Java GC
How to stop hate it and start love it

brought by Peter Kovgan

Contents

GC aim

Find objects, that are no longer in use and free the memory associated with those objects.

VM must periodically search the heap for unused object.

It is not sufficient only free memory. VM must do memory fragmentation (occasionally).

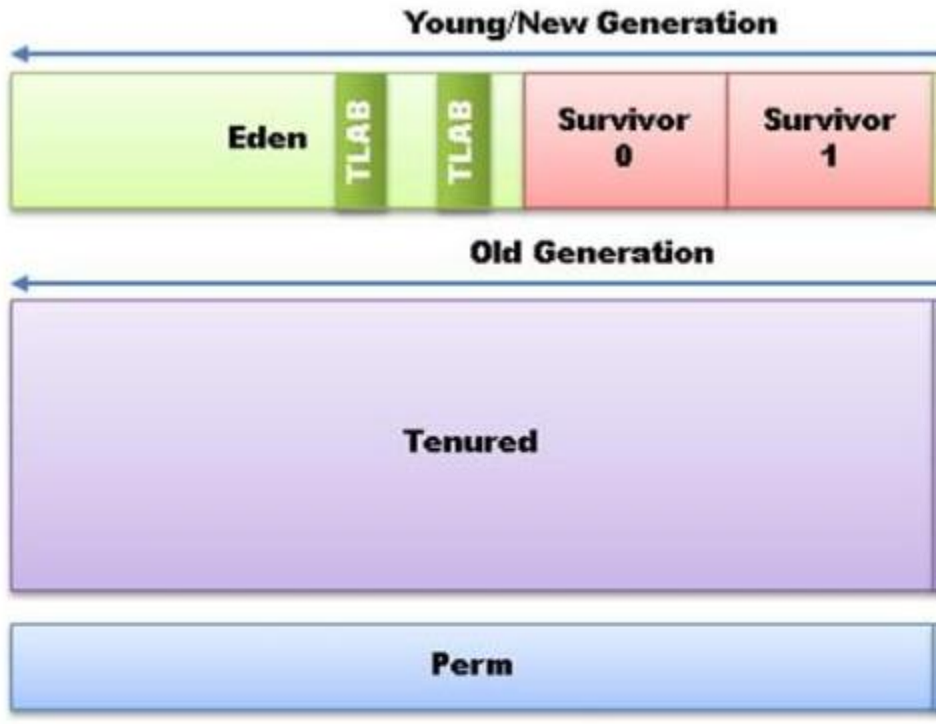
Note: Safepoints

Note, to perform some CG operations VM must stop all threads in “safepoints”. Thinking in GC way, you have to adopt thinking in “concurrent way”. Some time may be spent (from GC prospective) in “waiting for all threads entering the safepoint”. Then, to launch all threads from their safepoints – new thread’s re-scheduling must happen (CPU intensive operation) – that why GC execution may turn to be quite resource consuming.
More about safepoints:
http://chriskirk.blogspot.co.il/2013/09/what-is-java-safepoint.html
http://jpbempel.blogspot.co.il/2013/03/safety-first-safepoints.html

Java VM memory

Heap

That’s how generational heap looks



Parts of heap

Permanent space (*not a heap part, strictly speaking*)
<= Java 7 – PermGen
Java 8 – called metaspace (**MaxMetaspaceSize** can limit its size, by default it is limited only by available native memory).
To hold permanent metadata of loaded classes.
https://dzone.com/articles/java-8-permgen-metaspace

Young space = Eden + Survivor1 + Survivor2 – serves to collect new objects using minor collections

Tenured space (Old) - to collect objects survived in several minor collections in major collection.

TLAB - Thread Local Allocation Buffer – minimal chunk of memory provided for allocation. (Average allocation time in java almost always faster than in simple C app, but it takes more memory)

Native memory

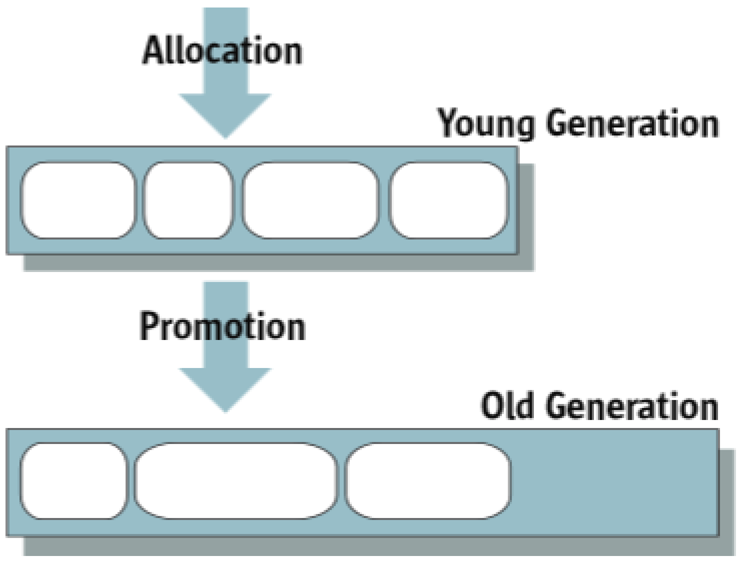
Note! Java also uses “native memory”. It is not a part of the heap. It is allocated for internal java work, by some libraries (jdbc,e.g.), by some APIs(JNI, etc).

Native memory usage, if not tracked, may expand beyond physical available memory and start use SWAP. Periodically check your native memory: add to your start script

-XX:NativeMemoryTracking=[off|summary|detail]

and explore result by command: jcmd <PID> VM.native_memory (not present in java 6).
Note, enabling that you have 5-10% overhead.

Java 6 users read that: http://www.oracle.com/technetwork/java/javase/memleaks-137499.html



Stack

Stack is region of memory, where all “operative”, method’s data stored
Stack != Heap

Stack operates fast (heap operates relatively slowly).

Stack contains frames, each frame defines 1 method.

Stack is FILO. Stack stores primitives and **references to objects**.

StackOverflowError means: An application recurses too deeply.

How to save objects in stack and avoid GC!

In latest java there is a nice feature, called "escape analysis" – this technique allows save **objects** in stack instead of heap, if the object accessed only by 1 thread and its life time limited by 1 method (1 frame).
No GC required for such objects. Use Java 8 for **better** escape analysis, although it is here in Java 7 by default.

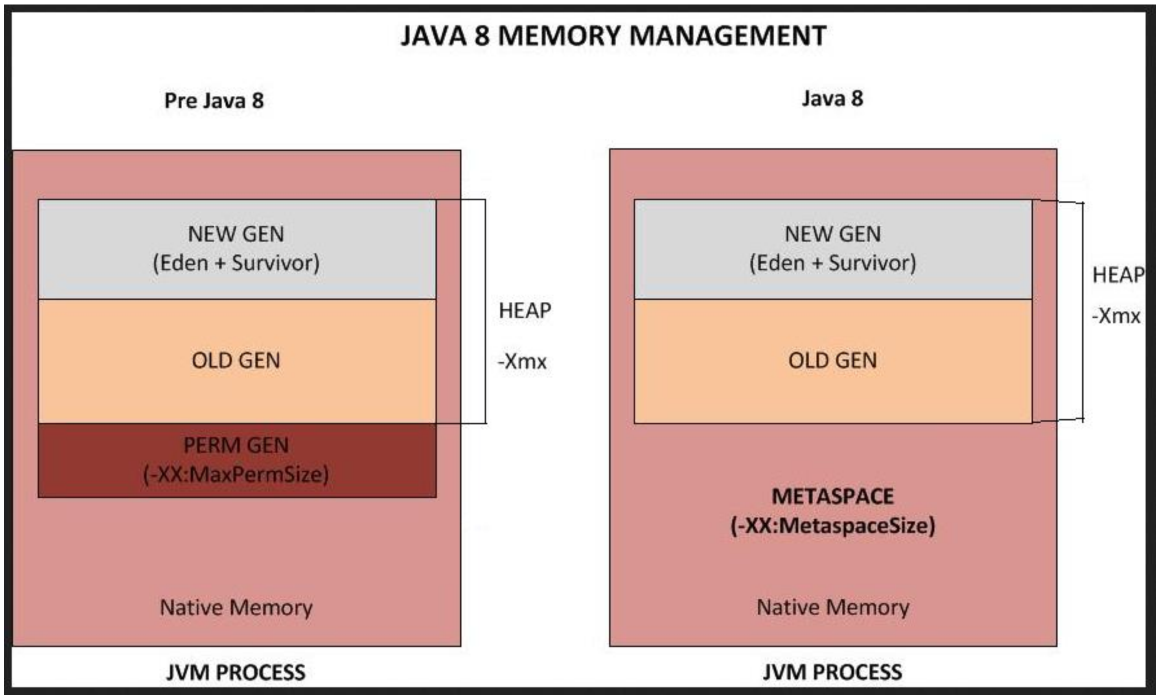
https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html#escapeAnalysis

