Daniel Espinosa
Eric Seamons
CS 330

Garbage Collection

Mark & Sweep and Large Objects

The reason as to why Mark & Sweep is preferable for large-object spaces is because it gives preference to space over time. Unlike other strategies for garbage collection, mark and sweep doesn’t need to copy any data at all, and because of this it can use the full size of the heap to allocate data. This in turn results in having to do garbage collection less frequently. Garbage collection adds considerable overhead, and if we are working with a very large object space then this means we are going to have to do garbage collection a lot more frequently. Therefore, delaying this operation as much as possible is very desirable and will in the end save more time than using some other more efficient garbage collection algorithm.

Type Tag Placement

Checking the type tag is an operation that happens very frequently. Therefore, we want to make this operation as fast as possible without introducing any overhead. Putting this tag at the last address would indeed be very problematic, especially when we have data of different sizes. If all the data had the same size, we would only need to add the offset to access the data (which would add some small overhead), but the real nightmare would come if the data had different sizes. We would have to create a mechanism to retrieve the tag every time we accessed an object because the tag would be in a different place for each object.

Intergenerational Pointers

A generational garbage collector handles references from new to old naturally because it uses the life of the object to pass it from one generation to another. The object ages by surviving each successive garbage collection cycle; the older the generation the less frequent garbage collection is done. The reason for this is that statistically objects die at a young age, so the older they get the less likely they are to be collected and the more likely they are to survive. Therefore, garbage collection is performed less frequently on older generations to prevent repetitive marking. Because of this, generational garbage collectors must keep track of variable mutation, since this signals when the object was “born” and it begins to age from that time onward. The Generational Collector doesn’t need to track the vector-set! operation because the vector is still pointing to the same object as before, the contents of the object has just changed. The generational Collector needs to keep track of the the set! operation because the value the variable held was changed.

Application

Knowing about garbage collection certainly can help us programmers have better coding practices. If anything, this particular assignment has made it abundantly clear that garbage collection comes at a cost, and sometimes it can be quite costly. Therefore, we must try to make use of it a little as possible, and this basically translates to minimizing the amount of objects that we instantiate in the heap. There are many instances in which this is unavoidable, but seeking to use the heap as little as possible and only when absolutely necessary will make it so that code is more efficient.

This is particularly true when we are coding for real time applications. In these scenarios the type of garbage collection we are using can definitely break time constraints (because of pause of execution to perform garbage collection). Different tasks require different types of garbage collection. When making online games, it is best to have garbage collection that frees memory little bits at a time. This prevents the game from freezing every once and awhile. Gamers don't tolerate that in game play, so how garbage collection is managed is very important. Therefore, we must choose carefully a garbage collector that will not interfere with our time constraints, or consider whether we should even use one at all (particularly if lives are at stake).