CMPSCI 630 Systems **Spring 2022**

Lecture 6

Lecturer: Emery Berger *Scribe: Kevin Chen, Bin Wang*

6.1 Space Configuration in Garbage-Collected Languages

In most garbage-collected languages such as Java, users can specify a maximum heap size. This determines the maximum amount of memory the runtime can make use of. When this limit is reached, the garbage collector may first try to reclaim as much memory as possible to make space for new objects. But what if absolutely no more memory can be reclaimed upon reaching the maximum heap size? For a user program, there is really no meaningful action to take besides shutting down. Though some systems can dynamically allocate memory as needed. More specifically, they maintain two separate memory limits, a soft limit and a hard limit. Whenever the soft limit is reached, they allocate more heap memory until the hard limit is reached.

6.2 Liveness vs. Reachability

An object is called live if it will continue being used some time in the future. Note that not all reachable objects are live because some objects may still be reachable, but the user has no intention of using it anytime in the future. In other words, the set of live objects is a subset of the set of reachable objects. The gap between these two sets of objects is called a leak in garbage-collected languages.

6.3 Space-Time Tradeoff in Garbage-Collected Languages

For a garbage-collected language, the execution time decreases as the maximum heap size increases and will finally converge to the ideal execution time without a garbage collector. In particular, if the maximum heap size equals the minimum heap size to keep all live objects, the execution time will become almost infinitely long. However, to make matters more complicated, some people argue that a garbage-collected language may perform even better than ideal since the garbage collector may compact objects in memory and thus improve spatial locality.

6.3.1 Empirical Results

An empirical study was conducted to quantify the space-time tradeoff in a garbage-collected language, in particular, Java. Researchers first ran a Java program as usual and recorded all object allocation and read/write accesses. Given the complete execution history, they then ran the same Java program again but replaced the garbage collector with perfect timings of `malloc()` and `free()`. According to their experiments, the general space-time tradeoff holds for Java. More specifically, the time overhead at $3\times$ minimum heap size is around 20%, and the time overhead at $5\times$ minimum heap size is almost negligible. However, the garbage-collected version will not perform better than the version with manual memory management.

6.3.2 Practical Significance

From a software engineering point of view, garbage collectors make it easier for programmers to produce correct programs. There has long been debate over how much overhead a garbage collector will incur and whether a garbage collector can even achieve potentially higher performance than programs with manual memory management. The reality is that there is no single definitive answer to this. One has to consider the space-time tradeoff in garbage-collected languages.

6.4 Precise GC vs Conservative GC

Q: *Why (under what conditions) does the generational hypothesis hold?*

A: It'a an interplay between the programming language, the culture, the libraries, and the style of coding, etc.

1. Non-use of stack. If a programming language doesn't have stack objects, it will naturally create a bunch of objects on the heap and many of them will die young. For example, LISP creates a lot of objects and they are all on the heap. In LISP there is no notion to explicitly allocate something on the stack. Also, if an object created in a function is still referenced by a pointer after the function scope ended, that object has to be allocated on the heap. Languages like Java or JavaScript uses a technique called *escape analysis* to identify objects that don't escape the function scopes they are defined in so that those objects may be allocated on the stack.

2. Functional programming style. In programming Languages like C/C++ you can modify objects in place and it's very efficient because no new objects need to created. For example, `sort()` in Python sorts a list in place, while in functional languages, `sort()` will return a new list that is sorted while the original list is unchanged. If every function call always creates a new object rather than changes an object in place (which is the case with functional programming languages) then you end up with many objects that you no longer need. It can also be a culture thing: you can program in functional style in non-functional programming languages.

3. There is also a problem specific to Java that is referred to as "non-struct" by the professor. In java, when you create an object from a class with other class members, you also create a bunch of new objects for each of the class members. So when you explicitly creates one objects you might be implicitly creating n objects for its non-primitive members. There is because in Java everything is a reference, while in C# `structs` the memory is laid out continuously for its members (no references) because C# has learned the lesson from Java. There is a programming language that even worse called Tcl. The professor then spent almost ten minutes making fun of the insanities in Tcl.

GC algorithms like mark sweep and reference counting are relatively straightforward. However, things get tricky when you have no information about pointer locations or types, e.g., in C/C++ or compiled code. We need some way to identify pointers in order to do GC. In Java, the compiler generates a *GC map* which stores information about pointer locations in the memory. Other language may contain type information and metadata with all the objects so you can look it up. In contrast, in C there is no type information (debug information is inadequate and it's often stripped when shipping binaries for various reasons). Relocation is also difficult because you can not tell whether whether a chunk of memory that appears to be a pointer is actually pointer: it may as well be user data that happens to look like an address.

Precise GC uses type information to do 1) reachability analysis (for garbage collection) and potentially 2) relocation (compaction). A precise GC is guaranteed to reclaim all unreachable objects.

For situations where type information is unavailable, conservative solves the GC problem but not relocation problem. Hans Boehm (who also created the calculator on Android that has clever arithmetic) came up with the idea of conservative GC when he was working on the implementation of the Russell programming language. He didn't want to implement GC maps in Russell because 1) he thinks it's too complicated and 2) it's hard to compile to C. Back in the days it was normal for people to make languages compile down to C because the C compiler is universal: every device on the planet has a C compiler. It also means that you don't have to worry about code generation and optimization. The problem is that when you compile to C, all the type information is lost.

Boehm's brilliant idea is that **it's OK to not reclaim everything that's reclaimable (unreachable)**. Conservative GC works in the following way: the GC scans every word in the memory and does a duck test for pointers. If the GC finds what appears to be a pointer, it will follow that pointer and also check through the memory region that is pointed to. The conservative GC can then do a mark sweep using the "pointers" it found. Note that some of the "pointers" might actually be data but the conservative GC will follow them anyway. The result is that some garbage that's reclaimable won't get reclaimed. However, if the GC has traversed everything that appears to be a pointer, it must have traversed the set of true pointers (because $\{\text{true pointers}\} \subseteq \{\text{conservative pointers}\}$), then what's left is definitely garbage. Boehm also did many clever tricks (e.g., range checking, blacklisting, and alignment checking) when implementing his conservative GC to make things more efficient. Boehm's implementation can be downloaded but it's very complicated and makes heavy use of macros.

Boehm later extended his conservative GC to work with multi-threaded programs. It's really hard to do because now the GC need to follow the globals and the stacks of every thread (this collection is called roots). What makes things worse is that there might be race conditions if the threads and the GC are running at the same time. The original system had to send interrupts to every thread, and all the threads will wait until GC is over. Later it was changed to use memory protection. It's a complicated system but it works.

There is an interesting problem about conservative GC: conservative GC has solved the problem of use after free since the 80s, but why hasn't the world embraced conservative GC? A possible explanation is the overhead associated with GC. However, different people may have different perspectives about what is "too much" overhead and people may not even be sure about the true overhead of GC because they are not even testing. The professor's answer is that this is a societal question rather than a technical question. The real answer to this question may actually be FUD (fear, uncertainty, doubt). Another problem is moral hazard: a GC that guarantees no use after free will make developers sloppier with memory which leads to worse memory consumption patterns. The professor is skeptical of the moral hazard argument because it hasn't been verified or proven, and it can be used against any advances.