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.

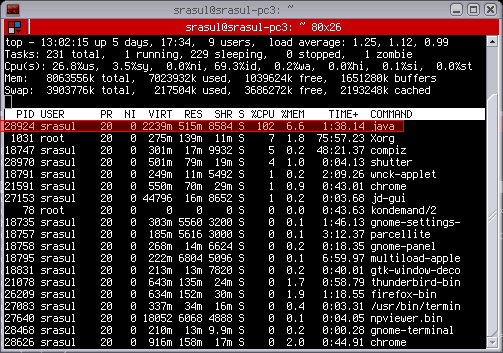
First lets start the StartThreads class.

$ 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:

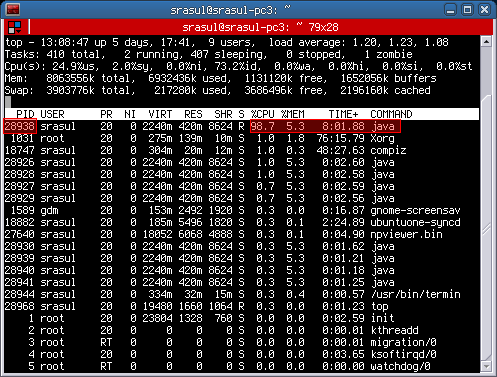
-H : Threads toggle

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.)

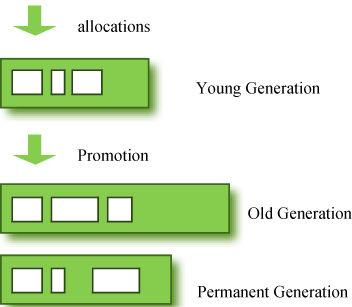
* Most objects soon become unreachable.
* References from old objects to young objects only exist in small numbers.

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.



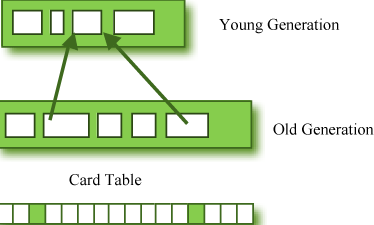
**Figure 1: GC Area & Data Flow.**

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.

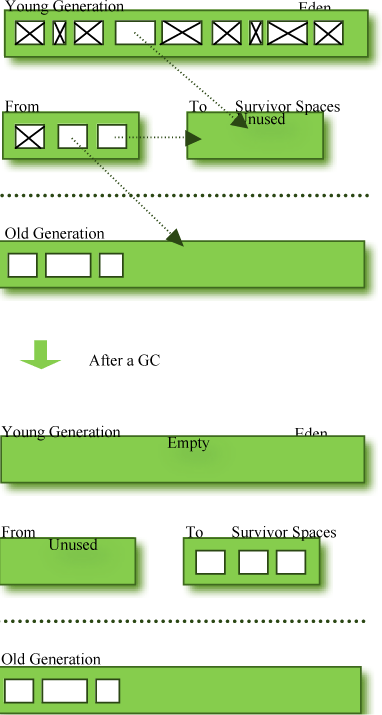
* One **Eden** space
* Two **Survivor** spaces

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.

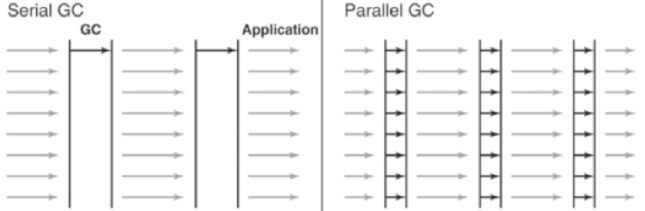
### Serial GC (-XX:+UseSerialGC)

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)



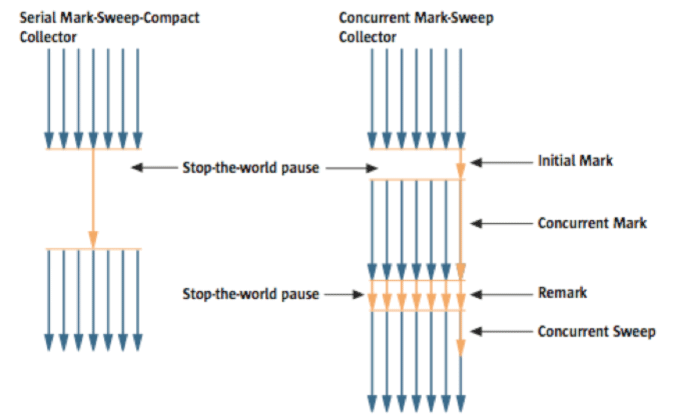
**Figure 4: Difference between the Serial GC and Parallel GC.**

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**.

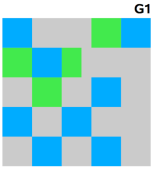
While this GC type has the advantage of short stop-the-world time, it also has the following disadvantages.

* 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”**

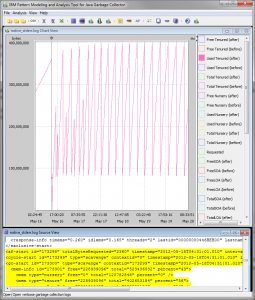
[](http://karunsubramanian.com/wp-content/uploads/2014/07/gc.png)

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.