Generational Garbage Collection, and Weak References

sk and dbtucker

November 22, 2000

In the previous lecture, we saw how to do garbage collection using the stop-and-copy algorithm, which divides memory into two semispaces. There are three benefits of this algorithm:

1. The amount of memory you touch is proportional to the amount of live data, not the total memory.
2. The copying procedure compacts the live data in the new semispace.
3. Allocation is easy—the free memory is always one large contiguous chunk.

However, the algorithm also has several negatives:

1. You can only use half of your memory.
2. You have to copy all of the live data.
3. You copy persistent (long-living) data many times.

Our next GC strategy relies on the following observation: 80-90% of objects don't persist beyond a short period (one million instructions). In this amount of time, you can typically allocate at most one megabyte of data. So, if we make each semispace two megabytes large, by the time one fills up, most of the data will be dead. The advantage is that now the GC will be fast; on the other hand, we are not using a lot of memory.

Generational garbage collectors

The idea in generational garbage collection is to divide memory into a number of generations, say four (see Figure 1). We could make each generation a semispace, but then we would still be wasting half of our memory. Remember, 80-90% of objects are very short-lived; conversely, the rest of the objects are very persistent (they exist for a very long time). Also, the persistent objects are often large, so we don't want to copy them if at all possible.

Here's our strategy for generational collection. We allocate objects into the 0th generation, whose semispaces can each contain two megabytes of data, as

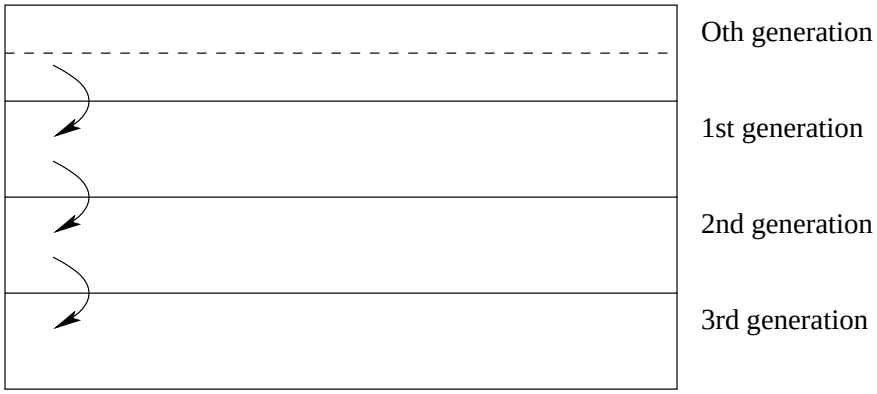

Figure 1: The heap in a generational GC

we discussed above. If an object has survived a few (two or three) GC's, we move it to the 1st generation. When the 1st generation fills up, we garbage collect those objects into the 2nd generation, and similarly we GC the 2nd into the 3rd. For collecting the 3rd generation, we'll have some special scheme that compacts the data.

What are the properties of this algorithm?

- We still copy all live objects in the 0th generation.
- We copy persistent objects very rarely, since they live in the 1st generation and higher.
- We are now using the vast majority of our available memory—the only wasted space is one of the semispaces in the 0th generation, and that's only two megabytes large.
- We usually have short GC times, since most of the time we only need to run stop-and-copy on the 0th generation.

There's one bit of the algorithm we haven't explained yet. When running stop-and-copy on the 0th generation, the GC doesn't follow pointers that go outside the 0th generation, because doing so could cause it to walk the entire heap. This decision is reasonable because in most cases, new objects refer to old objects, and old ones don't refer to new ones. For example, when you create a new list object, the list contains references to elements that are already on the heap.

How can you violate this assumption? The answer is mutation. Specifically, you can make an old object refer to a new one by using a function such as *set-car!*, which mutates a field of an object. (Note that using **set!** only changes a reference to an object, not part of an object.) If we use *set-car!* in a program, our current generational GC is broken—it might not mark live data in the 0th generation.

What can we do to overcome this problem? Here are some proposals:

1. Don't allow mutation.
2. When we perform a *set-car!* on an object, move that object back to the 0th generation. This approach has three drawbacks:
   (a) There was a good reason for the object being in a higher generation—it was persistent and possibly large.
   (b) By moving the object, you create a gap in memory at its old location.
   (c) You make the *set-car!* operation really expensive.
3. Keep a list of mutationally induced roots.
4. The actual solution used by generational GCs: set the dirty bit on that object's page. When collecting the 0th generation, you consider any dirty pages as part of the root set.

Weak pointers

Suppose you have a large data structure, say a binary tree. In order to reduce access time to popular subtrees, you keep references to those subtrees in a cache. When you no longer have any use the tree and lose the reference to the root element, you expect the GC to reclaim the memory occupied by the tree. However, the cached references to subtrees could still be live, and most of the tree could still be reachable through these references. Thus, the GC can't reclaim this memory.

To solve this problem, some languages have *weak pointers*. A weak pointer is a reference to an object that the GC does not follow during DFS. A weak pointer still must be updated like any other pointer when an object is moved. In our example, we should make the cached references be weak pointers to the large tree; then, the GC will not traverse them during DFS, and the tree will no longer be reachable. In order to make weak pointers safe, when you dereference a weak pointer, you must check whether the object still exists.