Garbage Collection Algorithms

Project for CSEP 521, Winter 2007

Rick Byers – rbyers@cs.washington.edu

1 Problem and applications

Almost all programming languages have a notion of a heap, or free-store, from which the programmer can request memory. Historically, the heap was managed explicitly by the programming by using allocate and release function calls in a library (such as malloc/free in C, and new/delete in C++). Explicit memory management has been a significant source of software bugs, and programmer pain (for example, using memory after it has been freed, freeing it twice, or forgetting to free it at all).

Starting around 1960, algorithms began to be studied for automatic memory management [McCarthy, 1960] [Collins, 1960]. With automatic memory management, a programmer can request space from the heap (eg. by instantiating a new object with a statement like "new MyObject()" in C# and Java), and the run-time system does the job of allocating the necessary memory and releasing it to be re-used when it's no longer needed. Automatic memory management is easier to use and less error-prone than explicit memory management, but is also usually considered to be less efficient and less flexible.

Automatic memory management is usually implemented by the runtime system using a family of algorithms called "garbage collection". Lisp was the first widespread programming language to adopt garbage collection in the early 60s. For various reasons (eg. complexity of implementation, efficiency, and stubbornness of programmers), garbage collection did not become popular in mainstream industrial programming until the 90s with the advent in popularity of the Java programming language. Today, virtually every modern high-level programming language has garbage collection as a core feature of it's design. Such languages include Java, C#, ML, Visual Basic, Python, Ruby, Scheme, Haskell (basically everything in widespread use except for C/C++). Garbage collection is generally considered to be one of the biggest improvements in programmer productivity since the advent of the high-level programming language.

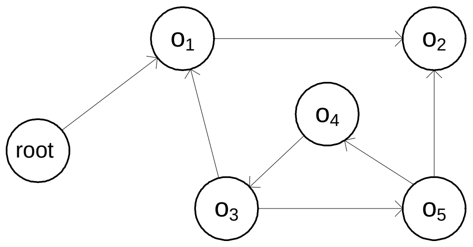
The basic problem of garbage collection is to identify the memory which is no-longer needed by the application, and to make it available for other allocations. Memory is determined to be no longer needed by the application, if there is no longer any way for the program to refer to the region of memory in question. The main challenge in designing a GC algorithm is to find one which is "efficient" for typical uses, where efficiency is measured according to many metrics (see part 3).

2 Formalization of the problem

A heap can be thought of as a set of objects, and a distinguished object known as the "root". Each object has an address, size, and set of references (which are addresses of other objects). Addresses can be thought of as non-negative integers in a contiguous address space (from 0 to some maximum limit N). In practice, GC heaps are normally implemented using a subset of a real process address space (with potentially non-contiguous "segments"), but we can conceptually map this onto a single contiguous logical address space. A heap has the constraint that no two objects may overlap. That is, given any two objects in the heap $o_1=(a_1,s_1)$, and $o_2=(a_2,s_2)$, it is always the case that either $a_1+s_1 \leq a_2$ (i.e. a_1 occurs entirely before a_2) or $a_2+s_2 \leq a_1$ (a_2 occurs entirely before a_1).

A heap can be modeled as a directed graph with the vertices o=(a,s) representing objects, and the edges representing the references between objects. The problem is then, given a heap at any point, modify the heap to improve the desired criteria (mainly availability of space for new allocations) while preserving the property that any object which is reachable from the root still exists in the modified heap. The most common and simplest modification is to remove some or all of the objects which have no path from the root. Other modifications are also possible, such as relocating (changing the address of) objects to coalesce regions of free space, or to preserve some ordering property.

For example, at some point in a program execution, the heap may look like the graph at right. If the garbage collector were to run at this point, it would determine that o_3, o_4 and o_5 are no longer reachable from the root, and therefore should be considered garbage. The collector may chose to remove some or all of these garbage objects, and possibly relocate some of the other reachable objects.



