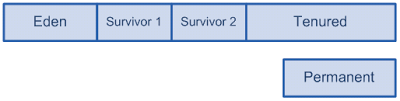
**Understanding GC pauses in JVM, HotSpot's minor GC.**

Stop the world pauses of JVM due to work of garbage collector are known foes of java based application. HotSpot JVM has a set of very advanced and tunable garbage collectors, but to find optimal configuration it is very important to understand an exact mechanics garbage collection algorithms. This article one of series explaining how exactly GC spends our precious CPU cycle during stop-the-world pauses. An algorithm for young space garbage collection in HotSpot is explained in this issue.

**Structure of heap**

Most of modern GCs are generational. That means java heap memory is separated into few spaces. Spaces are usually distinguished by “age” of objects. Objects are allocated in young space, then, if they survive long enough, eventually promoted to old (tenured) space. That approach rely on hypothesis that most object “die young”, i.e. majority of objects are becoming garbage shortly after being allocated. All HotSpot garbage collectors are separating memory into 5 spaces (though for G1 collector spaces may not be continuous).

[](http://3.bp.blogspot.com/-NwrMpaA2Wf8/TeYGgmdzLXI/AAAAAAAAI0c/GeBQSgDhXQE/s1600/blog-6.png)

* Eden are space there objects are allocated,
* Survivor spaces are used to receive object during young (or minor GC),
* Tenured space is for long lived objects,
* Permanent space is for JVM own objects (like classes and JITed code), it is behaves very like tenured space so we will ignore it for rest of article.

Eden and 2 survivor spaces together are called young space.

**HotSpot GC algorithms overview**

HotSpot JVM is implementing few algorithms for GC which are combined in few possible GC profiles.

* Serial generational collector (-XX:+UseSerialGC).
* Parallel for young space, serial for old space generational collector (-XX:+UseParallelGC).
* Parallel young and old space generational collector (-XX:+UseParallelOldGC).
* Concurrent mark sweep with serial young space collector (-XX:+UseConcMarkSweepGC   
  –XX:-UseParNewGC).
* Concurrent mark sweep with parallel young space collector (-XX:+UseConcMarkSweepGC –XX:+UseParNewGC).
* G1 garbage collector (-XX:+UseG1GC).

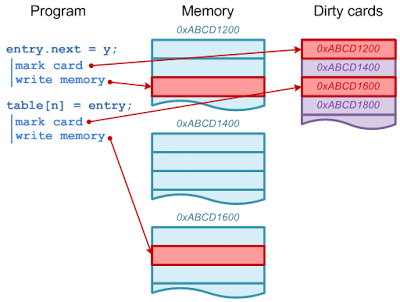
All profiles except G1 are using almost same young space collection algorithms (with serial vs parallel variations).

**Write barrier**

Key point of generational GC is what it does need to collect entire heap each time, but just portion of it (e.g. young space). But to achieve this JVM have to implement special machinery called “write barrier”. There 2 types of write barriers implemented in HotSpot: dirty cards and snapshot-at-the-beginning (SATB). SATB write barrier is used in G1 algorithms (which is not covered in this article). All other algorithms are using dirty cards.

**Dirty cards write-barrier (card marking)**

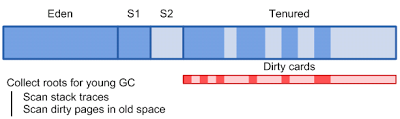
Principle of dirty card write-barrier is very simple. Each time when program modifies reference in memory, it should mark modified memory page as dirty. There is a special card table in JVM and each 512 byte page of memory has associated byte in card table.

[](http://4.bp.blogspot.com/-4BSXuryn7Ss/TeYHvOqwltI/AAAAAAAAI0g/kfH9vN0KgrI/s1600/blog-7.png)

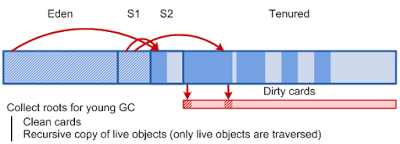
**Young space collection algorithm**

Almost all new objects (there are few exception when new object can be allocated directly in old space) are allocated in Eden space. To be more effective HotSpot is using thread local allocation blocks (TLAB) for allocation of new objects, but TLAB themselves are allocated in Eden. Once Eden becomes full minor GC is triggered. Goal of minor GC is to clear fresh garbage in Eden space. Copy-collection algorithm is used (live objects are copied to another space, and then whole space is marked as free memory). But before start collecting live objects, JVM should find all root references. Root references for minor GC are references from stack and all references from old space.