https://weblogs.java.net/blog/forax/archive/2009/10/06/jdk7-do-escape-analysis-default

http://www.burnison.ca/articles/escape-analysis-stack-allocated-objects

Use objects in scope of 1 method/1 thread to boost performance of your app – this will be stack based allocation/deallocation, no need in GC.

Metaspace in Java 8



Important notes

BTW1:
There is a "goal", set by GC for itself behind the curtain.
It is hidden in the parameter (applied by default): -XX:GCTimeRatio=99
That means, GC strives to take less than 1% of application time.
While it strives to do so, it changes its behavior with time to meet the goal.
(One of your aims as GC manager is to allow it, not to interfere in this process – write less parameters)

BTW2:
Quick GC is always in fight with low latency of application requests. So when you made everything to achieve as fast as possible GC, you might increase the average latency of your application without noticing that. Nothing comes without price.

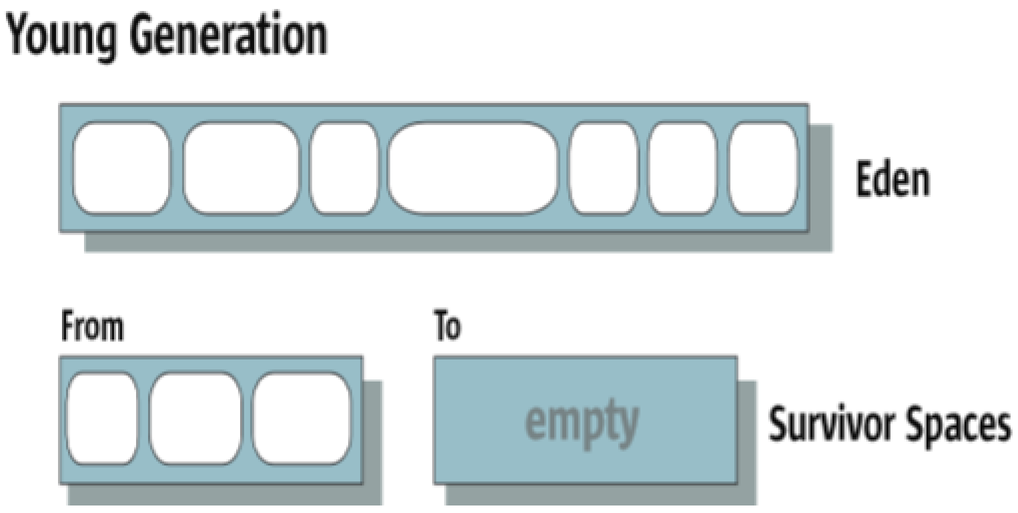
BTW3:
Throughput usually fights with latency, so increasing latency, you may win better throughput – so , as result, for high throughput systems – GC time is not such a concern.
(relevant to our support tools projects)

BTW3:
Trade systems can sacrifice throughput to reach consistent low latency – trade prefer ***deterministic latency*** *even if that hurts high throughput. To achieve that goal such system should strive to reduce amount of live objects. That's why trade systems should strive produce less(not so important)* ***and shorter living(important)*** *objects to minimize GC effect. (relevant to LPA, etc..)*

BTW4:
TLAB size based on eden size, threads number and thread allocation rate and changes (by default).
TLAB can be resized. If application allocates very big objects – consider increase TLAB size. But better use smaller objects. For more info see parameters: -XX:+PrintTLAB (to monitor) – XX:TLABSize=n (to define size instead of dynamic default) ,-XX:-ResizeTLAN (to avoid dynamic resize), -XX:MinTLABSize=n(set min size).
JFR is powerful in TLAB monitoring.

Minor and major GC

Minor GC



Generational GC assumes that a lot of objects die young.

What is dead object?

Object considered dead, if it cannot be traced from “GC ROOTS”

GC ROOTS: classes loaded by system class loader, live thread, stack local, JNI local,…etc..(a lot of them)

The default NewRatio for the Server JVM is 2: the old generation occupies 2/3 of the heap while the new generation occupies 1/3. (It is not so, with greater amount of memory, so do not be sure – monitor it)

When the young generation (more exactly Eden) filled up, all application threads must be stopped by the minor GC and young generation (Eden) emptied out. Objects without reference discarded, objects traceable from ROOTS – moved (most frequently to survivor space, but if it is full/small – to tenured).

That means **minor** GC always performs fragmentation (compaction) of Eden.

Phases of minor GC: Mark (who is traced from roots) and **Copy (move traced to another space)**

Note: All GC algorithms working with young generation are “stop-the-world” (!)

(Because they need a “safe point” in the application lifecycle)

Each java object has a header to keep info how many GC cycles it survived.

-XX:MaxTenuringThreshold defines how many (max) times object attempted to be collected before it promoted to Tenured(old) space. Default=15.

Journey of object has following pattern:

- allocated in eden
- copied from eden to survival space due to young GC
- copied from survival to (other) survival space due to young GC (this could happen few times)
- promoted from survival (or possible eden) to old space due to young GC (or full GC)

Actual tenuring threshold is **dynamically adjusted by JVM**, but MaxTenuringThreshold sets an upper limit on it.

"Premature promotion" is a problem. It means survivor space is too small.

If you set MaxTenuringThreshold=0, all objects will be promoted immediately.

Eventually tenured space is filled up. In such case (in some other cases) major GC performed.

Survivor space is self-adjusting

GC algorithm decides increase or decrease survivor space based on how full is the survivor space after GC.

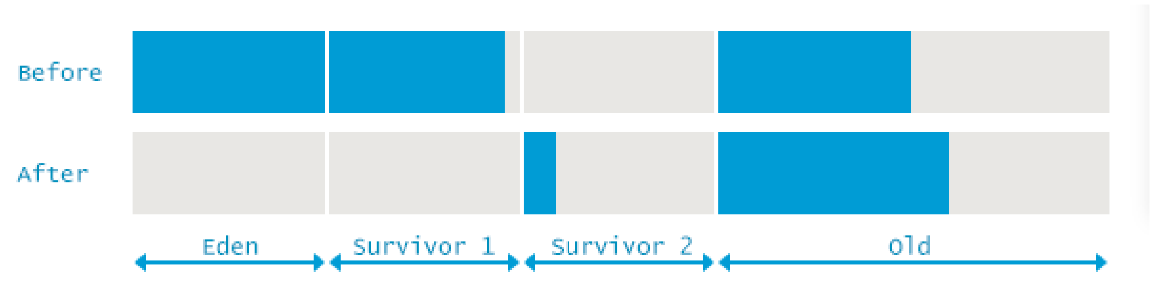
Survivor space=young_size / initial_survivior_ratio + 2 =   YS /8+2 = YS / 10 (by default)

JVM may increase SS to –XX: MinSurvivorRatio=N flag   => max SS size= YS/ MinSurvivorRatio(3 default) + 2 => YS/5 (by default) =>20% Young (by default)

So SS "changes" in runtime from 10% to 20% of young by default.