3 Garbage collection algorithms

Garbage collection algorithms have been an active field of research since 1960. There are many different variations on the basic GC algorithms, all of which attempt to maximize some metrics for typical allocation patterns. The dependence of an algorithm on the allocation pattern of the program means that there is usually no precise way to compare GC algorithms without also considering the exact context in which it will be used. In practice, GC algorithms are compared by using both imprecise general statements of benefits, and precise measurements of their behavior in specific benchmark scenarios. Some of the most important metrics for comparing GC algorithms include:

- Minimizing the time spent reclaiming memory.
- Minimizing the amount of wasted memory at all times.
- Minimizing the amount of memory necessary to perform a collection.
- Minimizing the time and resources necessary for the program to access the memory during normal execution (including maximizing CPU cache hit rate and minimizing OS page faults).
- The above usually implies maximizing the locality of reference, that is the tendency for objects which are used together, to be near each other in memory.
- Minimizing the pause-time experienced by an application during a collection.
- Minimizing the complexity of the algorithm itself (which in practice often translates into performance, adaptability, maintainability, correctness and security benefits).
- In specialized scenarios (such as small devices) there are often other metrics like maximizing battery life, or minimizing the number of writes to flash memory.

3.1 Mark-sweep

The earliest and most basic garbage collection algorithm is mark-sweep garbage collection [McCarthy, 1960], and most modern algorithms are a variant on it. Mark-sweep is a "stop-the-world" collector, which means that at some point when the program requests memory and none is available, the program is stopped and a full garbage collection is performed to free up space. In mark-sweep, each object has a "mark-bit" which is used during the collection process to track whether the object has been visited. Here is an algorithm for mark-sweep garbage collection implemented on top of some underlying explicit memory management routines, in which free regions of the heap are also considered objects with mark bits and a known size.

```
mark_sweep_collect() =
    mark(root)
    sweep()

mark(o) =
  If mark-bit(o)=0
    mark-bit(o)=1
    For p in references(o)
      mark(p)
    EndFor
  EndIf
```

```
sweep()
  o = 0
  While o < N
    If mark-bit(o)=1
      mark-bit(o)=0
    Else
      free(o)
    EndIf
    o = o + size(o)
  EndWhile
```

The mark-sweep algorithm operates in time linear in the size of the heap (i.e. O(N)). This doesn't directly tell us how much overhead it imposes on a program, because it must be invoked whenever an allocation fails, and so the overhead depends on parameters such as how big the heap is, and how much memory has become unreachable since the last GC. In practice, the overhead, as well as the pause-time, of mark-sweep collectors is high compared to other algorithms. Mark-sweep does however have the advantage of freeing all unused memory, but this free memory easily becomes fragmented (limiting the availability of larger contiguous regions). There is technically a space overhead for the mark-bit, but in practice a bit is usually re-purposed from some other run-time data structure, since it's only needed when the program is not running.

3.2 Semi-space

Semi-space garbage collection [Fenichel, 1969] is a copying algorithm, which means that reachable objects are relocated from one address to another during a collection. Available memory is divided into two equal-size regions called "from-space" and "to-space". Allocation is simply a matter of keeping a pointer into to-space which is incremented by the amount of memory requested for each allocation (that is, memory is allocated sequentially out of to-space). When there is insufficient space in to-space to fulfill an allocation, a collection is performed. A collection consists of swapping the roles of the regions, and copying the live objects from from-space to to-space, leaving a block of free space (corresponding to the memory used by all unreachable objects) at the end of the to-space. Since objects are moved during a collection, the addresses of all references must be updated. This is done by storing a "forwarding-address" for an object when it is copied out of from-space. Like the mark-bit, this forwarding-address can be thought of as an additional field of the object, but is usually implemented by temporarily repurposing some space from the object.

```
initialize() =
  tospace = 0
  fromspace = N/2
  allocPtr = tospace

allocate(n) =
  If allocPtr + n > tospace + N/2
    collect()
  EndIf
  If allocPtr + n > tospace + N/2
    fail “insufficient memory”
  EndIf
  o = allocPtr
  allocPtr = allocPtr + n
  return o
```

```
collect() =
  swap( fromspace, tospace )
  allocPtr = tospace
  root = copy(root)

copy(o) =
  If o has no forwarding address
    o’ = allocPtr
    allocPtr = allocPtr + size(o)
    copy the contents of o to o’
    forwarding-address(o) = o’
    ForEach reference r from o’
      r = copy(r)
    EndForEach
  EndIf
  return forwarding-address(o)
```

The primary benefits of semi-space collection over mark-sweep are that the allocation costs are extremely low (no need to maintain and search lists of free memory), and fragmentation is avoided. In addition to improving the efficiency and reliability of allocation, avoiding fragmentation also improves the locality of reference which means the program will typically run faster (due to paging and CPU cache effects). The primary drawback of semi-space is that it requires twice as much memory – at any given time during program execution, half of the available memory cannot be used. Semi-space collection executes in time proportional to the amount of reachable memory, and so unlike mark-sweep, can be very efficient if most memory is garbage at the time of collection. However, for a given heap size, semi-space requires many more collections than mark-sweep (since it only has half the space to work with), and so if most of the objects are reachable at the time of collection, semi-space becomes much less efficient than mark-sweep.

