:

Unix/Linux:

Use ‘ps -ef | grep <a string that can identify your application.>’. For example, ‘ ps -ef | grep java’.

You can use couple of commands to check the CPU utilization in Unix/Linux world

a. Using the command **‘top’**

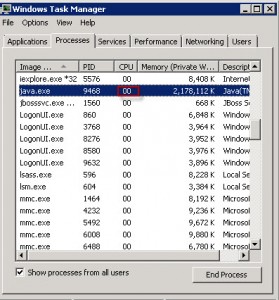
‘top’ is a very powerful tool to show the process level statistics including CPU usage.

b. Using the command **‘ps -eo pid,pcpu,args | grep <a string that can identify your application>’**

The second column in the output of the above command shows the percentage CPU used.

Windows:

Use **‘Task Manager’** to view the CPU Utilized by your application process.

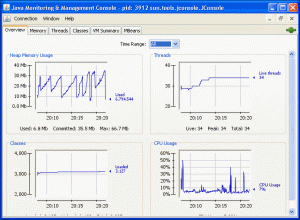
[](http://karunsubramanian.com/wp-content/uploads/2014/07/Task-Manager.jpg)

Note about Clusters:

If you have more than one instance of your application running on the same host (Vertical cluster), you must add the CPU utilized by all the instances to find out the net CPU Utilization of your application.

Other tools:

You can use **VisualVM** or **Jconsole** to monitor CPU utilized by your application.

[](http://karunsubramanian.com/wp-content/uploads/2014/07/jconsole.gif)

*Image from docs.oracle.com*

APM (Application Performance tools) can be of great help here. They not only monitor the CPU usage, they can **create alerts based on a preset threshold or automatically learned threshold**. They can also store the metrics for historical analysis.

**3. The sweet spot**

**So, it is really bad if your application consumes lot of CPU** ? Not necessarily. It is possible that your application indeed needs more CPU because of the type of processing it does. **When the only bottleneck is the CPU resource for your application, you have achieved the sweet spot.** This means you can increase the throughput just by increasing the amount of CPU (either by adding more CPUs to the same host or by adding a new cluster member)

Also note that some operating systems allow you to **setup a quota for CPU utilization**. For example you can assign maximum of **40% CPU to certain process**. This will help to avoid bringing down the entire server due to a bad application. Check with your Systems Administrator if such quota has been setup.

**4. Effect of Number of threads**

More number of active threads within the application, heavier the CPU usage will be. There will be**numerous context switches** because of excessive number of threads. In fact the role of  **Thread Pool**is to limit the number threads from going over a limit. Excessive number of threads can indicate a code issue or a poorly responding backend.

You can take a thread dump to review the threads inside the JVM.

Use ‘kill -3 < PID >’ in Unix/Linux to generate Thread dump. Use ‘jstack’ command in Windows to generate the thread dump.

There you have it. 4 things you need to know about CPU utilization of your java application. **Get into the habit of periodically monitoring the CPU utilized by your application.** Excessive CPU utilization can indicate an **underlying code issue**. It can also affect other applications running on the same host. The best way to monitor CPU utilization is to **invest in a good quality APM** (Application Performance Management) tool. The best way to defend CPU usage issues is to thoroughly **load test your application** in lower environments and resolve issues before deploying to Production.

Good luck.

# **5 not so easy ways to monitor the Heap Usage of your Java Application**

Are you on top of monitoring one of the most important metrics in your Java application ? I’m talking about Java Heap usage. If you think you don’t have to monitor the heap usage, think again.

“OutOfMemory” error is one of the deadliest and not keeping track of your heap usage means you are walking on a land mine – It can blow up all of a sudden.

Now that you know the importance of monitoring the Heap usage, how do you proceed ?

First of all, I would like to clarify one important thing.

The Memory utilization by the ‘process’ shown by operating system commands such as top (unix) or Task Manager (Windows) **is NOT the java heap usage**. What you see from Operating System perspective is the ‘process’ memory which **‘includes’** heap. We are not going to discuss about this metric.

What we are interested is the Heap size, which is limited by -Xmx command line option of the ‘java’ command. For example,

**java -Xmx1024m**

this means the java application can have maximum of 1024 Mega Bytes of Heap.

If you don’t know the maximum heap size for your application, check the command line of your java application to locate the -Xmx option.

There are quite a few ways you can capture the heap usage. Here are the 5 not so easy ways

**1. Use Jconsole**

Jconsole is a GUI that can be used to monitor performance metrics of a java application. It attaches to a running Java application either locally or remotely (i.e. you can monitor a Java application that is running on the same system as Jconsole, or running on a remote system).

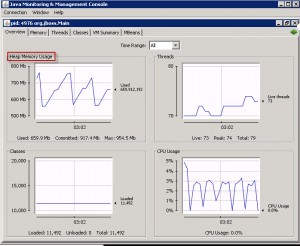
The command Jconsole is found in JDK\_HOME/bin (this means you must have JDK installed, not just JRE (a typical scenario in production environments).

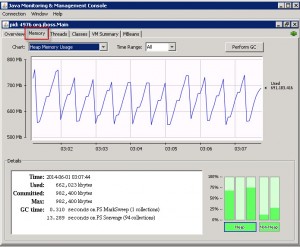
When running locally, all you need is the process id of the java application.

To invoke jconsole, just execute the following command. Example shown below is for Windows.

***JDK\_HOME/bin/jconsole.exe <PROCESS ID OF YOUR JAVA APPLICATION>***

Important Note: You must have the correct access to the application in order for jconsole to connect to the running jvm. Typically, you must be the user who is running the java application as well as the jconsole command.

