Automated garbage collection

Kain Alden
Department of computing and mathematics
University Of Derby
Derby ,England

Automated garbage collection makes programming easier for developers and can lead to less error prone code. The negative impact this automation has on performance can be largely reduced with an effective collection system.

I. Introduction

Automated garbage collection vs manual management of memory is a topic that has been much debated in the past. Those in favor of automated collection argue that it aids the developer be removing an entire category of bugs that could appear whilst also speeding up development time. Meanwhile the main argument against the automation is the performance issues caused that could be avoided should the memory be managed manually. With the increase in mobile devices it is important that automated garbage collection keeps improving to be effective for use on these devices.

II. What is Garbage collection

Automated garbage collection systems will keep track of all objects that are created by a program and use one of many algorithms to work out when that object is no longer going to be used. When the object is no longer in use the memory that it was stored in will be released back to the operating system. The alternative to this is that the programmer will be left to control the memory allocation. They will need to keep track of which memory is still in use and remember to release this memory once it is no longer needed. (Plumbr, 2017).

III. advantages

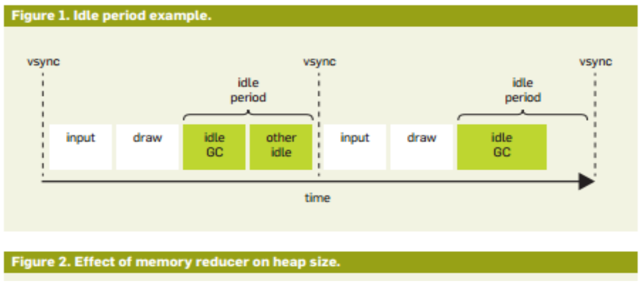
Automating the management of memory has several advantages over manual memory management. The first positive is the reduced development time. As developers, do not need to concern themselves with how memory is allocated they will be able to focus their time on the rest of their code. An additional positive of developers not needing to manage memory is that there will be less opportunity for bugs to occur. (Plumbr, 2017). Some of the most common bugs that are caused by poorly implemented memory management are double frees. This is where a piece of memory that has already been freed is freed for a second time. If the memory has been reallocated between the free commands, you would then be freeing a piece of memory that is still in use. (Erickson 2002). Another issue would be memory not being freed, this would cause a memory leak that would eventually slow the program down considerably. These types of bugs can have far reaching consequences and seriously harm the effectiveness of software.

IV. Improving Proformance

Whilst the advantages to automated garbage collection are obvious it is harder to understand the negative points. Simply saying that there is reduced performance does not give enough clarity as to how the problem appears and more importantly how it can be solved. A detailed look at the overhead caused by this automation and how these issues are being tackled will help to identify the best scenarios to use each type of memory management.

A. Time stalls

The most mentioned performance issue related to garbage collection is a time stall. A time stall is where there will be a momentary pause in the program. This will be caused by the garbage collection being executed. When running, the collection will prevent the application from running in real time. As these pauses are only for a short period in many applications they will not be an issue. However, in real time or interactive applications such as games they might cause a reduction in frame rate and smoothness. (Chang,2002)
Steps have been made to prevent stalls from happening as frequently. One way is to change how the collection functions slightly. Instead of doing larger infrequent clean-ups the garbage collection will instead do smaller more frequent clean-ups of memory. These are not only less impactful on the running of the code but also are easier to run simultaneously with the code. As running the smaller collections will not take as long the impact on performance is felt less by the user. (Biron ,2007). It is important to keep in mind that whilst the impact on users is reduced using smaller more frequent garbage collection it will lead to a higher overhead as not as much clean-up can be completed per cycle.
Another way to minimize the impact of time stalls is to set the garbage collection to run when the program is not utilizing as much computational power.
An example of a program that will complete collection when the program is idle is google chrome. The image below shows when the clean-up takes place. It is only allowed to happen after one frame is rendered and before the next frame needs to start being rendered. This prevents the garbage collection from impacting the user experience. (DEGENBAEV ,2016)

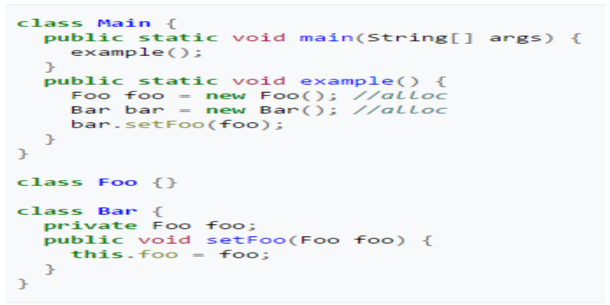


Source; (DEGENBAEV ,2016)

Even if the problem of stalling can be resolved. Automated garbage collection still requires additional computational and memory resources. This is inevitable as there is more code that needs to be run. In most cases this should not affect the program and will not be a consideration when choosing which language to use. However, some applications such as server side code will be running on relatively low power machines and will need to be optimized to a high degree. If this is the case any amount of overhead will become a significant issue. Modern garbage collections systems have features that will help to reduce the impact they have on performance as much as possible.