Use –XX: MinSurvivorRatio to set another size limit of SS. (**If you know why**)

Minor GC looks like that on time scale:



Major (and FULL)

Major and full garbage collections in Oracle VM

Reasons to perform major CG:
1) Major collections collect the *old* generation so that objects can be promoted from the *young* generation. **The *old* generation collector will try to predict** when it needs to collect to avoid a promotion failure from the *young* generation. In such case, old GC collector may start work without any trigger from young GC collector.

2) **Promotion failure** – failure to allocate space in tenured generation during minor GC. In case of promotional failure - major GCs are triggered by minor GCs - and this is FULL GC - it involves promoting all live objects from the *young* generations followed by a collection and compaction of the *old* generation. (promotion failure may happen even if tenured is 50% free, but very badly compacted - that's why compaction matters sometime)

That means: there are cases, when FULL GC happens and there are cases when only major GC happens.

BTW:
Developing app for deterministic GC (as for trade) , tuning GC for that purpose, we need try to avoid (or make as rare as possible):

-promotion failure

-major GC (if we especially cute)

Common phases

(it is common view, not real phases, only an idea)
- Mark – find object referenced from roots

- Clean– "delete" not reachable objects

- Compact – defragmentation

The serial collector

runs by default on Windows 32 bit machines, on single CPU machines

Single thread used to process the heap

Stops all app threads and also compacts the tenured generation.

-XX:+UseSerialGC – enables it

Where it is most effective:
< 100 Mb memory, Windows 32, 1 CPU

The throughput collector

Default on server class machines, 64 bit and multi CPU unix machines.

This one can use multiple threads to collect tenured (starting from v.7)

It always can use multiple threads to collect young

It does not work in parallel to app threads! It works in multiple threads sometime, but app **threads must be stopped**.

This collector is default in most cases, but if you need enable it do -XX:+UseParallelOldGC (for tenured)and -XX:+UseParallelGC (for young)

It called throughput, because it able provide highest throughput in situations, when occasional long GC pause is not a problem. Very suitable for server batch applications. Bad for trade.

The CMS collector

It is designed to minimize long "stop-the-world" pauses of previous collectors.

Use –XX:+UseConcMarkSweepGC   –XX:+UseParNewGC to enable it.

It makes "stop-the-world" during **young** generation collection.

It stops app threads for **very short pauses** during the major collection.

It performs most part of major collection not stopping the application threads.

There is no compaction performed by background threads.

If CMS does not get enough CPU to complete their tasks or it must be compacted, then it reverses to behavior of **simple serial collector** – it stops all threads, collects and compacts the tenured, using 1 thread. With Full GC CMS collector is == Serial.

Phases:

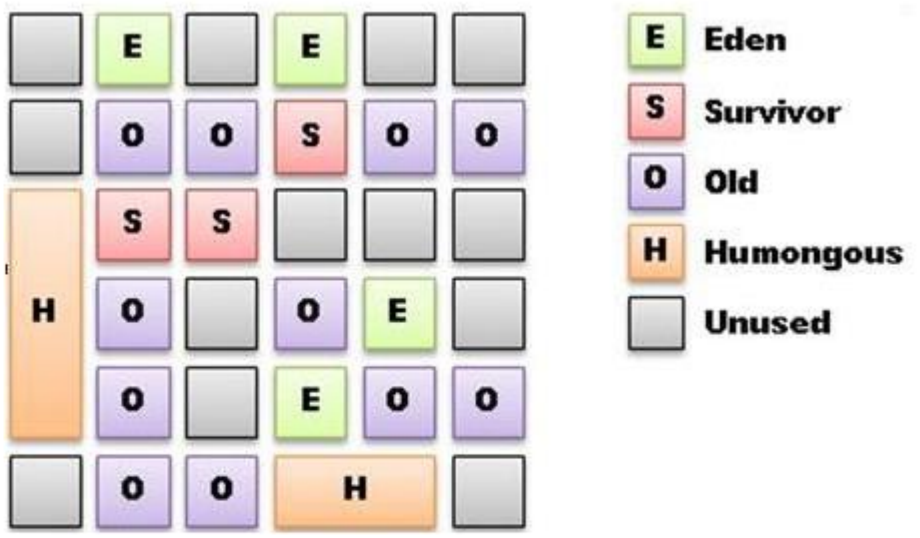
-Initial mark (**STW**) – finds initial references of objects close to ROOTS
-Concurrent mark – find all live objects
-Precleaning – optimization phase(optional), fixes possible problems of previous phase
-Remark (**STW**) – finally mark exactly what is live
-Sweep – adds dead objects to "free lists" (after that they are really "dead")
-Reset – cleanup the state of CMS collector

Trade-off: increased CPU usage

This GC may consume 1 CPU 100% of time, leaving you with 75% of CPU on 4 cores machine.

Appropriate for Trade apps. The best one till Java 8.

The G1 collector ("The Garbage first"): Java 8 and Big Heap friend.



Starts be acceptable since Java 7u10, has significant performance benefit from java 8

It separates heap on fixed size regions with variable purpose.

The concepts of allocation, copying to survivor space and promotion to old generation are similar to previous HotSpot GC implementations. Eden regions and survivor regions still make up the young generation.

For larger heaps, greater than 4 Gb, for minimal pauses.

Objects larger than 50% of a region are allocated in humongous ("huge") regions, that are a multiple of region size.

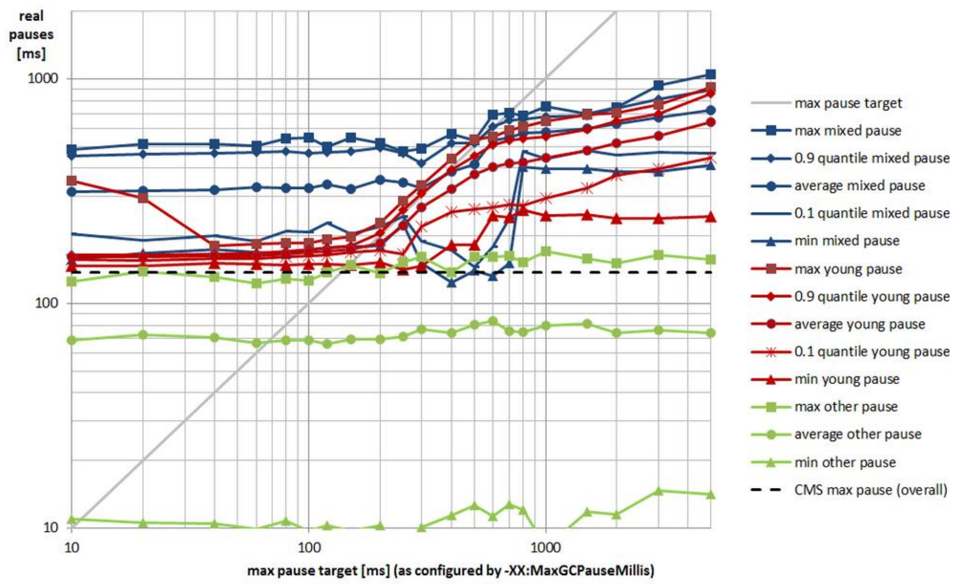
It has also separate regions in tenured generation.

So moving objects from region to region it compacts tenured space while it is going, reducing need in Full GC.

G1 tends to **reduce the frequency of the worst-case pauses** seen by CMS because of fragmentation at the cost of extended *minor* collections and incremental compactions of the *old* generation. Most pauses end up being constrained to regional rather than full heap compactions. Has problem with "popular objects".

-XX:+UseG1GC

-XX:MaxGCPauseMillis – set pause duration target(!), do not set it too small



Trade-off: increased CPU usage, increased memory usage and it is relatively new, so for some apps (strange cases) it may be not well adjusted.