[](http://karunsubramanian.com/wp-content/uploads/2014/06/jconsole.jpg)

[](http://karunsubramanian.com/wp-content/uploads/2014/06/jconsole1.jpg)

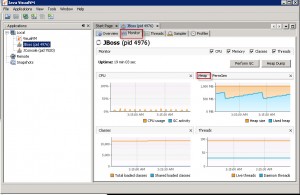
**2. Use VisualVM**

VisualVM is a relatively new tool that not only shows the performance metrics form the JVM but can be used to profile the application too.

Start visualvm by running the command

***JDK\_HOME/bin/jvisualvm.exe***

Visual VM automatically finds the available java applications and shows them on the left pane. Either double click or ‘right click and select open’. You can see tons of useful information now

[](http://karunsubramanian.com/wp-content/uploads/2014/06/visualvm.jpg)

**3. Use Jstat command**

jstat command comes with JDK as well. Invoke the command as follows

***JDK\_HOME/bin/jstat.exe -gc <PROCESS ID OF YOUR JAVA APPLICATION>***

Sample output:

*S0C S1C S0U S1U EC EU OC OU PC PU YGC YGCT FGC FGCT GCT*  
*22400 20736 224 0 306368 202917.9 699072 528814.9 74624 74552.1 112 13.496 1 0.311 13.807*

Here is the description of the fields.

S0C  Current survivor space 0 capacity (KB).  
S1C  Current survivor space 1 capacity (KB).  
S0U  Survivor space 0 utilization (KB).  
S1U  Survivor space 1 utilization (KB).  
EC  Current eden space capacity (KB).  
EU  Eden space utilization (KB).  
OC  Current old space capacity (KB).  
OU  Old space utilization (KB).  
PC  Current permanent space capacity (KB).  
PU  Permanent space utilization (KB).  
YGC  Number of young generation GC Events.  
YGCT  Young generation garbage collection time.  
FGC  Number of full GC events.  
FGCT  Full garbage collection time.  
GCT  Total garbage collection time.

When you add all the ‘utilizations’ i.e OU,PU,EU,S0U,S1U, you get the total Heap utilization.

**4. Use -verbose:gc command line option**

Add **-verbose:gc** command line option to the Java command and you will get detailed GC stats which will include Generational heap sizes before and after GC. You can also add the following options for Oracle HotSpot VM to get further information

**-XX:+PrintGCDetails and -XX:+PrintGCTimeStamps**

By analyzing the verbose GC output, you can determine the Heap usage. One small issue with this method is that you will get to see the metrics ONLY when a GC occurs (since a log entry is added ONLY when a GC occurs).

**5. Use the JEE Application Server facilities.**

If your application is a JEE application deployed on an application Server (such as IBM WebSphere or Oracle WebLogic), use the Admin console or JMX console (if available) to find out the heap usage. Method varies for each Application server. For example, with IBM WebSphere, you can use the Tivoli Performance Viewer that comes with the Admin Console to view performance metrics including the Heap usage. With JBoss, you can use the JMX console to grab the values for the MBean**jboss.system:type=ServerInfo**

I guess by now you know that all the above mentioned methods require some sort of effort to pull the information (trust me, when you try to have jconsole or visualvm connect to a remote Application, you would be thinking *‘why in the world this has to be this difficult ?’)*

**So, What is the easy way to monitor Heap usage ?**

**APM**

The easy way to monitor Heap usage is by using a commercial APM (Application Performance management tool) such as CA Wily APM, AppDynamics,New Relic,Riverbed etc.

APM tools not only monitor the heap usage, you can configure the tool to **Alert you when Heap usage is not normal**. In this way, **you can catch a production problem before it occurs.**Better yet, APM tools have **advanced capabilities to track the number of objects** (a big aid in troubleshooting**memory leak**).

Once you  have an APM solution deployed, you don’t ever have to worry about keeping track of the heap usage constantly.

Good luck

# **Troubleshooting GC: Step by Step instructions to analyze verbose GC logs**

So you enabled verbose GC logs and obtained the logs. Congratulations. You have done one of the most important steps in troubleshooting JVM memory issues. But merely having the log does not do any good. You need to analyze and determine if there is a smoking gun. Here is how to do it.

1. First make sure you collect the logs for at least few hours. Longer the better.

2. Open the log file in a text editor.

3. You are primarily looking for two things

**a. Is the GC happening very frequently?**

**b. Is each GC cycle taking long time?**

Is the GC happening very frequently:

In a healthy application GC cycles should be spaced out – at least by few minutes (especially the Majorcollection aka full gc). If you see GC kicking in ever few seconds, you may have an issue.

Is the GC cycle taking log time:

Ideally minor GCs should take few milliseconds and Full GCs can take about a second (even this is not acceptable in real time transactions environments)

Remember: Even with Concurrent Mark/Sweep collector, there is ‘JVM Pausing’ involved – meaning the entire JVM is paused. This can result in significant user experience degradation.

Let us look at an excerpt from an actual verbose GC log – Note that the exact format varies by JVM vendor and by version.

