Generational Garbage Collection
Theory and Best Practices
Overview

- Introduction to Generational Garbage Collection
  - The “nursery” space
    - aka the “young” generation
  - The “tenured” space
    - aka the “old” generation
  - Migrating from other garbage collection modes
Introduction to Generational Garbage Collection

- **Motivation:** Most objects die young

- Most objects are “temporary”
  - Used as part of a calculation or transform
  - Used as part of a business transaction

- Simple example: String concatenation
  - String str = new String("String ");
  - str += "Concatenated!";

  - Results in the creation of 3 objects:
    - String object, containing “String “
    - A StringBuffer, containing “String “, and with “Concatenated!” then appended
    - String object, containing the result: “String Concatenated!”

  - 2 of those 3 objects are no longer required!
Introduction to Generational Garbage Collection

- **Solution:** Garbage collect young objects more frequently

- Create an additional area for young objects (nursery)
  - Create new objects into the additional area
  - Garbage collection focuses on the new area
  - Objects that survive in the new area are moved to the main area

<table>
<thead>
<tr>
<th>Nursery Space</th>
<th>Tenured (old) Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>• New object allocations</td>
<td></td>
</tr>
<tr>
<td>• GC’d frequently</td>
<td></td>
</tr>
<tr>
<td>• Objects surviving from the nursery only</td>
<td></td>
</tr>
<tr>
<td>• GC’d infrequently</td>
<td></td>
</tr>
</tbody>
</table>
The nursery (young) space

- All objects allocated into the nursery space*
  - * unless objects are too large to fit into the nursery

- Garbage collection focusses on the nursery space
  - Garbage collected frequently
  - Garbage collections are fast (short in duration)
  - Most object do not survive a collection
Nursery space implementation

• Nursery is split into 2 spaces:
  • **Allocate space**: used for new allocations and objects that survived previous collections
  • **Survivor space**: used for objects surviving this collection

• Collection causes live objects to be:
  • copied from allocate space to survivor space
  • copied to the tenured space if they have survived sufficient collections

**Note**: spaces are not equal in size – not all objects will survive so Survivor space can be smaller than Allocate space. - “Tilt Ratio”
Nursery space considerations

- Nursery collections **work by copying data** from allocate to survivor
  - Copying of data is a relative expensive (time consuming) task

- Nursery collection **duration is proportional to amount of data copied**
  - Number of objects and size of nursery heap are only secondary factors*

- Only a finite / **fixed amount of data needs to copied**
  - The amount of data being used for any in-flight work (transactions)
  - ie. For a WebContainer with 50 threads, there can only be 50 in-flight transactions at any time

- **The duration of a nursery collection is fixed, and dependent on the size of a set of transactions**
  - Not dependent on the size of the nursery*

*size of the heap does have a small effect, but this is related to traversal of memory only
Optimal size for the nursery space

- Theory shows that the longer the time between nursery collections, the less times on average an object is copied:

<table>
<thead>
<tr>
<th>Time between collections (in transactions)</th>
<th>Average Number of times data is copied</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0.5</td>
<td>1</td>
</tr>
<tr>
<td>X1</td>
<td>0.5</td>
</tr>
<tr>
<td>X2</td>
<td>0.25</td>
</tr>
<tr>
<td>X4</td>
<td>0.125</td>
</tr>
</tbody>
</table>

Time between collections of > 1 transaction ensures data is not copied multiple times.
How large should the nursery be?

