Comp 411
Principles of Programming Languages
Lecture 30
Storage Managment

Corky Cartwright

November 21, 2011

Heap Management

In all of our interpreter designs (including those written in machine code), we have presumed the existence of a heap supporting dynamic allocation (`new` operations in C or Java).

For efficient space utilization, the heap must reclaim storage that is no longer in use. In C and C++, the programmer is responsible for explicitly reclaiming storage (the `malloc` and `new` operations in C and C++). In a well-written C/C++ program all dynamically allocated storage is freed just before it becomes inaccessible (no object becomes inaccessible prior to being freed; once freed, an object is never referenced again).

In practice, manual storage management is clumsy (interfaces become much more complex) and error-prone.

Manual Heap Management

- This subject may be boring but it is *non-trivial.*
- Crux of the problem: what happens when the original sequential block of storage is exhausted? Reallocate freed storage. But this task is much harder than allocating from the original sequential block. How are freed blocks managed? As a doubly linked list? Are adjacent blocks coalesced? How do you test for adjacent blocks? Ultimately allocating reused storage involves *search*. It is *not* a constant time operation. What is the best policy? First-fit? Best-fit? Should the pointer for the next search be the resumption value for the last search or start at the beginning?
- Only tractable approach to manual storage management: arenas [regions]. Unfortunately, arenas only work for some patterns of allocation/deallocation.
  An arena is a sequential block of storage that is freed as a unit. All of the objects allocated in an arena must have a common deallocation (death) time. The arena is gradually filled by allocation operations (directed to that arena), but nothing is freed until the arena is no longer relevant to the computation (*e.g*. a compiler begins a new phase, so data structures associated only with the previous phase can be deallocated).

Manual Heap Libraries

- A well-written heap library generally out-performs customized heap managers written by programmers (*e.g*. overriding `new` and `delete` in C++. Exception: well-chosen arena applications.
- If allocation and deallocation do not follow a simple discipline/pattern that is exploited by the manual heap manager, garbage collection generally wins.
- Typical penalty created by use of automatic storage management: a space penalty to hold a larger heap.

Manual Heap Libraries

- A well-written heap library generally out-performs customized heap managers written by programmers (*e.g*. overriding `new` and `delete` in C++. Exception: well-chosen arena applications.
- If allocation and deallocation do not follow a simple discipline/pattern that is exploited by the manual heap manager, garbage collection generally wins.
- Typical penalty created by use of automatic storage management: a space penalty to hold a larger heap.

Automatic Heap Management

- Two fundamental approaches to automatic heap management:
- *Reference counting*: every object includes a field that counts the number of objects (every data structure containing a heap pointer including an ordinary variable is called an object) pointing to it. Some schemes defer updating reference counts, but an object with a deferred count cannot be freed.
- *Garbage collection*: periodically (usually when no free space is left) the heap management system determines which objects in the heap have become inaccessible.
- The term *garbage collection* is not used consistently in the literature. In some cases, it means any approach to automatic heap management. In others it refers to schemes that do not rely on reference counting. I will use the latter convention.

Reference Counting

- Every object includes an additional field that counts the number of objects pointing to it; this field must be large enough so that it cannot overflow (*e.g.*, machine word [address] size).
- When an object is created, its reference count is set to 1 (and a pointer to it must be created within some other object, perhaps a variable).
- When a pointer field in an object is mutated, the reference count for the old object is decremented and the reference count for the new object is incremented. This combined operation must be atomic with respect to the checking for 0 reference counts.
- When a reference count for an object becomes 0, the object is *freed* (returned to free storage).