*18015.9631: [GC [PSYoungGen2: 399742K3->800K4(641984K5)] 818698K6–*

*>420282K7(2040128K8), 0.01727179 secs] [Times: user=0.0210 sys=0.0011,*

*real=0.02 secs12]*

*18015.981: [Full GC13 (System) [PSYoungGen: 800K->0K(641984K)]*

*[PSOldGen: 419482K->420201K(1398144K)] 420282K->420201K(2040128K)*

*[PSPermGen: 128361K->128361K(262144K)], 1.2065614 secs] [Times:*

*user=1.20 sys=0.00, real=1.20 secs]*

1: Time elapsed since the start of the application, in seconds. Useful to find out how often the GC runs

2: Indicates it is a YoungGen Collection (Minor collection)

3: Total Size of live objects before collection, in Kilo Bytes

4: Total Size of live objects after collection, in Kilo Bytes

5: Committed size of the heap in Young generation, in Kilo Bytes (i.e) the amount of heap available for

future allocation requests without expanding the heap

6: Total Size of the heap before collection, in Kilo Bytes

7. Total Size of the heap after collection, in Kilo Bytes

8: Total Committed size of heap (the amount of heap available for future allocation requests without

expanding the heap)

9: Time taken to perform this GC cycle, in seconds

10,11,12 : Break up of time spent in user,sys and wallclock time

13: Indicates it is a Full GC (Major collection)

Notice couple of things in the Full GC cycle

Hardly any space was reclaimed from Permgen. This is because permgen is used to hold class objects (metadata). Some third party applications (like javaagents) are known to consume lot of permgen space.

The time taken by Full GC is much more than Minor GC (1.2 seconds compared to 0.01 seconds). Imagine yours is a stock trading application. Pausing the application for couple of seconds for Full GC Is a huge deal indeed.

4. From the information above, determine if GC is occurring too frequently and/or taking lot of time for each cycle. From this, you can decided on what to do next (perhaps you need to increase the YoungGen size, perhaps the Max Heap size , perhaps pergmen size or perhaps all or none of the above). You see how useful verbose GC logs can be?

To update Permgen:

-XX:MaxPermSize

To update Max heap:

-Xmx

To update New Gen:

-XX:MaxNewSize.

Also see -XXNewRatio

5. Once you update any of the GC parameters mentioned above, collect verbose GC logs for few more hours and compare. You may have to do this few times to arrive at a sweet spot.

Note: Make sure you save the old gc logs whenever you restart the application

Note: Some JDK vendors like IBM provide much more detailed GC logs than the one shown above. So, make use of it.

Now that you know how to use verbose GC logs, let’s look at an important built-in tool that can be useful in troubleshooting JVM memory issues – **jconsole**

# **Troubleshooting using Java Thread Dumps – Part 1**

‘Threads’ is perhaps the **most misunderstood** component among Application support engineers and even among Software Developers. Take a look at the following questions that we typically come across in our glorious support career:

1. What does it mean to say ‘**There must be hung Threads’** in the Application?
2. Have you **exhausted** the available Threads in Application Server?
3. **Which Thread Pool** is used for Web Container?
4. Do I have to configure the **number of JDBC Connections** in the connection pool equal to the number of Threads I have in the Application Server?
5. How can I find out which thread is **currently running** in the Application Server?
6. Is ‘number of Threads’ **configurable**? Can I have unlimited Threads?
7. How can I find out all Threads that are currently **running**?
8. Is there a **Thread Dead Lock** going on?
9. How can I **kill** hung Threads?

Well, after going through this three part series, you will be able to answer all the above questions.

**What is a Thread?**

Let us clarify one thing first:

The name ‘Thread’ actually can appear in two contexts when we are dealing with JEE Application Servers.

1. Context 1: Thread as in ‘A thread explicitly created by Application Developer from the code (typically by extending the ‘Thread’ class or by implementing ‘Runnable’ interface).
2. Context 2: The Application Server Threads provided by the Application Server which is configurable via Application Server Admin utilities (mostly Admin Console)

In this lecture, whenever we say Thread, we will be talking about Context 2 (Application Server Threads). So when you are talking to a developer, be clear about this.

Let us define the Thread:

**Work enters Application Server via Threads**.  Everything the Application Server does is via Threads. For example, when the Application Server first starts, it spawns a Thread to load the Web Application.

Application Server can spawn multiple Threads to execute things simultaneously. For example, when a user is actively performing some work in the application, it usually occupies a Thread. Now, if a new user logs in, the Application Server can spawn a new thread and execute work for the new user using the new Thread.

Geek Note

A thread is also known as **LWP** – **Light Weight Process**, It is the basic unit of execution.

A Thread is created by a Process (Heavy weight Process). A process can create multiple Threads (Multi-Threaded Process) or just one Thread (Single Threaded Process). A process at least has one thread which is the PRIMARY Thread.

As you may have guessed, the JVM is a multi-Threaded Process.

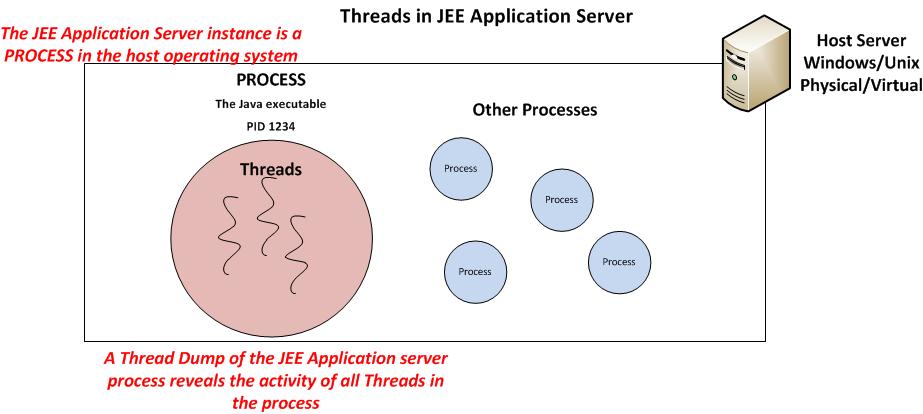
Each Thread has its own stack.