BTW:

Region_size = 1 << log (Xms / 2048), that means with Xmx32G and Xms2G region size=1Mb, there are 32000 regions(!) in that example.

Region size may be changed (-XX:G1HeapRegionSize=n), **good if you have 2048 regions - that would be G1 optimal configuration.**

More about it:

http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All

All GC (except Serial) use multiple threads

You can set:
-XX:ParallelGCThreads=Y

But you can use default:

Default: one thread per CPU , up to 8

Further: 1 thread per 5/8 CPUs

Example: 24 CPUs=>8+((24-8)*5/8)=18 threads

Make your choice

G1	CMS	Parallel	Serial
Huge RAM(heap) Java 8 or late 7 Enough CPU (Trade)	Enough CPU (Java 6+) Need predictable latency (Trade)	Need super productive batch jobs, tolerate occasional big GC pauses	Have small RAM, Have 1 CPU Windows 32 bit

Short GC applicative dark secrets:

-ensure most objects die fast

-minimize objects size (what really you need in your message?)

-**reduce processing time** (efficient parsing and delegating)

-less threads (use pools)

-less global variables

-immutable objects preferable

-use messaging instead of synchronization (consider move to scala from java and use actor pattern..e.g.)

Example of efficient app:
Kafka. It has 0 Full GCs.

Q&A

Can I tell GC when start?

No. You only can suggest. System.gc();

How can I tell, when object is dead?

finalize() method – but time when it called may be very inappropriate.
You should not base any applicative logic on finalize() method

Tuning

Important note

Remember - GC is adaptive, so let it tune itself first, then monitor it and profile the app/heap/threads, and only after that start TUNE, if you FOUND THE REASON!!!

Do not tune GC, if you have not monitored it, have not profiled it, know nothing about its actual use of resources!!!

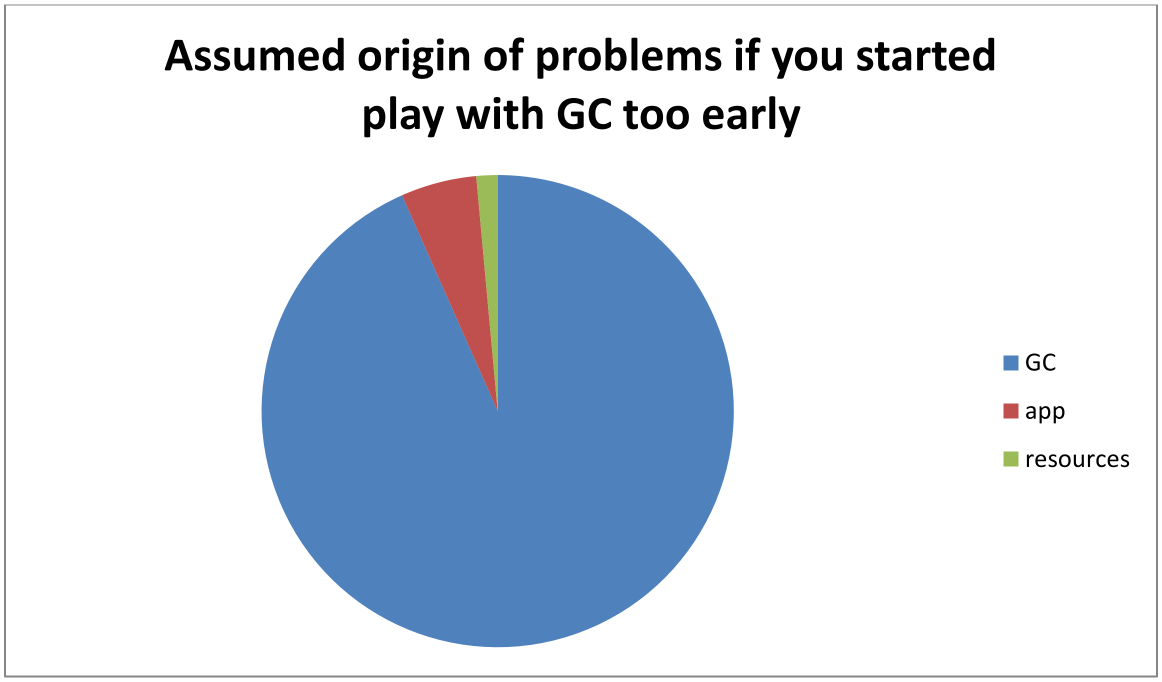
Do not tune GC, if you JUST ASSUME that GC is wrong.

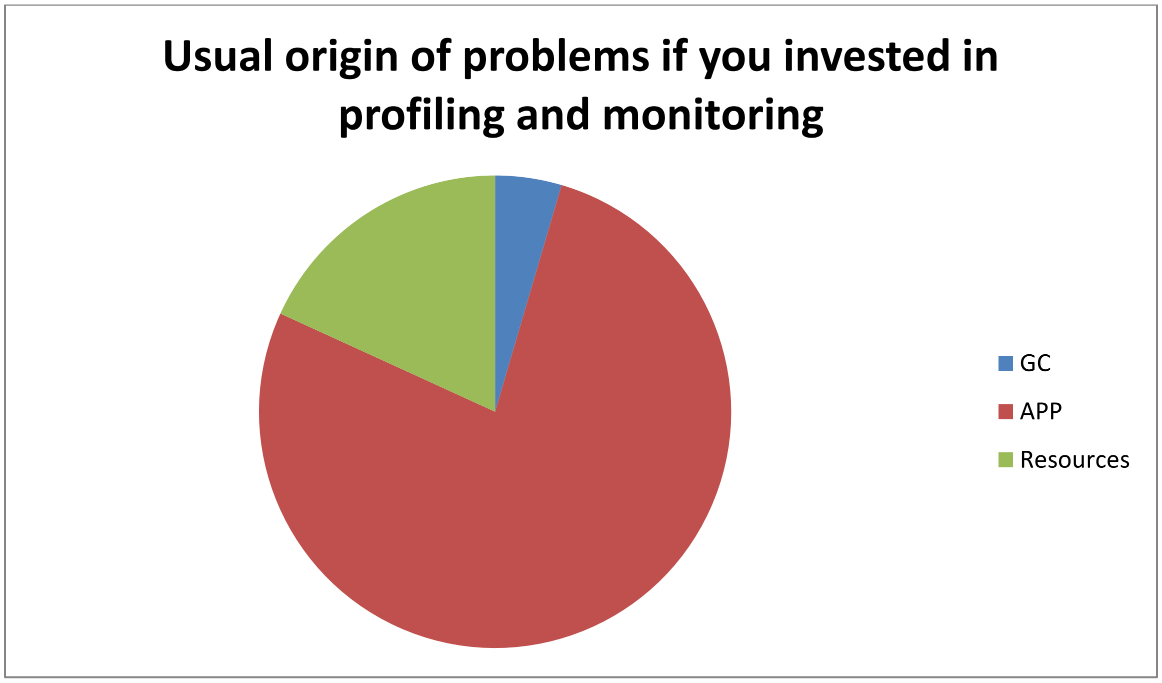
Do not tune it if you have not installed log of it, have not analyzed the log!!!

Do not tune GC, if you have not checked CPU and memory, swap of the box you run on.

We never TUNE something, that we have no idea of:
a) what is the current state?
b) what state we want achieve?
c) how we achieve that?

Many assumed GC problems are in fact applicative or resource problems.