B. Escape Analysis

One feature that garbage collection will use to reduce its impact at run time is escape analysis. The compiler will check each object and see if allocated inside an object and not accessible out of it. If it is the allocation can be done onto the stack instead of the heap. This will reduce the amount of overhead needed at runtime. (Gupta, n/a). In the below example foo is allocated and only accessible inside bar. This means that it can be allocate on the stack instead of the heap. Similarly, bar is only accessible inside example so can also be allocated on the stack. The allocations will be freed when the code exits from function. Having prior knowledge of when an allocation can be released will reduce the impact of run time garbage collection



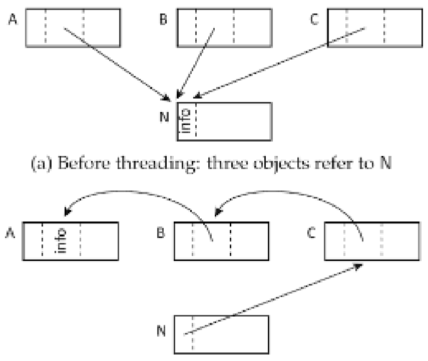
```
class Main {
  public static void main(String[] args) {
    example();
  }
  public static void example() {
    Foo foo = new Foo(); //alloc
    Bar bar = new Bar(); //alloc
    bar.setFoo(foo);
  }
}

class Foo {}

class Bar {
  private Foo foo;
  public void setFoo(Foo foo) {
    this.foo = foo;
  }
}
```

Source; (Wikipedia, n/a).

C. Heap Compression

As well as reducing the need for computational power at run time garbage collection has ways to reduce the amount of memory being used by the program. This is done by ensuring data on the heap remains organized. Requested memory will be allocated in one continues block of the requested size. With memory being allocated and freed frequently it is possible for gaps of free memory to appear on the heap. If these gaps are small they will likely not be reallocated. When a garbage collection cycle takes place, the heap will be compressed. (Abuaiadh ,2004). Data will be rearranged to defragment the heap meaning that the previously unused regions are now grouped into large enough sections to be useable. Ensuring all the free memory is accessible will prevent the program from slowing and increase performance.

One method of doing this is the Jonkers algorithm. When implemented the algorithm will work through the heap threading all the references. It will use theses threaded references to reallocate the stored data to one end of the heap. The below example shows how all the references to n would be threaded. The threading would reverse them so that all the references are accessible from n. These threaded references would be used to organize the heap. (Jones ,2016).



Source; . (Jones ,2016).

V. CONCLUSION

In conclusion, automated garbage collection allows for increased development speed and leads to less error prone code. However in order to become a more viable choice when developing for mobile systems it is important to minimize the impact that it can have on performance with modern techniques this impact can be negated to the point where garbage collection is comparable in terms of performance to manual memory management. Tools such as escapes analysis and heap compaction can reduce the amount of overhead needed to run garbage collection significantly. Meanwhile the impact of time stalls can be minimized by synchronizing the collection cycles with the programming.

These features along with the continual advancement in affordable computing power means that garbage collection is becoming ever more viable for real time and mobile applications.

References

i. Abuaiadh ,D. (2004). An Efficient Parallel Heap Compaction Algorithm. *OOPSLA '04 Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 1 (1), 224 - 236 .

ii. Biron ,B. (2007). *Real-time Java, Part 4: Real-time garbage collection.* Available: http://www.ibm.com/developerworks/java/library/j-rtj4/index.html. Last accessed 17/02/17.

iii. Chang, J.M. (2002). *A performance comparison between stop-the-world and multithreaded concurrent generational garbage collection for Java.* Available: http://ieeexplore.ieee.org.ezproxy.derby.ac.uk/document/995163/. Last accessed 17/02/17.

iv. DEGENBAEV ,U. (2016). Idle-Time Garbage Collection Scheduling. *Communications of the ACM*. 59 (10), P34-39.

v. Erickson,C. (2002). *Memory Leak Detection in Embedded Systems.* Available: http://www.linuxjournal.com/article/6059. Last accessed 17/02/17

vi. Gupta ,M. (n/a). *Escape Analysis for Java.* Available: http://www.cc.gatech.edu/~harrold/6340/cs6340_fall2009/Readings/choi99escape.pdf. Last accessed 17/02/17.

vii. Hertz ,M. (2016). *Quantifying the Performance of Garbage Collection vs. Explicit Memory Management.* Available: http://people.cs.umass.edu/~emery/pubs/gcvsmalloc.pdf. Last accessed 17/02/17.

viii. Jones ,R (2016). *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Unknown: CRC Press. p36-38

ix. Meurer ,B . (n/a). *Fast garbage compaction with interior pointers.* Available: http://benediktmeurer.de/files/fast-garbage-compaction-with-interior-pointers.pdf. Last accessed 17/02/17.

x. Oracle. (2016). *Java HotSpot™ Virtual Machine Performance Enhancements.* Available: http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#escapeAnalysis. Last accessed 17/02/17.

xi. plumbr. (2017). *What Is Garbage Collection?.* Available: https://plumbr.eu/handbook/what-is-garbage-collection. Last accessed 17/02/17.

Images sourced from

xii. DEGENBAEV ,U. (2016). Idle-Time GarbageCollection Scheduling. *Communications of the ACM*. 59 (10), P34-39.

xiii. Jones ,R (2016). *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Unknown: CRC Press. p36-38

xiv. Wikipedia. (n/a). *Escape analysis.* Available: https://en.wikipedia.org/wiki/Escape_analysis. Last accessed 2017.