All the Threads within a process share the global resources (such as Memory, Operating Systems Resources etc)

**Threads in a Java EE Application Server**

Remember: The Java EE Application Server (Weblogic, WebSphere etc) are n**othing but ‘processes’ as far as Operating System is concerne**. Each Application Server instance is a ‘java’ program running in the Operating System. This Application Server Java program is a multi-Threaded program. What is the purpose of this program? It loads and runs the JEE Application written by the developers. Some aspects of the Threads in JEE Application Server (for example: number of Threads) is configurable via the Admin tools provided by the Vendor (ex: Admin Console, wsadmin scripts etc).

Thread Management of one Application Server can differ from another (for example, Jboss and Weblogic)

[](http://karunsubramanian.com/wp-content/uploads/2014/10/Threads.jpg)

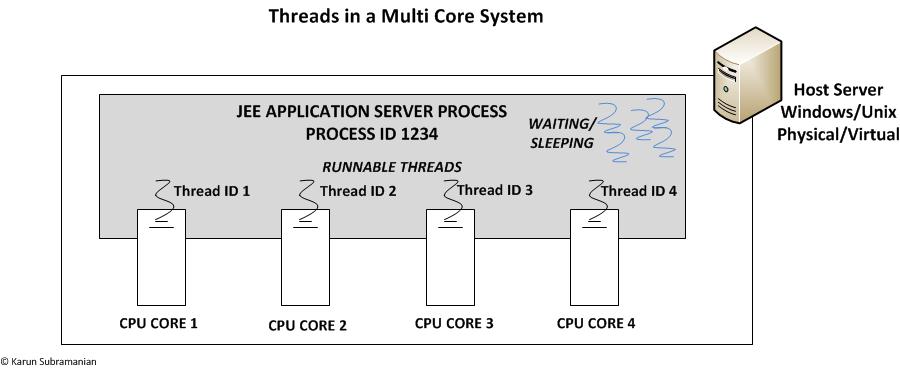
**What is a relationship between the Application Server Threads and the Number of CPUs?**

The Server can execute as many Threads simultaneously as the number of CPU cores**. Number of cores** you have in your server is one of the major factors in determining the number of Threads. **But it is not the ONLY factor.**

**Generally, you should not have to tweak the number of Threads** as newer versions of JEE Application servers automatically manage this. Use the following method to determine if you have the optimal number of Threads

1. Load the application using load testing tools like Jmeter or LoadRunner (or somehow mimic a peak load on the application)
2. Measure the CPU utilization of the Server (using Operating system tools is fine. For example Windows Task Manager or Unix ‘top’command)
3. If the CPU is mostly idle, you could technically increase the number of Threads
4. If the CPU is heavily utilized, you may want to reduce the number of Threads.

**Note**: Pay attention to the Memory usage as well as Threads consume memory. The above method is only approximate. Multiple iterations of load testing is required if you need to arrive at the best number.

[](http://karunsubramanian.com/wp-content/uploads/2014/10/Threads-multicore.jpg)

**When do you need to take a Thread dump?**

**The Thread dump will reveal the activity of all the threads within the JVM process at the time of the Thread dump**. It is like an *X-Ray* for the Application Server. You typically need to take a Thread Dump for following types of issues.

1. Hung Applications (Applications that do not respond)
2. Extremely poor response times
3. Very slow startup
4. Unusual amount of resource utilization (for example, 100% CPU usage most of the time)
5. Any time you want to see what the Application Server is doing

What you have learnt is a great start for using Thread dumps for troubleshooting. In part2, we will look at the following:

**Does taking Thread dump kill the Application Server? Are there any side effects?**

**How to take a thread dump in various Application Servers?**

**What is the most imporant piece of information you should look for in a Thread Dump ?**

Stay tuned and keep up the good work of keeping the lights on.

[**nitor Everything**](http://karunsubramanian.com/)

Resources for Application Performance Management and Monitoring

# **Troubleshooting using Java Thread Dumps – Part 2**

*by* KARUN SUBRAMANIAN *on* OCTOBER 26, 2014

Okay. Now that you have some fundamentals drilled in, let us look at how exactly to take the Thread Dump. But even before that, let us answer this question:

**Does taking Thread dump kill the Application Server? Are there any side effects?**

When you take the Thread dump, the JVM is momentarily paused to dump the stack. This does mean that the JVM is interrupted. But that’s about the effect taking the Thread dump should have on the JVM – a momentary pause. However since it is somewhat intrusive process, there is an extremely rare possibility that it does something adverse to the JVM, like getting aborted.  But in practice the cases like this are virtually none.

**How do you take a Thread Dump of a JEE Application Server?**

The procedure can vary greatly between Application Server Vendors and the Operating System in which the Application Server runs. Also, there can be more than one way to get the Thread Dump depends upon the Application Server.

**Jboss**

Windows

jstack –l <pid> > javacore.txt

<pid> is the Process id of the java process.

Jstack is a command that comes with JDK (6 and above)

Javacore.txt is the resulting Thread dump

Note: You may have to use psexec to run the jstack if you don’t have appropriate permissions.

Psexec –s jstack –l <pid> > javacore.txt

Psexec can be downloaded from Winsows Sysinternals Website. This somewhat similar to doing sudo in Unix.