Assumed origin of problems if you started play with GC too early
GC
app
resources


Usual origin of problems if you invested in profiling and monitoring
GC
APP
Resources

1. **If you tune nothing – it will work with defaults and may be it is the best you can do**
   (defaults depends on OS, CPU architecture 64/32, CPU number, available RAM)
   Example: For linux 64 bit default max heap size = MIN (32Gb , RAM / 4)

2. It is self-adjustable, so you need tune as less as possible, starting from minimum.
   -XX:-UseAdaptiveSizePolicy – will bring an end to almost all GC intelligence, do not use it without REAL NEED

3. Be careful defining –Xmx, if you take more, that you have RAM, OS may start use SWAP space for your heap – this is extremely slow operational mode

   To check your java process SWAP usage run command:
   grep --color VmSwap /proc/<pid>/status
   or
   free –m can show the SWAP usage for all processes

   Define –Xmx and –Xms
   For Sun/Oracle VM they better be equal for all server cases.
   (For IBM VM they better be different)

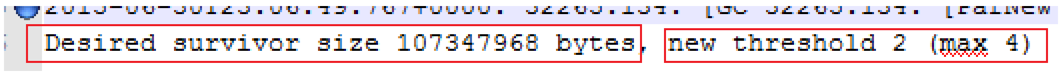
   Example: -Xmx2048m –Xms2048m

4. Now run with that definition and explore your GC logs and behavior using various tools (**See tools section at the bottom**)

5. Only if you found problems , you need additional tweaks

How to analyze GC Log (On the example of the CMS)

Promotion from young to tenured
It happens accordingly to “desired survivor size”.

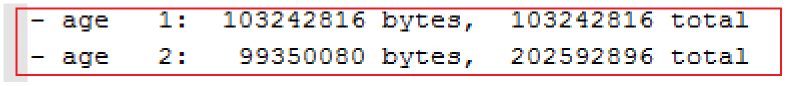


```
2013-06-30T23:06:49.767+0000: 32263.134: [GC 32263.134: [ParNew
Desired survivor size 107347968 bytes, new threshold 2 (max 4)
```

GC determined, that **desired survivor size** (better name is “**target utilization to “To” space”**) is 107 MB, formula is SURVIVOR_SPACE * TargetSurvivorRatio / 100 , default TargetSurvivorRatio=50,
so “**target utilization to “To” space**” by default is SURVIVOR_SPACE / 2 – half of SS.

If more objects left in SS after GC, than “**target utilization to “To” space**” allows, GC will try **promote** live objects to tenured earlier, than before.

Object ages in young generation after the minor GC



```
- age   1:  103242816 bytes,   103242816 total
- age   2:   99350080 bytes,   202592896 total
```

There are objects of 2 ages still “alive” in survivor space.

- age 1: {a} bytes, {a} total <- How many objects have survived one collection.

- age 2: {b} bytes, {a+b} total

The “total” at the age 2 shows, that 202Mb landed in “To” survivor space.

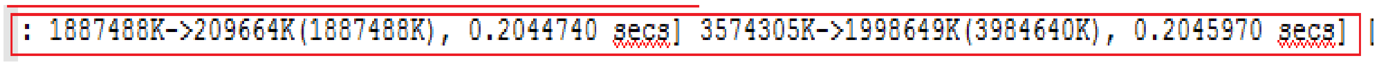
But **target utilization** is ~ 107Mb – that means GC did not meet its target and number of survived (live) objects in “To” space after GC is greater, than would be “desired” (not so good).

It may bring GC to conclusion: “I want meet “**desired utilization**” requirement, so I must promote **more** objects from survivor space to tenured (**more==earlier than before)**, reducing ping-pong time between “from” and “to” survivor spaces”.

Is it a problem? Not always. It is a problem only **if such “early promoted” objects had real chance to die in “ping-pong” phase**. It is **a warning**.

If you have too frequent Major/FULL GCs – that means this **warning is a problem –** that means too early promotion causes premature (problematic) usage of tenured space.

Generations change:

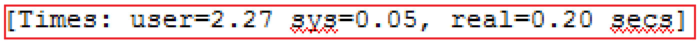


```
: 1887488K->209664K(1887488K), 0.2044740 secs] 3574305K->1998649K(3984640K), 0.2045970 secs] [
```

: {pre-collection young gen usage}K->{post-collection young gen usage}K({Total young gen size}K), {young gen time} secs]

{pre-collection heap size}K->{post-collection heap size}K({total heap size}K), {total collection time} secs]

Time the app stopped:



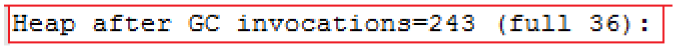
```
[Times: user=2.27 sys=0.05, real=0.20 secs]
```

Duration of the GC event, measured in different categories:

user – Total CPU time that was consumed by Garbage Collector threads during this collection

sys – Time spent by GC in OS calls or waiting for system event

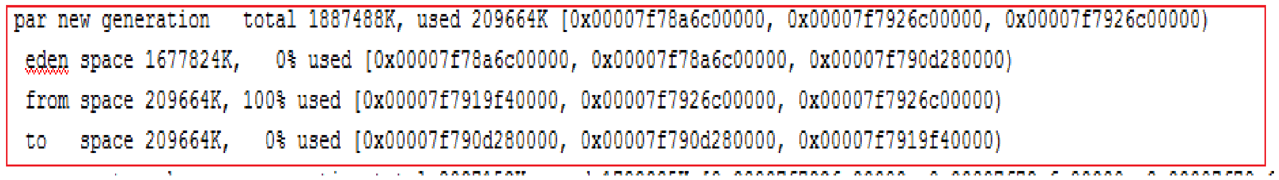
real (this is time stopped) – Clock time for which your application was stopped. (As Serial Garbage Collector always uses just a single thread, real time is thus equal to the sum of user and system times)



```
Heap after GC invocations=243 (full 36):
```

Before that moment there were 243 minor and 36 major (mostly easy) collections (they are not full!)
(these majors in CMS may work in parallel with app threads)

Young generation sizes(detailed view of young generation):



```
par new generation   total 1887488K, used 209664K [0x00007f78a6c00000, 0x00007f7926c00000, 0x00007f7926c00000)
 eden space 1677824K,   0% used [0x00007f78a6c00000, 0x00007f78a6c00000, 0x00007f790d280000)
 from space 209664K, 100% used [0x00007f7919f40000, 0x00007f7926c00000, 0x00007f7926c00000)
 to   space 209664K,   0% used [0x00007f790d280000, 0x00007f790d280000, 0x00007f7919f40000)
```

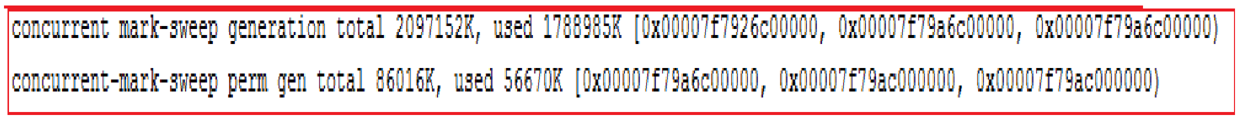
Par new generation – young generation statistics, total space, used space [pointers in memory, defining current size of the space]

Eden space size, % usage[pointers in memory, defining current size of the space]

From space size, % usage[pointers in memory, defining current size of the space]

To space size, % usage[pointers in memory, defining current size of the space]

Tenured and perm sizes (detailed view of tenured and perm):

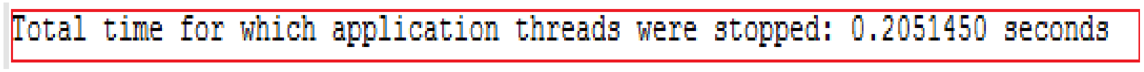


```
concurrent mark-sweep generation total 2097152K, used 1788985K [0x00007f7926c00000, 0x00007f79a6c00000, 0x00007f79a6c00000)
concurrent-mark-sweep perm gen total 86016K, used 56670K [0x00007f79a6c00000, 0x00007f79ac000000, 0x00007f79ac000000)
```

Tenured total, used, [pointers in memory, defining current size of the space]

Perm gen total, used [pointers in memory, defining current size of the space]

