Mark-sweep compared to Generational GC

Arvid Lorén

November 2022

1 Introduction

Whenever a program runs on your computer, that program needs to store data. Where does it store it's data? In memory. Computer memory comes in many forms and sizes, but this essay will be focusing on the heap. The heap is a big memory space where many computer programs can borrow chunks of memory to store their data. This act of borrowing memory is called allocating. When a program does not need a piece of data anymore, the computer needs to retrieve that piece of memory so that it can be used by other processes. Otherwise, the computer would soon run out of memory. This act of retrieving memory is called freeing.

In certain programming languages, for example C, this task of allocating and freeing has to be manually maintained. Fortunately, many modern languages have done away with that by implementing automatic memory management, also called garbage collection. This essay will focus on two methods for garbage collection, or GC for short, used in Java.

1.1 Java and Objects

The programming language Java is an object oriented language. Meaning that a program consists of many objects that interact with each other. It is these objects that the GC is concerned with.

1.2 Garbage Collection

A garbage collector is a program which has the purpose of freeing all unreachable objects of some other running program. An object is considered to be garbage if the program is unable to access it in any way. The tricky part is how to go about finding garbage. There are different solutions to this. The purpose of this essay is to examine two of them more closely, consider their strengths and weaknesses and compare them using the measurements throughput, latency and jitter.

2 Mark-sweep

2.1 How it works

The Mark-sweep algorithm is a two-step process. The running program will be suspended, and the process performed. Because of this, it is known as a stop the world-GC. The two steps are the mark phase and the sweep phase.

2.1.1 Mark phase

Every object has a mark bit, which is set to 0 upon creation. In the mark phase, the GC will start at the root of the program and traverse through all objects that are reachable. When an object is visited by the GC, its mark bit is set to 1 and it is then considered to be live. When the mark phase is done, all reachable objects are live and the rest are considered garbage.

2.1.2 Sweep phase

In the sweep phase, the GC traverses heap space used by the program. When an object is encountered, the GC looks at the mark bit. If it is 0, the memory can be freed. If it is 1, the GC resets it to 0 in preparation for the next collection. When the sweep phase is done, all garbage should be collected and all reachable objects are living happily.

2.2 Strengths

This method takes care of something called "cyclic references", which is when a certain amount of objects point to each other in a cyclic manner. The case of cyclic references is not taken care of by the common GC-method reference counting.

Because every object in the cycle is pointed to by some other object, they will never be considered garbage by a reference counting GC. The Mark-sweep algorithm solves this, as none of the object in the cycle will be marked as live in the mark phase, and thus collected in the sweep phase.

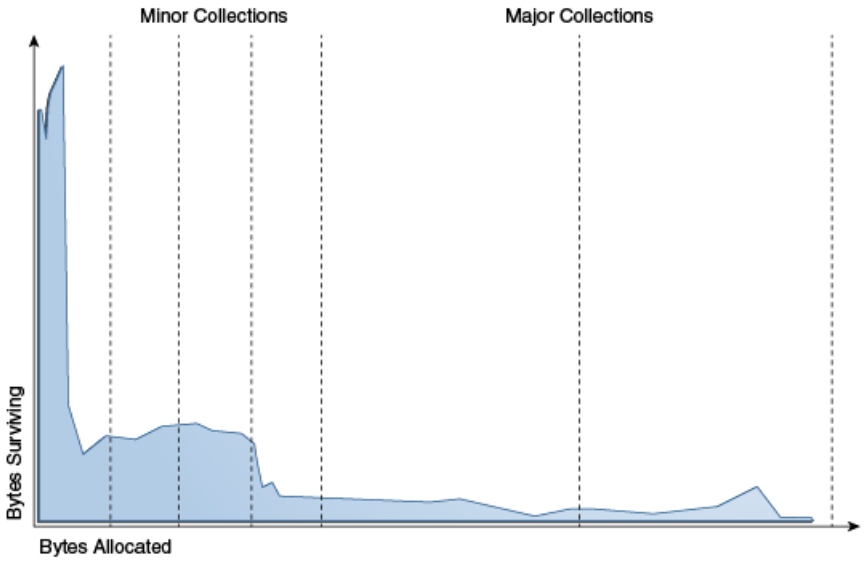
2.3 Weaknesses

As mentioned above, the program will be completely suspended while the Mark-sweep GC runs. It would naturally be more efficient to keep the program running all the time. New garbage collectors have been developed to offer solutions to this problem. For example Concurrent GC, which performs garbage collection on a different thread than the program is running on. This means the the running program never has to stop completely. Basic Mark-sweep is not concurrent, and having to pause the program is therefore it's biggest flaw.

