Stop-and-Copy Garbage Collection

sk and dbtucker

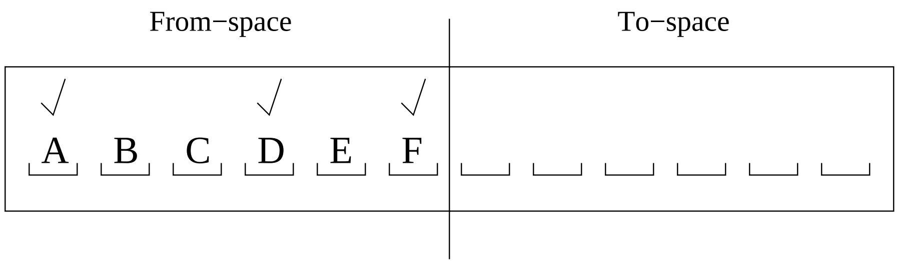
November 20, 2000

In the previous lecture, we studied an algorithm for garbage collection called mark-and-sweep. The good thing about this algorithm is that it works—it frees all non-reachable data structures, even those containing cyclic references. However, there are several drawbacks to mark-and-sweep:

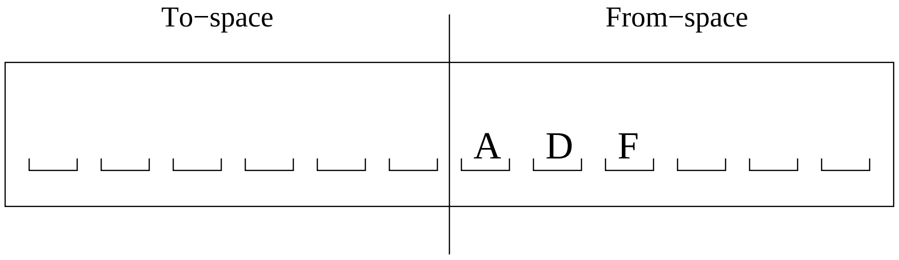
1. You touch every spot in memory during the sweep phase of the algorithm. (In the mark phase, we only touch "live" memory, a cost which any reachability-based scheme will incur.) Thus, the sweep phase can take a very long time.
2. The non-allocated memory becomes fragmented, so you may have enough total free memory for a particular object, but no contiguous chunk large enough to store it.
3. Memory allocation can be time-consuming, because you have to search for a contiguous chunk that can hold the new object.

Stop & Copy

A different method, called *Stop & Copy*, was proposed by Fenichel and Yochelson in 1970. This algorithm gets rid of the sweep phase, and doesn't fragment free memory. The idea is to split all of memory into two pieces, one called the *From-space* and the other called the *To-space*. Say we have twelve cells of memory:



The first six cells are filled with data: the objects A through F. At this point in the program, only A, D, and F are reachable. When we run DFS to find live objects, instead of marking each object, we copy the object to the next free cell of the To-space. Once the DFS has completed, the To-space will consist of all the live objects, and they will be contiguous in memory. Now we swap the names of the two semispaces:



How does this algorithm avoid doing a sweep phase? Rather than trying to determine which objects are unreachable, which requires a sweep, the algorithm just copies out live objects to the To-space, and then everything in the From-space can be treated as dead. Notice also that the largest contiguous free space in the new From-space is three cells, which is equal to the total free memory.

This algorithm is also known as *semispace swapping.* Here's the high-level picture:

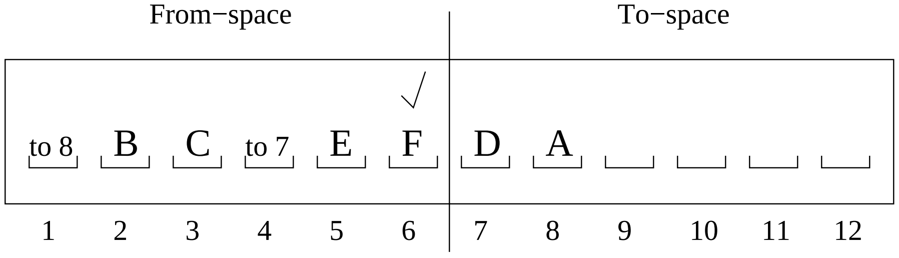
1. Allocate new memory in the next available slot in the From-space.
2. When you run out of free memory in the From-space, stop everything and do garbage collection:
    (a) run DFS, and when you visit an object in the From-space, copy it to the To-space
    (b) swap the names of the From-space and To-space

There's one crucial detail of this algorithm we haven't yet addressed. When you move an object to a different location in memory, you need to update any references to this object. Where do we find these references? First, the elements of the root set (that is, the variables on the stack) will need to be updated. Second, an object on the heap can have a reference to another object. When you encounter a reference to an object during DFS, there are two possibilities:

1. The referenced object has not been visited yet. In this case, you run DFS on any children referenced by the object. When DFS returns, you copy the object to the To-space, and place a forwarding pointer in its cell in the From-space. For example, if you copy the object to cell 42 in the To-space, you put a note in its old From-space location saying "moved to cell 42". Finally, you update the reference to point to the the new memory location.
2. The referenced object has already been visited. In this case, you just update the reference to be the forwarding pointer left in place of the object.

Since we only copy an object after we have copied all of its children, we won't have any objects in the To-space that refer to objects in the From-space.

Let's look at a picture. Say object A has a reference to object D. The heap would look like this after the objects A and D have been visited:



Now if object F also has a reference to D, the algorithm will see that D has already been copied to the To-space, and update F's reference to be cell 7.

Here's another approach: instead of leaving forwarding pointers, we add a level of indirection. So, a reference would index into some table, and the value in the table would be the memory location of the object. Then, when you move an object, you simply update the table. The problem with this approach is that every object reference now requires *two* dereferences.

Look at today's handout. The code on pages 1 and 2 is an implementation of mark-and-sweep which uses registers =*tmp1*=, =*tmp2*=, and =*tmp3*= to hold temporary variables (instead of using **let**). We add these three registers to the root set for the mark phase. The code on pages 3 and 4 is an implementation of the stop-and-copy algorithm. However, there's a bug; if the GC is invoked when *cons/prim* tries to acquire three bytes, the references to the two components (*fst-val* and *rst-val*) of the *cons* pair might not be in the root set. The code on pages 5 and 6 fixes this bug by passing the components in registers =*tmp1*= and =*tmp2*=, which we know will be marked.

Locality of reference

Locality of reference is the notion that if we access something, chances are high that we will soon access something close. There are two types of locality: spatial and temporal. Spatial locality of reference says that we are likely to access objects that are close in memory. Temporal locality says that if we access an object, we are likely to access it again in the near future.

Almost all programs exhibit a high degree of locality of reference. The stop-and-copy algorithm *causes* greater spatial locality by compaction—it places related objects close to each other in memory. Computers exploit locality of reference by using a hierarchy of memory caches. The only reason a cache improves performance is the assumption that when a program accesses a datum, it will soon thereafter access that datum, or some other datum near it.