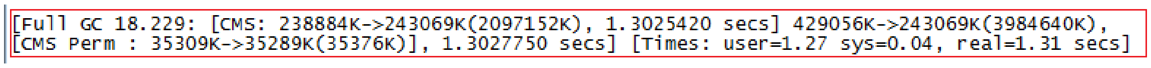
Time the app threads were stopped:



```
Total time for which application threads were stopped: 0.2051450 seconds
```

Not only includes the GC pauses. Other reasons why app threads may be stopped – locks, synchronization, etc… (profile with JProfiler to find potential problems)

If Full GC happens, it is indicated by Full GC log, see times and frequency of FULL GC



```
[Full GC 18.229: [CMS: 238884K->243069K(2097152K), 1.3025420 secs] 429056K->243069K(3984640K),
[CMS Perm : 35309K->35289K(35376K)], 1.3027750 secs] [Times: user=1.27 sys=0.04, real=1.31 secs]
```

[Full GC <time from start> : [CMS: pre->post(total) tenured, time to clean] heap before -> heap after(max heap),
[CMS Perm: – cleaned up permanent generation (pre->post (max) usage of permanent generation, gc took 1.3028 sec)

See more about logs:

http://blog.ragozin.info/2011/12/garbage-collection-in-hotspot-jvm.html

Tools to analyze GC (and heap)

Recommendation: develop on JDK, not on JRE

You have more profiling options tools in JDK

1. See what current parameters are (Run first without your custom parameters to see what this machine brings by default – then start customize, if you unsatisfied)

jmap -heap <PID>

This command prints parameters of the heap – well to know before start GC analysis and tuning.
Unfortunately not all JDK have this option. But try.
If you failed, try another profiling tool

2. How to enable GC log.
   Enable GC and VM parameters logging:

   -Xloggc:${APP_DIR}/jvm.log, //equal to -verbose:gc, but you know, where place the log
   -XX:+PrintGCDetails, //more detailed info about minor GC
   -XX:+PrintGCApplicationStoppedTime, //print time of stopped app
   -XX:+PrintGCApplicationConcurrentTime ,//print time of running up
   -XX:+PrintPromotionFailure, //alert about promotion failure
   -XX:+PrintTenuringDistribution, //print ages of objects after minor collections
   -XX:+PrintGCDateStamps(work from 6u20) //human readable dates

   In addition, to know, what parameters constrain the VM:
   -XX:+PrintCommandLineFlags //Dumps ALL the default flags and values
   Java 7+:
   -XX:+PrintFlagsInitial //Dumps ALL the flags before processing command line and ergonomics.
   -XX:+PrintFlagsFinal //Dumps ALL the flags after processing command line and ergonomics.

3. Enable JMX remote access (you can see threads, heap and VM parameters in runtime with JRE<JDK>/bin/**jconsole**):

   -Dcom.sun.management.jmxremote.port=${JMX_PORT}
   -Dcom.sun.management.jmxremote
   -Dcom.sun.management.jmxremote.authenticate=false
   -Dcom.sun.management.jmxremote.ssl=false

4. **Recommended:** Visual VM with plugin Visual GC (https://visualvm.java.net/) with jstatd (part of JDK, not in JRE) and VisualGC plugin:
   http://www.oracle.com/technetwork/java/visualgc-136680.html

   create policy file: **jstatd.all.policy** file content(change to point your JDK):

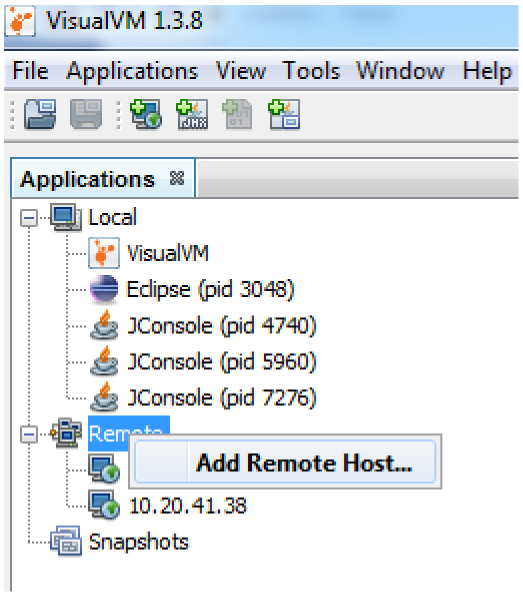
```
grant codebase "file:/path/to/jdk/lib/tools.jar" {
  permission java.security.AllPermission;
};
```

This command will start statistics collection on server (write without \n , in one line):

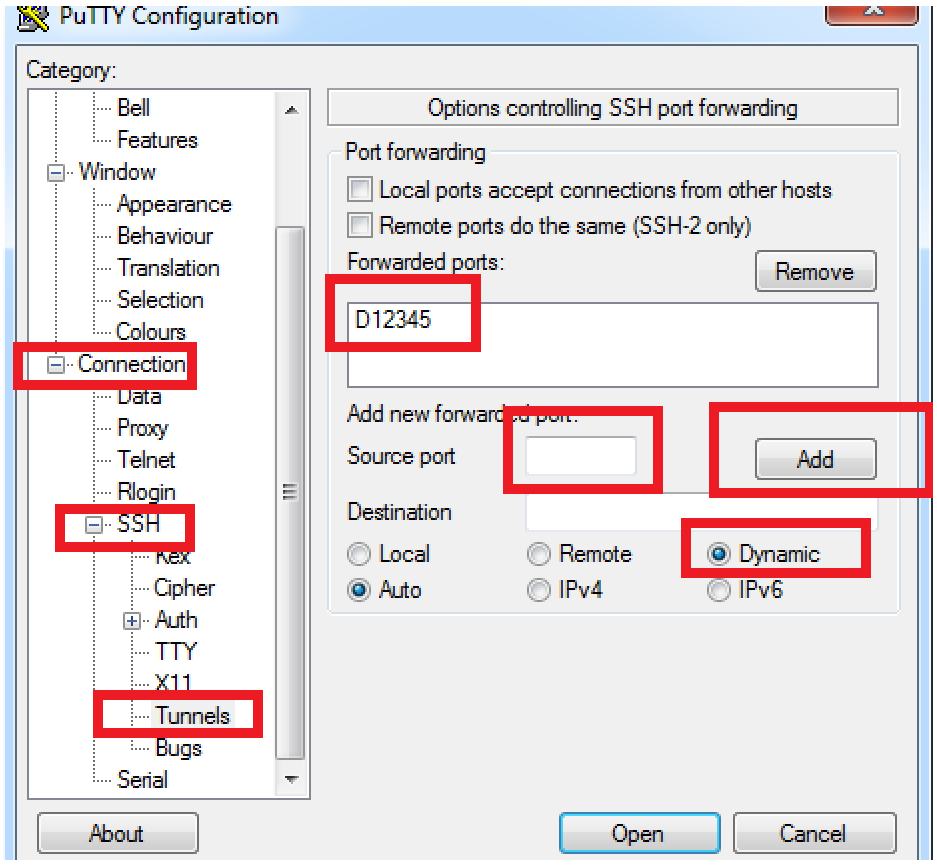
```
/path/to/jdk/bin/jstatd -p 12345
-J-Djava.security.policy=/path/to/policy/file/jstatd.all.policy
```

-p 12345 means port, on which statistics will be exposed for remote user

In VisualVM, configure to listen on this port:



If you have problem to connect to this host (host behind proxy and allows only SSH access), use SSH tunnel of putty:



Port forwarding

☐ Local ports accept connections from other hosts

☐ Remote ports do the same (SSH-2 only)

Forwarded ports: D12345

Add new forwarded port:

Source port:

