ATOMIC OBJECT

September 3, 2014 by: Ken Fox

21 Comments

# Visualizing Garbage Collection Algorithms

Most developers take automatic garbage collection for granted. It's just another amazing feature provided by our language run-times to make our jobs easier.

But if you try to peek inside a modern garbage collector, it's very difficult to see how they actually work. There are thousands of implementation details that will confuse you unless you already have a good understanding of what it's trying to do and how they can go fantastically wrong.

I've built a toy with five different garbage collection algorithms. Small animations were created from the run-time behavior. You can find larger animations and the code to create them at github.com/kenfox/gc-viz. It surprised me how much a simple animation reveals about these important algorithms.

## Cleanup At The End: aka No GC


The simplest possible way of cleaning up garbage is to just wait until a task is done and dispose of everything at once. This is a surprisingly useful technique, especially if you have a way of breaking up a task into pieces. The Apache web server, for example, creates a small pool of memory per request and throws the entire pool away when the request completes.

The small animation to the right represents a running program. The entire image represents the program's memory. Memory starts out colored black, which means it isn't used. Areas that flash bright green or yellow are memory reads or writes. The color decays over time so you can see how memory was used, but also see current activity. If you watch carefully, you can see patterns emerge where the program begins to ignore some memory. Those areas have become garbage — they are not used and not reachable by the program. Everything else that isn't garbage is "live".

The program easily fits in memory without needing to worry about cleaning up garbage while the program is running. I'm going to stick with this simple program for the rest of the examples.

## Reference Counting Collector


Another simple solution is to keep a count of how many times you are using a resource (an object in memory, in this case) and dispose of it when the count drops to zero. This is the most common technique that developers use when they add garbage collection to an existing system —it's the only garbage collector that easily integrates with other resource managers and existing code bases. Apple learned this lesson after releasing a mark-sweep collector for Objective-C. It caused so many problems that they retracted the feature and replaced it with an automated reference counting collector that works much better with existing code.

The animation shows the same program as above, but this time it will try to dispose of garbage by keeping a reference count on each object in memory. The red flashes indicate reference counting activity. A very useful property of reference counting is that garbage is detected as soon as possible — you can sometimes see a flash of red immediately followed by the area turning black.

Unfortunately reference counting has a lot of problems. Worst of all, it can't handle cyclic structures. These are very common — anything with a parent or reverse reference creates a cycle which will leak memory. Reference counting also has very high overhead — you can see in the animation that red flashes are constantly happening even when memory use is not increasing. Arithmetic is fast on a modern CPU, but memory is slow, and the counters are being loaded and saved to memory often. All these counter updates also make it difficult to have read-only or thread-safe data.

Reference counting is an amortized algorithm (the overhead is spread over the run-time of the program), but it's an accidentally amortized algorithm that can't guarantee response times. For example, say a program is working with a very large tree structure. The last piece of the program that uses the tree will trigger the disposal of the entire tree, which Murphy will guarantee happens when the user least desires

the delay. None of the other algorithms here are amortized either though, so accidentally amortized may be a feature depending on your data. (All of these algorithms do have concurrent or partially-concurrent variations, but those are beyond the capabilities of my toy program to demonstrate.)

## Mark-Sweep Collector


Mark-sweep eliminates some of the problems of reference count. It can easily handle cyclic structures and it has lower overhead since it doesn't need to maintain counts.

It gives up being able to detect garbage immediately. You can see that in the animation where there's a period of activity without any red flashes, then suddenly a bunch of red flashes indicate where it is marking live objects. After marking is finished, it sweeps over all of memory and disposes of garbage. You can see that in the animation too — several areas turn black all at once instead of more spread out over time in the reference counting approach.

Mark-sweep requires more implementation consistency than reference counting and is more difficult to retrofit into existing systems. The mark phase requires being able to traverse all live data, even data encapsulated within an object. If an object doesn't provide traversal, it's probably too risky to attempt to retrofit mark-sweep into the code. The other weakness of mark-sweep is the fact the sweep phase must sweep over all of memory to find garbage. For systems that do not generate much garbage, this is not an issue, but modern functional programming style generates enormous amounts of garbage.

## Mark-Compact Collector


One thing you may have noticed in the previous animations is that objects never move. Once an object is allocated in memory, it stays in the same place even if memory turns into a fragmented sea of islands surrounded by black. The next two algorithms change that, but with completely different approaches.

Mark-compact disposes of memory, not by just marking it free, but by moving objects down into the free space. Objects always stay in the same memory order — an object

allocated before another object will always be lower in memory — but gaps caused by disposed objects will be closed up by objects moving down.

The crazy idea of moving objects means that new objects can always just be created at the end of used memory. This is called a "bump" allocator and is as cheap as stack allocation, but without the limitations of stack size. Some systems using bump allocators don't even use call stacks for data storage, they just allocate call frames in the heap and treat them like any other object.

Another benefit, sometimes more theory than practice, is that when objects are compacted like this, programs have better memory access patterns that are friendly to modern hardware memory caches. It's far from certain you will see this benefit, though — the memory allocators used in reference counting and mark-sweep are complex, but also very well debugged and very efficient.

Mark-compact is a complex algorithm requiring several passes over all allocated objects. In the animation you can see the red flashes of live object marking followed by lots of reads and writes as destinations are computed, objects are moved and finally references are fixed to point to where objects have moved. The main benefit of all this complexity is operating under extremely low memory overhead. Oracle's Hotspot JVM uses several different garbage collection algorithms. The tenured object space uses mark-compact.

## Copying Collector


The last algorithm I've animated is the foundation of most high-performance garbage collection systems. It's a moving collector like mark-compact, but it's incredibly simple. It uses two memory spaces and simply copies live objects back and forth between them. In practice, there are more than two spaces and the spaces are used for different generations of objects. New objects are created in one space, get copied to another space if they survive, and finally copied to a tenured space if they are very long-lived. If you hear a garbage collector described as generational or ephemeral, it's usually a multi-space copy collector.

Other than simplicity and flexibility, the main advantage of this algorithm is that it only spends time on live objects. There is no separate mark phase that must be later

swept or compacted. Objects are immediately copied during live object traversal, and object references are patched up by following a broken-heart reference where the object used to be.

In the animation, you can see there are several collections where almost all the data is copied from one space to the other. This is a terrible situation for this algorithm and shows one of the reasons why people talk about tuning garbage collectors. If you can size your memory and tune your allocations so that most objects are dead when the collection begins, then you get a fantastic combination of safe functional programming style and high performance.

## Related Posts

### Hexagonal Architecture in Action

by Will Pleasant-Ryan

### When & How to Solve Problems with Genetic Algorithms

by Shawn Anderson

### Time and Relative Distance in Source (Code)

by Will Pleasant-Ryan