Unix

jstack –l <pid> > javacore.txt

Or

kill -3 < pid>

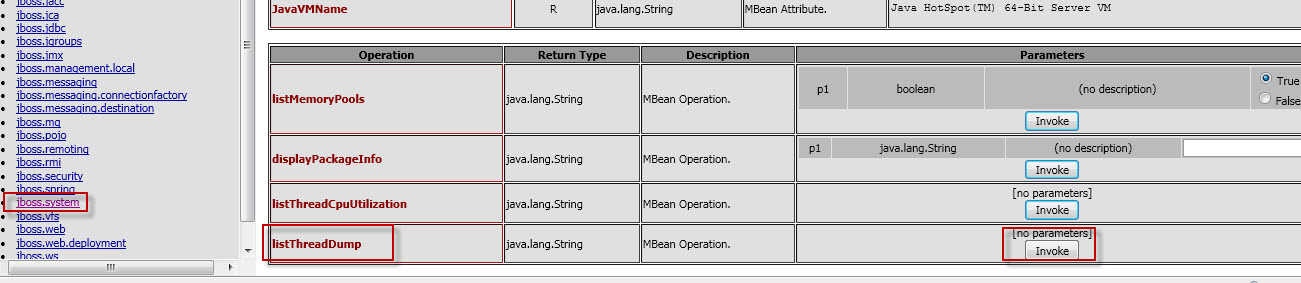
<pid> is the Process id of the java process.

Jstack is a command that comes with JDK (6 and above)

Javacore.txt is the resulting Thread dump

With kill -3, the thread dump may go to the server.log

Note: You can also use the JMX Console to trigger a Thread dump. Navigate to ‘jboss.system type->ServerInfo’. Scroll down and click the ‘invoke’ button for the operation ‘listThreadDump’

[](http://karunsubramanian.com/wp-content/uploads/2014/10/jmx1.jpg)

**WebSphere:**

The jstack for Windows and ‘kill -3’ for Unix mentioned above in the ‘Jboss’ section will work for WebSphere as well.

In addition, you can use ‘wsadmin’ to trigger a Thread dump

<WSHOME>/wsadmin.sh -lang jython

wsadmin>serverProcess=AdminControl.completeObjectName(‘type=JVM,process=myServer,\*’)

Replace ‘myServer’ with the Application Server name

wsadmin>AdminControl.invoke(server,’dumpThreads’)

The Thread dump will usually go to the Profile Home with a name such as javacore.date.time.processid.txt

**WebLogic**

The jstack for Windows and ‘kill -3’ for Unix mentioned above in the ‘Jboss’ section will work for WeLogic as well.

In addition there are several ways you can obtain Thread dump of a WebLogic Application Server. One popular way is to use WLST (Weblogic scripting tool)

Set the environment using setDomain.sh or setDomain.cmd

java weblogic.WLST connect(“<username>”,”<password>”,”t3://<url>:<port>”) threadDump()

The thread dump will be generated in Thread\_Dump\_<Servername>.txt

**Note:**If you are running Jrockit JVM, there is yet another easy way to take a thread dump. Just run

jrcmd <pid> print\_threads

**How many thread dumps do you need to take?**

Very important question. **You must always take at least three Thread dumps with 10 to 20** **seconds interval**. Why? This is how we can verify if a Thread is really stuck, by identifying the same Thread in all the three Thread Dumps executing the same line of code. Make sure you create Three different Thread dumps i.e. don’t overwrite the same Thread dump.

**What do you need to look for in a Thread dump?**

You got the thread dump. **Now what do we do with it**?. Most of us **don’t have a clue** on what to look in a thread dump and quite honestly there are not to be blamed. Thread dump will have 90% useless data. However, with practice you can quickly become a master of troubleshooting using Thread dumps.

**The most important piece of information (often times, the only piece of information) you need to look for is the RUNNABLE THREADS**. These are the threads that were executing in the JVM at the time of Thread dump. In Java, a Thread has various states as follows:

**NEW**: A new Thread. Not started yet.

**RUNNABLE**: A thread that is executing in the Java virtual machine.

**BLOCKED**: A thread that is blocked waiting for a monitor lock .

**WAITING**: A thread that is waiting indefinitely for another thread.

**TIMED\_WAITING**: A thread that is waiting for another thread for a specified time.

**TERMINATED**:  An exited Thread

**Note**: The above states are Oracle HotSpot JVM specific. Jrockit JVM, for example has states such as ‘alive’ and ‘parked’. And IBM JVM has states like ‘zombie’ and ‘suspended’.

In the next part, we are going to look at the fun part:

**Step by Step procedure to analyze a Thread dump**

**Using Free tools to analyze the Thread dump**

**Answer all the burning questions that we asked at the beginning of part1**

Stay tuned.

# **Troubleshooting using Java Thread Dumps – Part 3**

*by* KARUN SUBRAMANIAN *on* OCTOBER 30, 2014

Alright. Now let us dive into step-by-step procedure on actually analyzing the Thread Dump.

**Step by Step procedure to analyze a Thread dump**

1. Open the Thread dump in your favorite text editor.  
**Note**: Java Threadump (aka javacore) is generally a text file that can be opened using any text editor. If you cannot open it, chances are it is not a JVM thread dump. Perhaps you are trying to open a heap dump or an Operating System core file.  
2. Search for the string **RUNNABLE**. If you are running IBM JVM, look for the string **“state:R”.**  
3. There can be lot of RUNNABLE threads. What you want to do is pay attention to the Threads t**hat are running your application code**. They will be evident from stack trace. Look for com.<companyname>.<packagename>.<classname>.<methodname>. You can ignore Application Server specific Threads. It is not unusual to see tons of Application Server threads (for example listen sockets) in the Thread dump.  
**Example**: The Thread below is clearly **NOT** running your application code at that moment  
because it does not show any “com.<companyname>.<packagename>.<classname>.<methodname” in its stack trace.

