CMPSCI 630 Systems **Spring 2022**

Lecture 6

Lecturer: Emery Berger *Scribe: Ryan Lee, Trace Crowdis*

6.1 Garbage Collection Finalizers

Garbage collectors function both as a memory allocator and as an automatic reclaimer of unused memory. One difference between garbage collection and manual memory management that results from automatic reclamation is that they do not have the same behavior with respect to object destruction. With manual memory management, it is possible to specify destructors for objects, ensuring that cleanup of external resources (e.g. sockets) occurs in a timely manner. While objects in memory managed by a garbage collector can have finalizers, the timing of them running depends on GC passes and thus cannot be guaranteed.

6.2 Garbage Collection, Caching, and Strong and Weak Pointers

Garbage collection introduces problems with caches, in that the existence of an object in a cache should not prevent its reclamation. This problem was solved with the creation of weak pointers that do not prevent an objects reclamation, and that instead return null if dereferenced after collection. However, this creates another problem: namely that objects could be evicted from cache between initial load and reuse if a collection pass runs between the two events. There is no good solution to this, other than by tuning cache parameters and eviction algorithms.

6.3 Garbage Collection and System Memory Pressure

Operating system notifications for low memory sounds like a natural mechanism that might help alleviate memory pressure. However, no operating system actually implements push notifications for this purpose, though Linux recently introduced a ‘psi‘ system that applications could poll to check for memory pressure.

More generally, when Linux runs low on memory, it can swap processes out to disk or, if necessary, invoke the OOM killer to terminate processes with high memory consumption. The OOM killer was (and remains) a hacky solution: in the past, it even attempted to kill the Linux kernel, although it is slightly better in this respect now.

6.4 Free-list Memory Allocation Structures

Memory allocators often use a free-list structure: the allocator finds a sufficiently large chunk of free memory to hand over, and deallocation keeps the chunk in a linked list to allow for easier reuse by later allocations. Adjacent freed chunks can also be coalesced to create bigger chunks. There are a number of ways chunks can be found.

6.5 Best Allocator against Adversary

The first-fit memory allocator scans the free chunks and returns the first one large enough for the requested allocation, while the best-fit memory allocator returns the "best" one, i.e. the smallest one large enough for the requested allocator. First-fit turns out to be as good as optimal in the worst case, while best-fit actually performs horribly. That optimal scenario actually occurs when the object size is constant. When you pit a best allocator against an adversary, one that performs perfectly, the actual amount of memory used is $O(log(\frac{S}{s})) * M$, where S is the biggest memory size the program will request, s is the smallest memory size, and M is the memory the process would use in a perfect scenario.

The term fragmentation refers to the quantity of memory consumed over the ideal requested memory. The implication being that in some cases a certain amount of memory will always be wasted.

6.6 Compaction

There is a distinction between the memory the user thinks they are using and the physical memory. From the users perspective the first thing instantiated in memory is at the first address of memory but for the physical memory it could be stored in any free space. With this scheme the virtual memory space can be larger than the actual physical memory. When one set of virtual memory runs into another, perform a swap to free up space for the active virtual memory. This can go wrong leading to page faults where a program tries to access virtual memory that isn't loaded on physical memory. Compacting is the process by which active physical memory is relocated into a compact group, effectively organizing physical memory into active at the front and free after.

Meshing is a process of virtual memory compaction in order to save physical space. It relies on the ability to find pointers and once it does it performs a "fix-up" which points it to another piece of virtual memory so that they will be stored together in physical memory.

6.7 Garbage Collection

There are two main metrics when it comes to garbage collection (GC), compaction (space efficiency) and ease of use (correctness). In C/C++ if you call free(-) too soon it could lead to an error when you later try to use what was freed, this is "unsound". An Address Sanitizer is a memory error detector that can detect just this, along with other errors. The original form of garbage collection was a stop-the-world full heap collection method, meaning it froze all other processes while it checked and freed memory from the entire heap. This was probably done by Mark Sweep (MS) or Mark Sweep Compact (MSC). In the original GC this was generally done when memory ran out but this poses a problem for compaction which is more expensive the closer to being full the heap is.

A form of GC is called Generational GC where the "Strong" Generational Hypothesis is: most objects die young. It goes like this: build a "nursery"/"eden"/"young generation" (first memory allocation), then some processes will die (the hypothesis) so copy the survivors to the real heap "old generation", then burn the nursery. This allows the full heap to fill slowly reducing the quantity of calls to GC.

Similar to Generational GC is Semispace GC. First divide the heap into two sections and fill up only one and call it the "from space", then when full relocate all live objects to the unused half and call it the "to space", then switch the names and repeat. The relocation between the from and to space results in a natural compaction as you line up live objects one after the other in the to space. The obvious limit to this method is

that you are limiting your effective memory to half of the available memory. Another instance of the classic space-time trade off problem.

Garbage collection takes up about 3-5x as much space as memory allocation in the same amount of time. There are three types of GC: concurrent, incremental, and parallel.

6.8 $2 < 3$

$2 < 3$