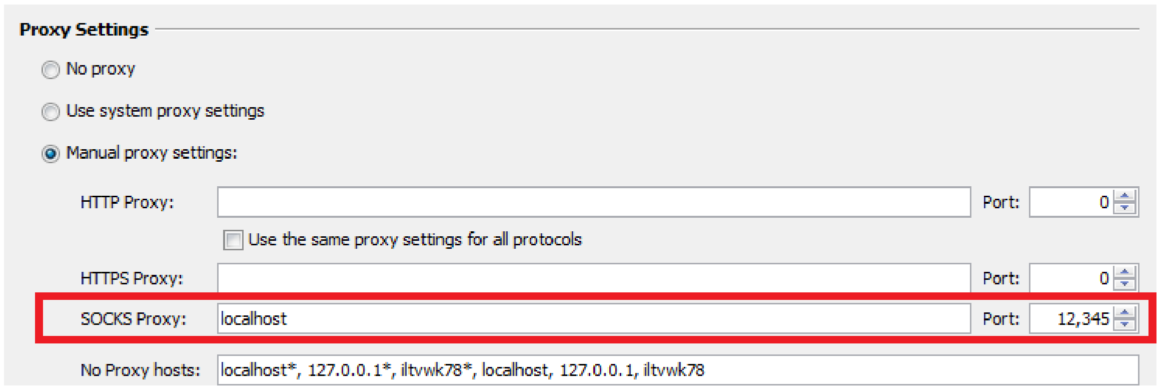
Destination:

Local Remote [✓ Dynamic]

[✓ Auto] IPv4 IPv6

And configure VisualVM to accept this tunnel:

Tools->Options->Network->



Proxy Settings

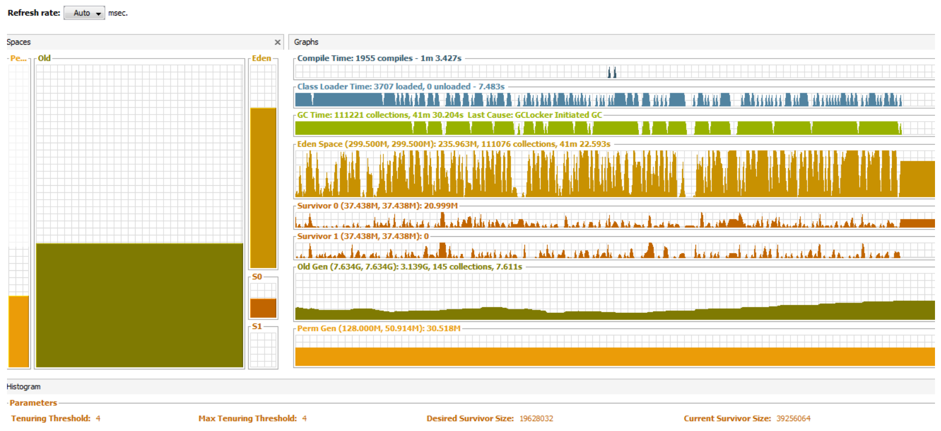
No proxy

Use system proxy settings

[✓ Manual proxy settings:]

HTTP Proxy:		Port:	0
	☐ Use the same proxy settings for all protocols		
HTTPS Proxy:		Port:	0
SOCKS Proxy:	localhost	Port:	12,345
No Proxy hosts:	localhost*, 127.0.0.1*, iltvwk78*, localhost, 127.0.0.1, iltvwk78		

You must see nice GC UI:



Refresh rate: Auto msec.

Compile Time: 1955 compiles - 1m 3.427s

Class Loader Time: 3707 loaded, 0 unloaded - 7.483s

GC Time: 111221 collections, 41m 30.204s Last Cause: GCLocker Initiated GC

Eden Space (299.500M, 299.500M): 235.963M, 111076 collections, 41m 22.593s

Survivor 0 (37.438M, 37.438M): 20.999M

Survivor 1 (37.438M, 37.438M): 0

Old Gen (7.634G, 7.634G): 3.139G, 145 collections, 7.611s

Perm Gen (128.000M, 50.914M): 30.518M

Parameters

Tenuring Threshold: 4 Max Tenuring Threshold: 4 Desired Survivor Size: 19628032 Current Survivor Size: 39256064

5. Jconsole and JMX enabled on the app level
   (SSH client may be required to provide a tunnel - additional tool, that you need to connect computers, that reside behind the proxy and have only SSH port open)

6. **Recommended:** GC log analyzers, example - http://sourceforge.net/projects/gcviewer/

7. Jstat(jre and jdk) or jstat**d (jdk)** – to obtain different stats
8. jinfo (jdk) – take vm parameters and system properties
9. Samurai (http://yusuke.homeip.net/samurai/en/index.html) – not sure it works with last java or with all GC

Heap analysis:

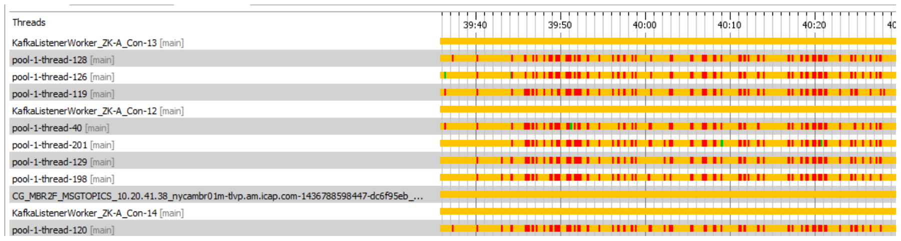
1. JMC (Java mission control) and (JFR) Java Flight Recorder (commercial), since JDK 7u40 – part of the Oracle JDK.
2. Jmap to get heap:
   Example: jmap -dump:format=b,file=heap.bin <pid>

   jmap -histo <PID> to count objects with their sizes online (good to see who grows too much interactively)

3. Hprof – heap profiler
4. HAT – heap analysis tool
5. **But recommended:** JProfiler
6. Memory Analyzer (Eclipse) – tries to load everything in memory, not very good

Threads analysis:

ps huH p <PID> | wc –l
jstack <pid>
tda (https://java.net/projects/tda) + jconsole
JProfiler
(Example JProfiler output)



Problems and solutions

Recommendation: develop on JDK, not on JRE
You have more profiling options tools in JDK

Problem	Solution
App cannot start	Check that your max heap is sufficient/not too large. Check that permanent generation is sufficient and increase it, if it is not. -XX:MaxPermSize=254m Check, that your compiler is <= of your runtime java To ensure you chose the right Java in runtime, always explicitly define JAVA_HOME=…, PATH=$JAVA_HOME/bin:$PATH, do not believe it is given. Check CPU less /proc/cpuinfo Check memory free –m Be sure the thing you run is OK for that box. With some apps, check, that the box (less /etc/security/limits.conf) has sufficient amount of allowed threads to open for the user: elvis_presly soft nproc 1000
Frequent FULL GC	May be memory leak (use jmap and JProfiler to see what is growing)