“control: Socket[addr=/190.184.61.31,port=27241,localport=63741]” daemon prio=6  
tid=0x000000001437d000 nid=0x1244 runnable [0x000000005673f000]  
java.lang.Thread.State: RUNNABLE  
at java.net.SocketInputStream.socketRead0(Native Method)at java.net.SocketInputStream.read(SocketInputStream.java:129)  
at java.net.SocketInputStream.read(SocketInputStream.java:182)  
at java.io.FilterInputStream.read(FilterInputStream.java:66)  
at  
org.jboss.remoting.transport.bisocket.BisocketServerInvoker$CControlConnectionThread.run(BisocketServerInvoker.java:1033)

Locked ownable synchronizers:– None

4. Once you locate the RUNNABLE threads that are executing your application code, find out which method is being executed by following the stack trace. **You may get  assistance from development team if needed.**Also note the Thread ID.  
5. Now open the other thread dumps that you took and see if the Thread ID identified in the  
previous Thread dump is still executing the same method. If it is, you are closing in on the  
root cause. Open the next Thread dump and see if the same Thread is doing the same thing. If it is, you have a solid lead in finding the root cause.  
6. Find out what exactly the method does. May be it is waiting for response from a remote web service. May be it is waiting for response from a Database Server. Whatever it is, that is most likely the root cause.

**Note**: Another useful observation from a Thread dump is the ‘number of Threads’. If you see the total number of threads in unusually amounts (for example several hundreds or even couple of thousands), you have a problem.

**Note**: IBM JVM prints a WARNING in SystemOut.log if a Thread has been running for a  
configured amount of time.

**Note:** Weblogic will declare a Thread as ‘Stuck’ if it is active for a configured amount of time (default 600 seconds). The health of the Application Server will also change to ‘WARNING’.

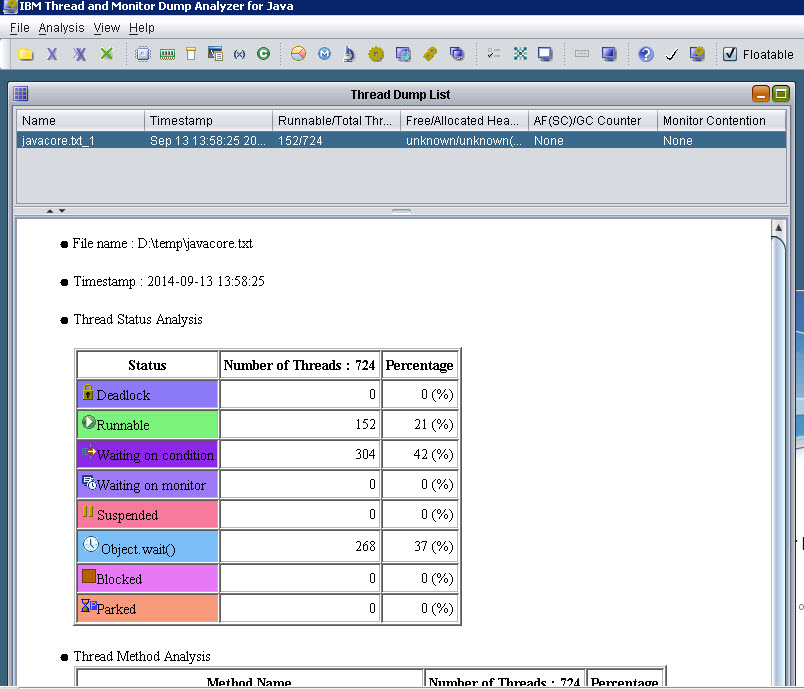
Thread dead locks

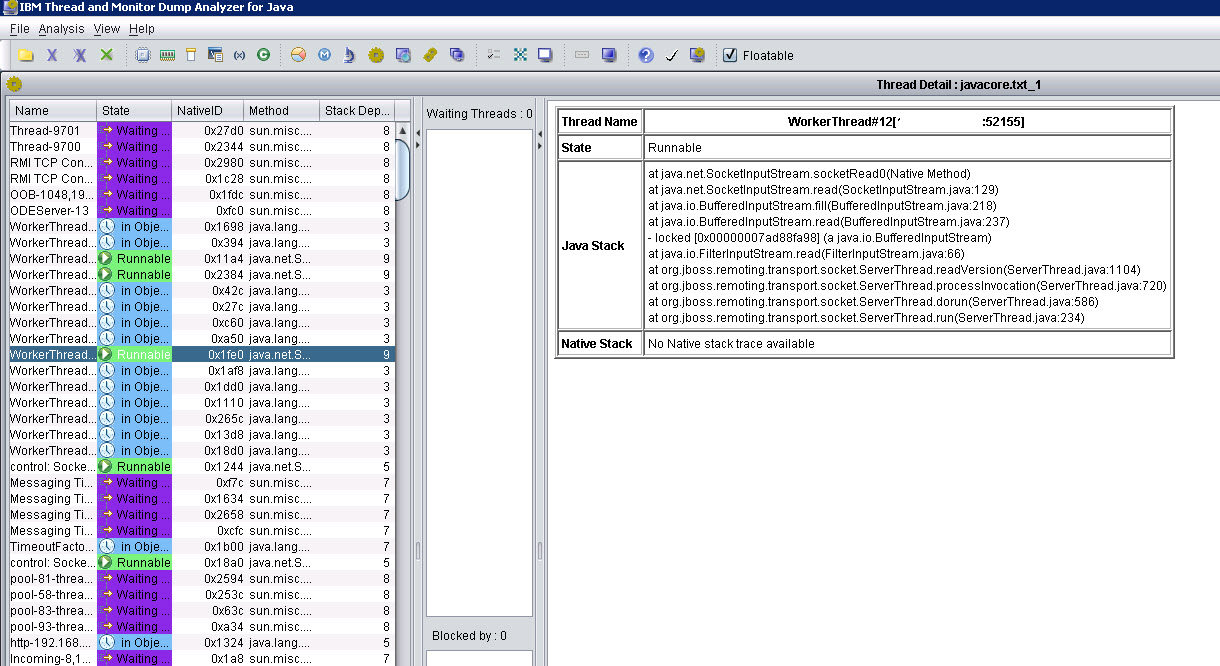
Occasionally you may run into a dead lock situation. Where Thread 1 is waiting for Thread 2 to finish and Thread 2 is waiting for Thread 1 to finish. Few JVMs actually detect the dead locks automatically and print them in Standard Out Log file.