Another problem is fragmentation. This is when allocated chunks of memory become scattered on the heap, leaving scattered chunks of free memory. Why is this a problem? Because when the computer is asked to allocate memory of a certain size, it will search for a *continuous* free heap chunk of that size. This is also solved by more complicated garbage collectors, for example compacting GC's.

3 Generational Garbage Collection

The general idea of Generational GC is that collection time can be optimized based on some observations. These observations are known as the "weak generational hypothesis", which states that most objects die young. The theory is empirically observed and holds for most programs. Figure 1 shows the typical distribution for lifetimes of objects. This means that at any given point, young objects will often be garbage and older objects will not. Keep this in mind while reading the next section!



([1])

3.1 How it works

The idea is to separate the programs heap space into two generations: the old and the young. The young generation consists of three different areas: Eden and two survivor spaces. One of the survivor spaces will be active and the other one passive. When a new object is created, it is placed in Eden. Eden eventually fills up and needs to be collected.

According to the weak generational hypothesis, most objects will be garbage at this point and the collection will be very fruitful. The objects that are not collected are transferred to the active survivor space. At every collection of the young generation, the objects in the active survivor space are also collected. The objects that survive this are moved to the passive, which now becomes the active one.

When the survivor space fills up, it is time for a collection of both the young and the old generations. If such a collection was run every time, the old generation would probably contain much fewer garbage objects compared to the amount of live ones. It is because of this generational GC is more efficient.

3.2 Pros

We are separating young and old objects, and choose how often to collect each are based on the "weak generational hypothesis". This is for most programs more efficient than simply traversing the whole heap to check for garbage every time, since we are proportionally collecting more garbage in relation to heap space searched.

3.3 Cons

This method shares a weakness with Mark-sweep, in that the program on which the GC is being run has to be paused during collection.

In the cases that the weak generational hypothesis doesn't hold, generational GC will be less efficient than traversing the whole heap every time.

4 Comparison

Mark-sweep and Generational GC is a bit tricky to compare directly, since Mark-sweep is an algorithm to collect garbage and Generational GC is a way to make the collection more efficient. That means that a GC can make use of both. When collection is due in the different generations, Mark-sweep can be used.

4.1 Throughput

Throughput is a measurement of how much work is done over time. High throughput is desired, since it means that we get a lot of work done. For a GC, that would be the overall performance. Stop the world-GC's generally have high throughput. Since we are not collecting any garbage while the program is running, there is no overhead.

Meaning, the program has one less thing to do, and can execute other tasks more efficiently. Compared to GC's that are working simultaneously as the program, like reference counting, executing the program tasks would take longer since GC also has to be run.

4.2 Latency

Latency is a measurement for how much time a certain process takes to finish. Low latency is desired, since that means the process is fast. For garbage collection, we are interested in collection duration. This is a bigger issue for stop the world-GCs. GCs that run alongside the program collects garbage almost instantaneously, whereas stop the world-GCs have to pause the program and then performs the collection.

The latency of Mark-sweep is proportional to the amount of live objects in the program. If there are a lot of live objects, the mark sweep algorithm will take longer. The latency for Generational GC varies. If we perform collection only on the young generation, the time it takes will be much shorter than if we also collect the old generation.

4.3 Jitter

Jitter is measurement of variation in latency. For example, if a program responds quickly or slowly all the time, it has low jitter. On the other hand, if it responds quickly sometimes and slowly sometimes, the variation in latency is high and therefore, so is the jitter. The Mark-sweep algorithm could potentially have high jitter, if the amount of live objects differ vastly between collections. But if GC is run at steady intervals, it should be fairly consistent. Generational GC, on the other hand, will almost definitely have high jitter. The cause of this is explained in the previous section.

5 Conclusion

My conclusion is that Mark-sweep and Generational GC deals with different issues and are in practice very compatible with each other. Furthermore, they have similar properties when compared with respect to throughput, latency and jitter.

Bibliography

Cited references

[1] *3 Generations*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html.

Uncited references

[2] *Garbage Collection Impact on Application Performance*. URL: https://www.azul.com/blog/garbage-collection-application-performance-impact/.

[3] *Generational Garbage Collection and Heap Memory*. URL: https://incusdata.com/blog/generational-garbage-collection-and-heap-memory.

[4] *Generations of Garbage Collection*. URL: https://deepak4blogger.blogspot.com/2017/09/generations-improving-performance.html.

[5] *Mark-and-Sweep: Garbage Collection Algorithm*. URL: https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/.