3.3 Other variations

There are many other variations on the basic GC algorithms above. Due to space constraints, I will give only a brief mention to a few of these variations.

Compacting garbage collectors typically use an algorithm like mark-sweep, but also re-arrange the objects to coalesce free-space to avoid fragmentation. This also often has the benefit of keeping the objects in memory ordered by their allocation time, which typically improves the locality of reference. Compaction has most of the benefits of the semi-space algorithm (efficient and reliable allocation), without the cost of the additional memory. Compaction does, however, require significantly more time and bookkeeping during collection to copy objects and update object references appropriately.

Generational garbage collectors are designed under the assumption that objects which are created recently are more likely to be garbage than objects which have been alive for a long time. Such collectors typically divide the heap into two or three generations, promoting objects from a generation to the next older one when they survive a collection. With some careful bookkeeping, it is then possible to perform partial collections of only the one or two younger generations, avoiding the cost of scanning through old objects which are likely to sill be alive. This results in a lower overhead in most scenarios, at the cost of some additional complexity.

Incremental garbage collectors attempt to minimize the pause time incurred due to a collection, at the expense of more overhead overall relative to stop-the-world collectors. This typically involves doing a little bit of the GC work at every allocation. This trade-off is appropriate for some interactive applications such as graphics-intensive programs where the user may notice a pause of even a fraction of a second. Concurrent garbage collectors are incremental collectors which perform collection in parallel with the program execution by using multiple CPUs.

Most high-performance modern industrial GCs (including the one in Microsoft's Common Language Runtime) are generational mark-sweep compacting collectors, with an optional concurrent mode for low-latency applications.

4 Open problems

Despite active research for the past 40 years, there are many open problems in garbage collection. Working on the Common Language Runtime team at Microsoft, I see that the GC team (which has some extremely bright people) is constantly faced with new challenges and unusual scenarios in which performance should be improved. GC algorithm design seems like more of an art than a science – constantly trading off various parameters based on the priority of expected usage models. I suspect that GC algorithm research will continue to be an active area of research for the next 40 years.

In addition to improving performance in various situations, some concrete areas of research include:

1. Interaction with non-memory resources. How should a GC account for the fact that objects often hold other resources which have an associated cost, and sometimes strict requirements on when they must be released for correct behavior of the program? For example, a process can typically only have a certain number of file handles open at once. Objects representing files may keep the associated file handles open until they are collected. How should a GC know to collect unused file objects? Could the idea of garbage collection be generalized to cover all OS resources in addition to just memory?
2. How should a GC algorithm and OS memory management system co-operate? For example, when should the OS notify a GC to perform a collection rather than page out some memory to disk? When should a GC avoid a collection because the cost of bringing in the pages from disk will be higher than the benefit of freeing any memory? Could the GC-related properties of a memory page somehow be summarized in memory, when the page is paged-out to disk by the OS?
3. How can we design GC algorithms which have good average-case performance, while limiting their worst-case performance to some provable bound? This is necessary for real-time applications which need hard guarantees on the time which certain operations will take to complete.
4. How can a GC algorithm making effective use of highly parallel computers (eg. >32 processors cores).

5. How should we design good tools and techniques for diagnosing and visualizing problematic memory usage, such as memory leaks and other performance problems?

5 Conclusions

Garbage collection has been a mixed blessing for the software industry. On the one hand, it does seem to have greatly improved the productivity of software developers, making writing correct software easier in the common cases. On the other hand, it adds a lot of complexity and uncertainty to the implementation of a programming environment. There are many different variations on GC algorithms, and they are usually impossible to evaluate strictly mathematically without considering a specific usage model.

With garbage collection it is easier to reason about the correctness of a program, but usually harder to reason about it's performance. Reasoning about performance of a program in a GC environment Overall, garbage collection shifts the burden for a difficult problem from the programmer to the programming environment implementer, which for most software engineering scenarios is an excellent trade-off.

6 References

[Collins, 1960] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655-657, December 1960

[Fenichel, 1969] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611-612, November 1969

[McCarthy, 1960] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184-195, 1960