To find out dead locks from Thread Dump, look for the ‘WAITING’ threads and see what it is waiting on. As before, don’t worry about Application Server specific Threads .Just focus on Threads that are running the application code.

**Using free tools to analyze Thread dump**

Instead of viewing the Thread dump in a text editor, you can use any of the free Thread dump analyzer tools. My favorite is IBM Thread and Monitor Dump Analyzer for Java. See couple of screen shots below:

[](http://karunsubramanian.com/wp-content/uploads/2014/10/IBM-Thread-Analzyer.jpg)

[](http://karunsubramanian.com/wp-content/uploads/2014/10/IBM-Thread-Analzyer1.jpg)

Few folks have had success with **Samurai**, another tool for analyzing Java thread dumps. There is also**TDA (Thread dump analyzer**) which is primarily used for HotSpot JVM only. And Visual VM has a plugin named ‘**Threads Inspector**’ which you may want to try out as well.

With this newly gained knowledge, let’s try to answer the questions asked at the beginning of this lesson.

1. **What does it mean to say ‘There must be hung Threads’ in the Application Server?**

It means one or more Threads are executing the same thing for long duration. Take 3 Thread dumps at 10 seconds interval and check the ‘RUNNABLE’ Threads that are running your application code (the stack trace should tell you if it is running the application code or some application server specific internal code). Mostly hung threads are due to non responding remote systems.

**2. Have you exhausted the available Threads in Application Server?**

It is possible to exhaust all the Threads in an Application Server. It is a serious issue though. The CPU utilization on the Server will be extremely high (close to or equal to 100%). Take a Thread dump if you can and see all the ‘RUNNABLE’ Threads. It could be cause of a non-responding backend system and your application does not have a timeout (indefinitely waiting for remote system to respond). Note that some Application Server lets you leave the ‘Maximum number of Threads’ dynamic. Meaning the Application server is **free to spawn as many threads as possible**. **This is NOT a good practice**. Always have an upper limit on the number of Threads.

**3. Which Thread Pool is used for Web Container?**

It depends on the Application Server product. For example, with WebSphere, there is a Web Container’ Thread pool that you can configure.

**4. Do I have to configure the number of JDBC Connections in the connection pool equal to the number of Threads I have in the Application Server?**

No. Number of JDBC connections and number of Threads do NOT have 1:1 relationship. Typically you will have fewer Threads than the number of JDBC connections. This is because the same Thread can be reused for several connections.

**5. How can I find out which thread is currently running right now in the Application Server?**

Take a Thread dump and look for ‘RUNNABLE’ Threads.

**6. Is ‘number of Threads’ configurable? Can I have unlimited Threads?**  
It depends on the Application Server product you are using. Generally yes. It is configurable. Make sure you have an upper limit (start with perhaps 50 threads) for the Thread Pool. Only a through Load test can help you determine the optimal upper limit. You cannot have ‘unlimited Threads’. It is limited by your hardware.**It is NOT  good practice to have Application Server ‘automatically expand’ the Thread Pool size**, unless you are absolutely sure about what you are doing.

**7. How can I find out all Threads that are currently running?**

Take a Thread dump. Thread dump reveals ALL the Threads in the JVM. It does not matter which STATE the Thread is in.

**8. Is there a Thread Dead Lock going on?**

Some JVMs (Jrockit) will print in System Out if there is a dead lock. You can always take a Thread dump and see if there are dead locks. You can use free tools like**IBM Thread and Monitor Dump Analyzer for Java** to analyze the Thread dump.

**9. How can I kill hung (or stuck) Threads?**

First of all it is a risky business to forcibly kill a Thread. What if it leaves corrupted data in the database? This is precisely why **Oracle deprecated Thread.stop (method)**. This technically means ‘**you cannot kill a Thread in Java**’. **Restarting the JVM is an option** but it will terminate other Threads that may be doing useful work. But every environment is different. If you absolutely know the application you are managing in and out and exactly know what it is doing, **you can try interrupting stuck Threads by some creative ways**. For example, if the thread is waiting for response from a remote service, can you stop that service? When you stop that service, most probably your application will receive**‘socket closed by peer’ and exit the Thread.** But there is no guarantee.Oracle introduced **Thread.interrupt** that may work in some situations. But you will have to write a program to do this.For WebSphere Application Server, you can try “**Hung Thread Interrupter**” tool from **IBM Alpha Works.**

There you have it. Troubleshooting Thread issues, though intimidating, **can be made lot easier by carefully analyzing Thread dumps**. Why don’t you take a Thread dump of an Application Server (perhaps in your test environment) and start looking around ?. The more you familiarize with Thread dumps, better you will become with analyzing them. Always consult the Application Server manuals when in doubt. Keep reading articles related to Threads and Thread dump Analysis available on the Internet. Before you know you will be a master at troubleshooting using Thread dumps.