- Ideally as large as possible!
  - The larger the nursery, the longer the time between GC cycles
  - The same amount of data is copied regardless
  - Therefore the larger the nursery, the lower the GC overhead

  - Large nurseries also mean very large objects are unlikely to be allocated directly into the tenured space

- Disadvantages of very large nursery spaces:
  - Lots a physical memory and process address space is required
    - Not necessarily possible on 32bit hardware
The tenured (old) space

- Exactly the same as the Java heap in the non-Generational case
  - “New” objects just happen to be copied (tenured) from the nursery space
  - Meaning less garbage to collect, and much fewer GC cycles occurring

- Garbage collected using parallel concurrent mark/sweep with compaction avoidance
  - The same as running “optavgpause” in the non-Generational case
  - Designed to use available CPUs and processing power using GC helper threads:
    - Additional parked thread per available processing unit
    - Wakes up during GC to share workload
    - Configured using -Xgcthreads
  - Reduces GC pause times by marking and sweeping concurrently
    - Reduction in pause times of 90 to 95% vs. non-concurrent GC
Parallel and Concurrent Mark Sweep Collection

Parallel Mark/Sweep (2 CPUs)

+Concurrent MARK
+any idle cpu time

+Concurrent Sweep
+any idle cpu time

M S

Concurrent Kickoff
- Application Thread
- GC Helper Thread
- Application Thread Performing GC
- Application Thread Marking Concurrently
- Application Thread Sweeping Concurrently
Concurrent Mark – hidden object issue

- Higher heap usage…
The “correct” tenured heap size

- GC will adapt heap size to keep occupancy between 40% and 70%
  - Heap occupancy over 70% causes frequent GC cycles
    - Which generally means reduced performance
  - Heap occupancy below 40% means infrequent GC cycles, but cycles longer than they needs to be
    - Which means longer pause times that necessary
    - Which generally means reduced performance

- The maximum heap size setting should therefore be 43% larger than the maximum occupancy of the application
  - Maximum occupancy + 43% means occupancy at 70% of total heap
  - eg. For 70MB occupancy, 100MB Max heap required, which is 70MB + 43% of 70MB
The “correct” tenured heap size
Fixed heap sizes vs. Variable heap sizes

- Should the heap size be “fixed”?  
  - ie. Minimum heap size (-Xms) = Maximum heap size (-Xmx)?

- Each option has advantages and disadvantages  
  - As for most performance tuning, you must select which is right for the particular application

- Variable Heap Sizes  
  - GC will adapt heap size to keep occupancy between 40% and 70%  
  - Expands and Shrinks the Java heap  
  - Allows for scenario where usage varies over time  
  - Where variations would take usage outside of the 40-70% window

- Fixed Heap Sizes  
  - Does not expand or shrink the Java heap
Heap expansion and shrinkage

- Act of heap expansion and shrinkage is relatively “cheap”

- However, a compaction of the Java heap is sometimes required
  - **Expansion**: for some expansions, GC may have already compacted to try to allocate the object before expansion
  - **Shrinkage**: GC may need to compact to move objects from the area of the heap being “shrunk”

- Whilst expansion and shrinkage optimizes heap occupancy, it (usually) does so at the cost of compaction cycles
Conditions for expansion

- Not enough free space available for object allocation after GC has complete
  - Occurs after a compaction cycle
  - Typically occurs where there is fragmentation or during rapid occupancy growth (ie, application startup)

- Heap occupancy is over 70%
  - Compaction unlikely

- More than 13% of time is spent in GC
  - Compaction unlikely
Conditions for shrinkage

- Heap occupancy is under 40%

- And the following is not true:
  - Heap has been recently expanded (last 3 cycles)
  - GC is a result of a System.GC() call

- Compaction occurs if:
  - An object exists in the area being shrunk
  - GC did not shrink on the previous cycle

- Compaction is therefore likely to occur
Introduction to -Xminf and -Xmaxf

- The –Xmaxf and –Xminf settings control the 40% and 70% occupancy bounds
  - **-Xmaxf**: the maximum heap space free before shrinkage (default is 0.6 for 60%)
  - **-Xminf**: the minimum heap space before expansion (default is 0.3 for 70%)

- Can be used to “move” optimum occupancy window if required by the application
  - eg. Lower heap utilization required for more infrequent GC cycles

- Can be used to prevent shrinkage
  - -Xmaxf1.0 would mean shrinkage only when heap is 100% free
  - Would completely remove shrinkage capability
Introduction to -Xmine and -Xmaxe

