CS 131 Programming Languages

November 24, 2009

Performance

Heap management issues: how can we write a garbage collector that can do its job reliably and fast? When you're managing objects in a pool of memory, the objects can point to other objects, and your job is to figure out which objects are actually needed by the program and which aren't. For this we need to look at the program's *roots* (global variable that points to the heap). Pointers from program's static data, registers, and stack may point to the objects in memory pool. We need to take the transitive closure of the "pointed-to" relation (X → Y → Z = X → Z). Using these two finds, we can find accessible objects and get rid of the inaccessible ones.

How would one find program's roots? For example, Python interpreter uses reference count to immediately reclaim certain objects; however, that still doesn't catch circularly linked lists. To find program's roots, Python accesses the *global namespace dictionary* which members point to the heap. There's a built-in variable inside the Python interpreter that contains the address of global namespace. Instruction pointer is another root as it points to the next instruction to be executed. Environment pointer points to the frame of current function and is also a root.

It's possible to write an interpreter for C but it's hardly ever done. C is usually used as compiled code and libraries. Most important primitives that manage the heap are `malloc` and `free`. The difficulty arises because `malloc` is only given the number of bytes to be allocated, and the C runtime doesn't have any information regarding which pieces of data inside resulting data structure is actually pointers into the heap and which isn't. How would one write a garbage collector for C?

H. J. Boehm from Xerox/PARC invented *conservative garbage collector* that "finds" pointers by looking at a bit pattern of the data structure members. One problem with this approach is that it sometimes falsely thinks an object is in use when it's not (treating certain large numbers as pointers). Another problem is performance since it must examine all storage to find roots. However, it works even in C when there's no help from runtime, and it results in a smaller memory footprint.

Besides locating objects in use, we also need to keep track of the unused space. We can maintain a linked list, each node of which lists the address of the next guy in the list as well as the size of the memory block that this node links to. This approach is known as *free list*. The disadvantage is external fragmentation – if there's no block big enough to hold the requested data structure, the call fails. Another disadvantage is the existence of *runts* at the start of the list that chew up CPU time. Some workarounds are *next fit algorithm* and *best fit algorithm* that find the most appropriate block for the given size of data structure. Another workaround is to keep the free space contiguous so that there's no fragmentation; this will require moving allocated objects around and updating pointers to those objects (impossible in a conservative garbage collector). An approach called *quick list approach for common sizes* exploits the fact that most programs do a majority of allocations using common sizes of data structures (such as 8-byte cons structure in Scheme). The memory manager can then allocate a big data structure within which it can efficiently allocate and free these fixed-size chunks of memory.

Cost Models

We've been talking about costs all throughout this course. Costs include time, space (RAM), battery (power), network, etc. How does one develop a cost model? Important thing here is a mental model of implementation that's "close enough" to make performance predictions that actually work. For example, lists (as in Scheme) can be analyzed by finding the cost of operations on them. Prepending to a list is O(1), finding length of a list is O(n), appending lists is O(2n) for (append X Y Z). Appending a short list to a long one is fine while appending a long list to a short one is a much bigger cost (last list is not touched during this operation). To summarize, operations that are cheap in one language (such as prepending a list in Scheme) might be more expensive in other language (such as prepending a list in Python).

Traditional garbage collection is becoming obsolete. *Generation-based garbage collector* used in Java is based on several ideas: instead of having one big heap, it is divided into "generations" (as in genealogy): "old" and "stable" objects are in the first heap while very recent objects are in the last heap. Most often, pointers go "left" in this scheme (where "left" is earlier generations). Objects that are older are usually meant to stay around for a while, while new objects are more likely to disappear soon. Therefore, garbage is more likely to be found on the right (later generations). Free space is represented by a pointer and `malloc` simply moves it when allocating new memory – which results in a very fast execution (~10 instructions). To allocate a new generation, usually roots

(objects in use) are moved to a newly allocated generation, after which the newly allocated generation is appended to the end of the generation list, becoming the "youngest" ones. This is known as *copying collector* and its downside is that it needs to change the pointers after moving the object (not as big of a downside since these "young" objects don't have many pointers pointing at them). The advantage of this collector is that its cost is proportional to the number of objects in use.

Associated with all the objects in Java is a method `hashCode()` and `finalize()`. Garbage collector calls `finalize()` before reclaiming the storage occupied by the object. This method is user code that will run before the object is free. Generation-based garbage collector cannot call `finalize()` because it doesn't look at the object it frees. Therefore, Java actually has two garbage collectors: generation based and mark-and-sweep.