# **Troubleshooting GC: Using Jconsole**

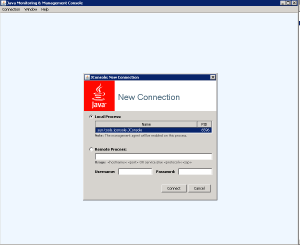
*by* KARUN SUBRAMANIAN *on* JUNE 11, 2015

One of the built-in tools that can be used to view the memory usage graphically is jconsole. It comes with JDK. Jconsole uses JMX (Java Management Extension) to pull several vital stats from the running Java application. While jconsole may seem little outdated (and it is) it is still valuable if you want a quick GUI to view JVM stats

**How to invoke Jconsole ?**

Jconsole is in <JDKHOME>/bin

In windows, you just type ‘jconsole’ to bring up Jconsole GUI (provided you have the PATH environment variable updated to include <JDKHOME>/bin

[](http://karunsubramanian.com/wp-content/uploads/2015/06/Jconsole-1.png)

Note: If you are in an Unix environment, you may have to export the DISPLAY to point to your PC to invoke GUI

The opening screen shows the available java processes that jconsole can monitor. If you don’t see your application in the list, you are NOT running jconsole as the user that is running your application. If this is the case, you can try enabling JMX for remote management as follows:

**Very critical Note: Use the options below ONLY in development environment as it opens serious loop holes.**

Add the following command line options to the java command line of your application and restart the application:

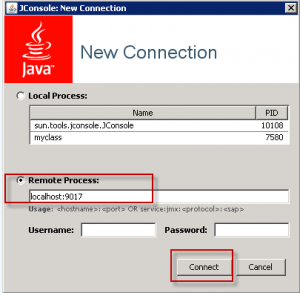
-Dcom.sun.management.jmxremote.port=<portNum>. For example,

-Dcom.sun.management.jmxremote.port=9017

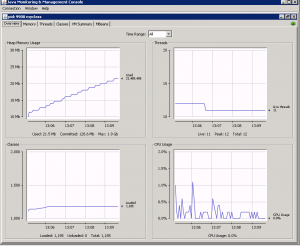
-Dcom.sun.management.jmxremote.authenticate=false

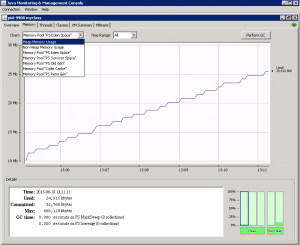
-Dcom.sun.management.jmxremote.ssl=false

Once the application is started, invoke jconsole and type ‘localhost:9017’ in the ‘Remote processes’ field and click ‘Connect’ as shown below:

[](http://karunsubramanian.com/wp-content/uploads/2015/06/Jconsole-2png.png)

Once connected, you get access to wealth of information. See couple of screen shots below:

[](http://karunsubramanian.com/wp-content/uploads/2015/06/Jconsole-3.png)

[](http://karunsubramanian.com/wp-content/uploads/2015/06/Jconsole-4.png)

Neat, hah?

Next let us look at an another unsung built in monitoring tool – **visualvm**

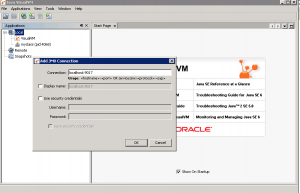
# **Troubleshooting GC: VisualVM**

VisualVM is a free monitoring tool for the JVM. It is much more feature rich than Jconsole. It comes built in on the Oracle JDKs. You can also download a standalone version from visualvm.java.net.

**Invoking visualvm:**

The tool ‘jvisualvm’ is located in bin directory under ‘JDK Home’.

**Jvisualvm**

[](http://karunsubramanian.com/wp-content/uploads/2015/06/visualvm4.png)

Note, similar to jconsole, if you are running visualvm with the same user id as the application, the application will appear under ‘Local’ and you can begin monitoring right away. However, if that is notthe case, you must first enable JMX remote management by adding the following parameters to the Java command line.

**Very critical Note: Use the options below ONLY in development environment as it opens serious loop holes**.

Add the following command line options to the java command line of your application and restart the application:

-Dcom.sun.management.jmxremote.port=<portNum>. For example,

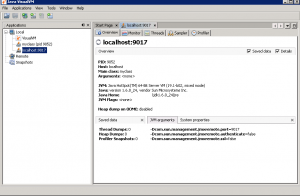
-Dcom.sun.management.jmxremote.port=9017

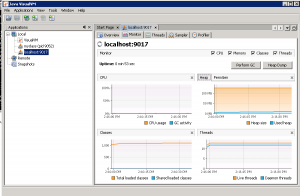
-Dcom.sun.management.jmxremote.authenticate=false

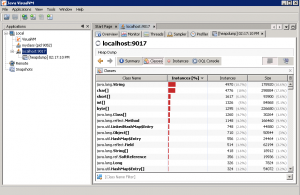
-Dcom.sun.management.jmxremote.ssl=false

Once the application is restarted, you can connect to it from visualVM by adding a ‘JMX connection’ as shown above (right click on ‘Local’ and choose ‘Add JMX Connection’)

Once connected, you can play with various graphs, even take a heap dump on the fly and analyze it. See the screen shots below:

[](http://karunsubramanian.com/wp-content/uploads/2015/06/visualvm3.png)

[](http://karunsubramanian.com/wp-content/uploads/2015/06/visualvm2.png)

[](http://karunsubramanian.com/wp-content/uploads/2015/06/visualvm1.png)

Awesome progress!! You have come a long way in troubleshooting JVM Memory issues. Now, you must be thinking – this is all good, but is there a tool that can do the research for me and just display the results and recommendations? You are in luck. IBM has just the right tool for you –**IBM Pattern Modeling and Analysis Tool for Java Garbage Collector**. This neat tool takes your verbose GC log and does the analysis for you. Let’s take a look.