- The -Xmaxe and -Xmine settings control the bounds of the size of each expansion step
  - **-Xmaxe**: the maximum amount of memory to add to the heap size in the case of expansion (default is unlimited)
  - **-Xmine**: the minimum amount of memory to add to the heap size in the case of expansion (default is 1MB)

- Can be used to reduce/prevent compaction due to expansion
  - Reduce expansions by setting a large -Xmine
Garbage Collection managed heap sizing

- Too Frequent Garbage Collection
- Long Garbage Collection Cycles

Heap Size

Expansion (>= -Xmine)

Heap Occupancy

Memory

Time
Fixed or variable?

- Again, dependent on application

- For “flat” memory usage, use fixed

- For widely varying memory usage, consider variable

- Variable provides more flexibility and ability to avoid OutOfMemoryErrors
  - Some of the disadvantages can be avoided:
    - -Xms set to lowest steady state memory usage prevents expansion at startup
    - -Xmaxf1 will remove shrinkage
    - -Xminf can be used to prevent compaction before expansion
    - -Xmine can be used to reduce expansions
Putting the two together...

- Nursery space and Tenured space are actually allocated as a single chunk of memory
  - Actually possible for the boundary between the nursery and tenured spaces to move:

```
| Nursery Space | Tenured (old) Space |
```

- However this is **not recommended**

- **Recommended mode** is to:
  - **Fix the nursery size** at as large a value as possible
  - **Allow the tenured heap size to vary** according to usage

```
| Nursery Space | Tenured (old) Space |
```

Max Heap Size

Heap Size
Choosing between Generational and Non-Generational modes

- **Rate of Garbage Collection**
  - High rates of object “burn” point to large numbers of transitional objects, and therefore the application may well benefit from the use of gencon

- **Large Object Allocations?**
  - The allocation of very large objects adversely affects gencon unless the nursery is sufficiently large enough. The application may well benefit from optavgpause

- **Large heap usage variations**
  - The optavgpause algorithms are best suited to consistent allocation profiles
  - To a certain extent this applies to gencon as well
  - However, gencon may be better suited

- **Rule of thumb:** if GC overhead is > 10%, you’ve most likely chosen the wrong one
Migrating from other GC modes

- Other garbage collection modes do not have a nursery heap
  - Maximum heap size (-Xmx) is tenured heap only

- When migrating to generational it can be required to increase the maximum heap size
  - Non-generational: -Xmx1024M gives 1G tenured heap
  - Generational: -Xmx1024M gives 64M nursery and 960M tenured

- As some of the nursery is survivor space, there is a net reduction in available Java heap
  - “Tilt Ratio” determines how much is “lost”

- Recommended **starting point is to set the tenured heap to the previous maximum heap size:**
  - ie. -Xmos = -Xms and -Xmox = -Xmx

- And **allocate the nursery and an additional heap space**

- This means there is a **net increase in memory usage when moving to generational**
Example of Generational vs Non-Generational
Monitoring GC activity

- Use of Verbose GC logging
  - only data that is required for GC performance tuning
  - Graph Verbose GC output using GC and Memory Visualizer (GCMV) from ISA

- Activated using command line options

  -verbose:gc
  -Xverbosegclog:[DIR_PATH][FILE_NAME]
  -Xverbosegclog:[DIR_PATH][FILE_NAME],X,Y
  – where:

    [DIR_PATH] is the directory where the file should be written
    [FILE_NAME] is the name of the file to write the logging to
    X is the number of files to
    Y is the number of GC cycles a file should contain

- Performance Cost:
  – (very) basic testing shows a 1% overhead for GC duration of 200ms
  – eg. if application GC overhead is 5%, it would become 5.05%
Gencon could handle a higher “rate of garbage collection”

Gencon had a smaller percentage of time in garbage collection

Gencon had a shorter maximum pause time
Rate of garbage collection

- Gencon provides less frequent long Garbage Collection cycles
- Gencon provides a shorter longest Garbage Collection cycle
Questions?