Normally collection of all reference from old space will require scanning through all objects in old space. That is why we need write-barrier. All objects in young space have been created (or relocated) since last reset of write-barrier, so non-dirty pages cannot have references into young space. This means we can scan only object in dirty pages.

[](http://4.bp.blogspot.com/-JeoPvpns3II/TeYILJO5mnI/AAAAAAAAI0k/rIuloiOSZ-I/s1600/blog-8.png)

Once initial reference set is collected, dirty cards are reset and JVM starts coping of live objects from Eden and one of survivor spaces into other survivor space. JVM only need to spend time on live objects. Relocating of object also requires updating of references pointing to it.

[](http://3.bp.blogspot.com/-ooT_iuZ694I/TeYIcAQrbgI/AAAAAAAAI0o/x4dy6cb2dyA/s1600/blog-9.png)

While JVM is updating references to relocated object, memory pages get marked again, so we can be sure what on next young GC only dirty pages has references to young space.

[http://3.bp.blogspot.com/-EDGMs03LT80/TpYFZaL3PrI/AAAAAAAAKG8/COdI9A2niNc/s400/blog-9.png](http://3.bp.blogspot.com/-EDGMs03LT80/TpYFZaL3PrI/AAAAAAAAKG8/COdI9A2niNc/s1600/blog-9.png)

Finally we have Eden and one survivor space clean (and ready for allocation) and one survivor space filled with objects.

**Object promotion**

If object is not cleared during young GC it will be eventually copied (promoted) to old space. Promotion occurs in following situations:

* -XX:+AlwaysTenure  makes JVM to promote objects directly to old space instead of survivor  space (survivor spaces are not used in this case).
* once survivor space is full, all remaining live object are relocated directly to old space.
* If object has survived certain number of young space collections, it will be promoted to old space (required number of collections can be adjusted using –XX:MaxTenuringThreshold option and –XX:TargetSurvivorRatio JVM options).

**Allocation of new objects in old space**

It would be beneficial if we could possibly allocate long lived objects directly in old space. Unfortunately there is no way to instruct JVM to do this for particular object. But there are few cases when object can be allocated directly in old space.

* Option -XX:PretenureSizeThreshold=<n> instructs JVM what all objects larger  than <n> bytes should be allocated directly in old space (though if object size fits TLAB, JVM will allocate it in TLAB and thus young space, so you should also limit TLAB size).
* If object is larger than size of Eden space it also will be allocated in old space.

Unlike application objects, system objects are always allocated by JVM directly in permanent space.

**Parallel execution**

Most of task during young space collection can be done in parallel. If there are several CPUs available, JVM can utilize them to compress duration of stop-the-world pause during collection. Number of threads can be configured in HotSpot JVM by –XX:ParallelGCTreads= parameter. By default JVM will choose number of thread by number of available CPU. As expected, serial version of collector will ignore this parameter because it can use only one CPU. Using parallel collection reduces time of stop-the-world pause by factor close to number of physical cores.

**Time of young GC**

Young space collection happens during stop-the-world pause (all non-GC-related threads in JVM are suspended). Wall clock time of stop-the-world pause is very important factor for applications (especially applications requiring fast response time). Parallel execution affects wall clock time of pause but not work effort to be done.

Let’s summarize components of young GC pause. Total pause time can be written as:

*Tyoung = Tstack\_scan + Tcard\_scan + Told\_scan+ Tcopy* ; there *Tyoung* is total time of young GC pause, *Tstack\_scan* is time to scan root in stacks, *Tcard\_scan*is time to scan card table to find dirty pages, *Told\_scan*  is time to scan roots in old space and *Tcopy* is time to copy live objects (1).

Thread stack are usually very small, so major factors affecting time of young GC is *Tcard\_scan* ,*Told\_scan* and *Tcopy*.

