CMPSCI 630 Systems **Spring 2017**

Lecture 4

Lecturer: Emery Berger *Scribe: Vinitra Ramasubramaniam, Arjun Sreedharan*

4.1 Classification of Garbage Collectors

4.1.1 Conservative vs. Precise

In certain environments, it's not possible for a garbage collector to identify whether an object in memory is an address valid for garbage collection. So, the GC algorithm scans memory and makes a conservative assumption that it if looks like a valid address and that address points to is something valid, it can be considered for GC. Such garbage collectors are said to be conservative.

In other environments, garbage collectors have access to type information. The garbage collector can then do GC on those objects it precisely knows is a pointer. This type of garbage collectors are said be be precise.

4.1.2 Stop-The-World vs. Concurrent

In Stop-The-World garbage collectors, garbage collection is never interleaved with the user program (referred to as *mutator*). When the user program executes and the heap is near exhaustion, the user program is paused and the garbage collector runs. Once the garbage collector completes, the user program is resumed. Thus, Stop-The-World garbage collectors have a high latency but gives good throughput. Figure **??** shows how stop the word garbage collector works.

Concurrent garbage collectors interleave the garbage collector as one or more thread along with the user

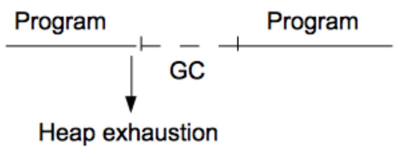

Figure 4.1.2.1: Stop the world Garbage Collector

program. Since, the garbage collector runs concurrently, a virtual lock needs to be used on the heap memory to avoid clashes between the garbage collector and the user program. They provide reduced latency but with a trade-off of bad throughput.

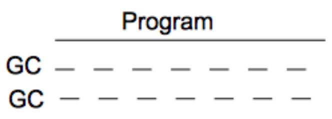

Figure 4.1.2.2: Concurrent Garbage Collector

4.1.3 Serial vs. Parallel

Serial garbage collectors execute only a single thread of the garbage collector program. However, a parallel garbage collector on the other hand, runs many garbage collector threads in parallel. Figure **??** shows an embarrassing parallel program and Figure **??** shows an embarrassingly (inherently) sequential program.

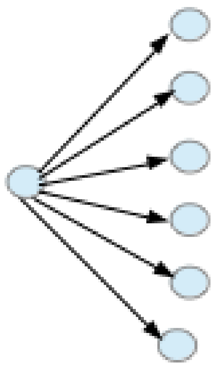

Figure 4.1.3.3: Embarrassingly parallel program

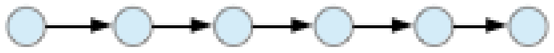

Figure 4.1.3.4: Embarrassingly sequential program

4.1.4 Non-moving/Non-relocating vs. Compacting

Bump-pointer allocation is fast and the garbage collector is invoked only when we need more memory for allocation. Over time, the heap will get fragmented. Fragmentation is given by Equation **??**. So as to defragment the memory, the garbage collector identifies all pointers to memory and moves them to adjacent blocks. This is referred to as *compaction*. A GC Map which contains the pointers of all objects can be obtained from the compiler for compaction.

$$fragmentation = \frac{\text{Amount of heap allocated}}{\text{Amount of heap in use}} \tag{4.1.4.1}$$

4.2 Incremental Garbage Collector

Garbage collection is done in little bits incrementally. For example, the garbage collector can perform marking on a portion instead of marking the whole program.

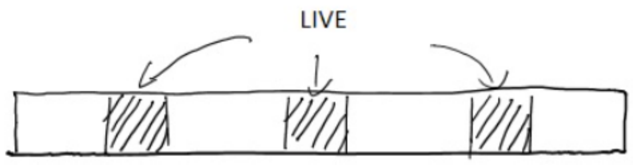

Figure 4.1.4.5: Fragmentation

4.3 Generational Garbage Collector

Generational garbage collector (also known as Partial Garbage Collector) is based on the generational hypothesis. The generational hypothesis states that most objects in a program die young - referred to as object mortality. In generational garbage collector, there are 2 heaps - *nursery* and *old*. The garbage collector keeps track of the pointers in *old* to *nursery* in a "remembered set". During garbage collection, only the globals, stacks, "remembered set" and *nursery* are scanned and memory is reclaimed from the *nursery* (referred to as Minor Collection). Any data that wasn't reclaimed in the *nursery* is copied to *old* and all pointers are updated. *old* is scanned and reclaimed once in a while (and this is referred to as Major Collection). Similarly, many different generation levels can exist in a generational garbage collector. Some collectors also have the *oldest* generation referred to as *immortal* because objects that are moved to this level are never reclaimed for the entirety of the program. Figure **??** shows generational garbage collector.

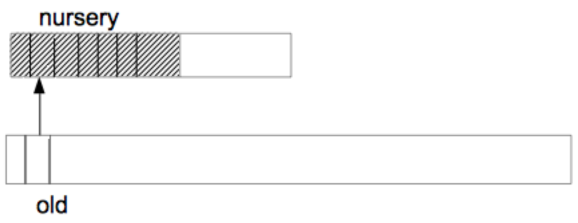

Figure 4.3.0.6: Generational Garbage Collector

4.4 Semi-space Garbage Collector

In a semi-space garbage collector, the heap is divided into 2 - *from* and *to*. This requires 2X memory. Memory is allocated only from the *from* heap. Once the *from* heap is full, the garbage collector copies compactly data from the *from* heap to the *to* heap. All pointers are updated and forwarding pointers are left for memory in the *from* heap. Now the *from* heap becomes the *to* heap and vice versa. Figure **??** shows how semi-space garbage collectors work.

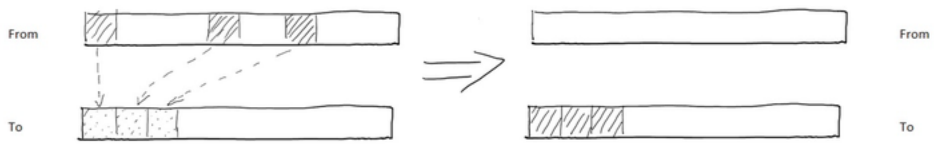

Figure 4.4.0.7: Semi-space Garbage Collector

4.5 Thrashing and Locality

Thrashing is an undesirable situation that occurs due to a constant state of paging, where data is swapped between memory and disk. Thrashing causes high degradation of performance.

There are 2 types of locality depending on how you access data in space and time:

1. Temporal Locality - refers to the reuse of same data within a small amount of time. Data that is most recently used has a higher probability of being used again.
2. Spatial Locality - refers to the idea that you are very likely to access neighboring locations in the very recent future when you access some data location.

Caches use the idea of locality and help reduce latency. Garbage collectors generally have bad locality as they are very random in the memory locations they access.