	May be heap is small (increase the heap)
Your think you have large GC pauses (in minor or/and in major).	1. Make sure you do not mix applicative problems and GC pauses. Carefully observe GC and application logs. Check, that pause is not IO relative or lock relative. **Profile your app.** “Total time for which application threads were stopped: 0.0002190 seconds” – includes all threads idle time, no matter what a reason, not only GC. 2. Make sure you carefully use threads. Check your threads number and if it grows uncontrollable – fix that, using thread pool. ps huH p <PID> | wc –l jstack <PID> tda 3. Make sure your CPU is sufficient. If several apps share the same BOX, be sure they all have CPU time. Efficient GC depends on free CPU. 4. Make sure your IO(disk and sockets) is not the pause reason. 5. Make sure your algorithm is good (right) for you -XX:+UseSerialGC -XX:+UseParallelOldGC -XX:+UseParallelGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+UseG1GC 6. Make sure you have enough heap 7. Make sure you have heap, that is not too large 8. Make sure, that minor collections are not too frequent, they better look like saw’s teeth (שיניים של מסור) See the pic below ** To improve situation increase young space. -XX:NewSize and -XX:MaxNewSize or -XX:NewRatio (relative to tenured) That usually means you also need increase the whole

	heap. (Tenured must be at least the same as young, in most cases it better be 2-3 times larger than young). **Note: Good "teeth" are possible only if your app written right!** 9. Check GC log for small survivor space, See picture *** If this is the case, you need increase survivor space, Use -XX:SurvivorRatio to change this space size relatively to eden. 10. May be you major GC is too long. in such case – carefully reduce heap size and pay attention – may be you have no enough CPU time for GC and your shiny CMS converts to Serial collector in such conditions(It may happen). 11. Have Java 8 and enough memory – use G1 12. Very specific and last: Address nitty-dritty settings of your specific garbage collector – that is for hard and unusual cases. Here you need exactly know what you do.
Too frequent FULL GC	Memory leak? Try set (on tests only!): -XX:+PrintClassHistogramBeforeFullGC -XX:+PrintClassHistogramAfterFullGC To find a reason (over populating object) in logs. Premature promotion? (small heap) LOW CPU? May be objects retired to Old too early? Check, may be something can be done with -XX:MaxTenuringThreshold -XX:TenuringThreshold May be you need more PING-PONG on the way to OLD?
Your app starts work slowly and more slowly and ..	Check your RAM and swap space. Check your CPUs. May be what you have is very weak box?

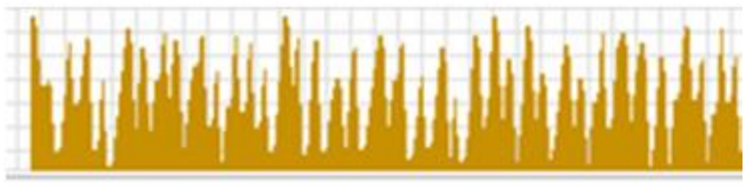
You CPU is high, while your application must process a lot of data.	It is excellent! You won! Mostly LOW CPU means a problem. Low CPU is usually IO problem. But be sure GC has enough CPU for its work.
Your CPU is low, and GC pauses are still large	**<u>May be</u> some CPU time is ready for GC, but you do not use it.** Try to change number of threads, that do the GC job: 1) ParallelGCThreads (default = 8+((N-8) * 5 / 8), N – CPU cores) works for: -collection of young, when XX:+UseParallelGC or XX:+UseParNewGC or XX:+UseG1GC -collection of old, when XX:+UseParallelOldGC -stop the world phases of CMS (not , when FULL GC) -stop the world phases of G1 (not, when FULL GC) 2) Only for CMS collector: -XX:ConcGCThreads (default=(3+ParallelGCThreads) / 4) CMS background threads performing **not** stop the world operations.
You found, that your app is simple, predictable, must not surprise you and you want gain yet 1-3% of performance from the app. **You are so sure of yourself**, that nobody can stop you from…	..disabling of adaptive sizing.. very hard step for people, who know what they do: -XX:-UseAdaptiveSizePolicy You will define all sizes (eden, survivor,old, perm) of all generations strictly and without possibility for collector to change them in runtime. This step stops all adaptations – providing very small benefit.
You are working with particular collector and need the last performance possible from it. Use this collector specific settings.	Read the collector manual and if you are sure, change the defaults. **examples for CMS**: 1) when start concurrent part of GC, default ~70% -XX:CMSInitiatingOccupancyFraction=N -XX:+UseCMSInitiatingOccupancyOnly 2)-XX:ConcGCThreads -XX:ConcGCThreads (default=(3+ParallelGCThreads) / 4) CMS background threads performing **not** stop the world operations.
Your manager tells you, that C++ application will do that much faster (not investing in code review and profiling).	It depends who writes it. Good C++ programmer, able leverage all BOX CPU cores in concurrent programming is 1 on 10000. (Exactly such programmers wrote Java GC)

	Network latency (10-200msc) in most cases kills the benefit of C++ in distributed system. In any case, you need be highly suspicious if such claim has been made without proper profiling and code review.

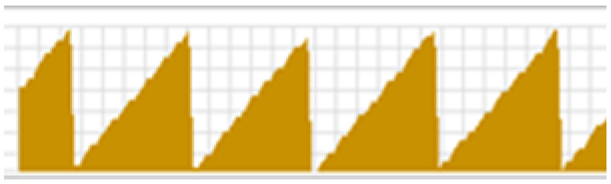
Minor GC problems illustrations

** Frequent minor GC

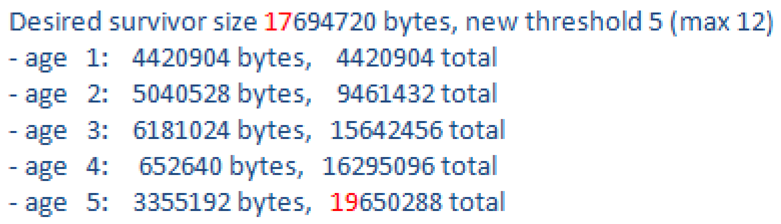
Bad minor GC (too frequent and not stable):



Fine minor GC:



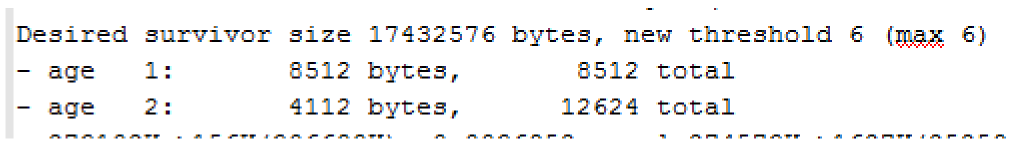
*** Insufficient survivor space (with -XX:+PrintTenuringDistribution, **works with CMS collector**)



```
Desired survivor size 17694720 bytes, new threshold 5 (max 12)
- age   1:   4420904 bytes,    4420904 total
- age   2:   5040528 bytes,    9461432 total
- age   3:   6181024 bytes,   15642456 total
- age   4:    652640 bytes,   16295096 total
- age   5:   3355192 bytes,   19650288 total
```

a) total after 5-th (oldest possible) age collection is 19Mb, while target is 17 Mb
It means survivor space is smaller, than required.

b) much better look:



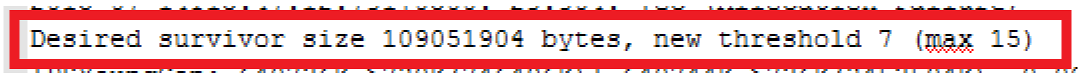
```
Desired survivor size 17432576 bytes, new threshold 6 (max 6)
- age   1:        8512 bytes,        8512 total
- age   2:        4112 bytes,       12624 total
```

Target utilization is **17** Mb ("Desired survivor size")
Total after utilization is **12** kb, much less than 17 Mb – that means - almost every new object died during the ping-pong phase

It is also much nicer, when each elder age has less remaining objects, than the previous age, **that means the app works in more predictable manner** – elder objects have more chance to die. This is cool. Try achieve that.

Note:

If **ages** are NOT VISIBLE in log, looks at this:



```
Desired survivor size 109051904 bytes, new threshold 7 (max 15)
```

If new threshold is going smaller and smaller with time and riches 1,2, that is the sign of small survivor space.

Potential problems that we can see

1 – total is greater than desired

2 – greater ages contain greater number of objects than younger ages

(it tells "too frequent and too inefficient minors, hey, may be your young is too small!")

Material

Memory leaks:

http://www.oracle.com/technetwork/java/javase/memleaks-137499.html

Java 6 GC

http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

Java 8 GC

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/

G1

http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html

GC blogs

http://www.infoq.com/articles/Java_Garbage_Collection_Distilled

https://blog.codecentric.de/en/2012/08/useful-jvm-flags-part-5-young-generation-garbage-collection/

Java memory model

http://www.cs.umd.edu/~pugh/java/memoryModel/

GC Logs

https://plumbr.eu/blog/garbage-collection/understanding-garbage-collection-logs