Another important parameter is frequency of young GC. Period between young collections is mainly determined by application allocation rate (bytes per second) and size of Eden space.

*Pyoung = Seden / Ralloc*; there *Pyoung*  is period between young GC, *Seden* is size of Eden and *Ralloc* is rate of memory allocations (bytes per second) (2).

*Tstack\_scan*– can be considered application specific constant.

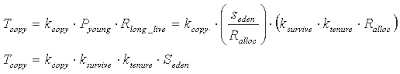
*Tcard\_scan*– is proportional to size of old space. Literally, JVM have to check single byte of table for each 512 bytes of heap (e.g. 8G of heap -> 16m to scan).

*Told\_scan*  – is proportional to number of dirty cards in old space at the moment of young GC. If we assume that references to young space are distributed randomly in old space, then we can provide following formula for time of old space scanning.

[http://1.bp.blogspot.com/-D9UacWPh_x4/TeYKoHMc-EI/AAAAAAAAI0s/3YWMMlhNOjA/s1600/blog-10.png](http://1.bp.blogspot.com/-D9UacWPh_x4/TeYKoHMc-EI/AAAAAAAAI0s/3YWMMlhNOjA/s1600/blog-10.png)

; there *Sold* is size of old space and *D*, *kcard* and*ncard* are coefficients specific for application(3).

*Tcopy*– is proportional to number of live objects in heap. We can approximate it by formula:

[](http://2.bp.blogspot.com/-3YjJgH1l3Gc/TeYK9r5iNPI/AAAAAAAAI0w/nYXtPAliYys/s1600/blog-11.png)

There *kcopy*is effort to copy object, *Rlong\_live* is rate of allocation of long lived objects,

[http://2.bp.blogspot.com/-29LaBbj0Y5s/TeYLeZ7bkjI/AAAAAAAAI00/rHq8rc61h_Q/s1600/blog-12.png](http://2.bp.blogspot.com/-29LaBbj0Y5s/TeYLeZ7bkjI/AAAAAAAAI00/rHq8rc61h_Q/s1600/blog-12.png)

(*ksurvive* usually very small), and *ktenure* is a coefficient to approximate aging of object in young space before tenuring (*ktenure* ≥ 1) (4).

Now we can analyze how various JVM options may affect time and frequency of young GC.

**Size of old space.**

Size of old space is affecting *Tcard\_scan* and *Told\_scan* part of young GC pause time according to formulas above. So we as we are increasing size of old space (read total heap size) time of young GC pauses will grow and it cannot be helped. After certain size of heap (usually 4-8 Gb) time of young collection is dominated by *Tcard\_scan* (technically *Tcopy* can be even greater than *Tcard\_scan* , but it usually can be controlled by tuning of other GC options).

HotSpot JVM options:

-Xmx=<n> -Xms=<n>

**Size of Eden space**

Period between young GC is proportional to size of Eden. *Tcopy* is also proportional to size of eden but in practice *ksurvive* can be so small that for some applications we can forget about Tcopy. Unfortunately time between young GC will also affect coefficient*D* in equation (4). Though dependency between *D* and *Pyoung* is very application specific, increasing *Pyoung* will increase *D* and as a consequence *Tscan\_old*.

HotSpot JVM options:

-XX:NewSize=<n> -XX:MaxNewSize=<n>

**Size of survivor space.**

Size of survivor space puts hard limit of how much objects can stay in young space between collections. Changing size of survivor space may affect *ktenure*(or mat not, e.g. if *ktenure*is already 1).

HotSpot JVM options:

-XX:SurviorRatio=<n>

**Max tenuring threshold**and**target survivor ratio**

These two JVM options also allow artificially adjust *ktenure*.

HotSpot JVM options:

-XX:TargetSurviorRatio=<n>  
‑XX:MaxTenuringThreshold=<n>  
-XX:+AlwaysTenure  
-XX:+NeverTenure

**Pretenuring threshold**

For certain applications using pretenuring threshold could reduce*ksurvive* due to allocation of long lived object directly in old space.

HotSpot JVM options:

-XX:PretenureThreshold=<n>