13 Dynamic Storage Allocation

Why isn't static allocation sufficient for everything? Unpredictability: can't predict ahead of time how much memory, or in what form, will be needed:

- Recursive procedures. Even regular procedures are hard to predict (data dependencies).
- Complex data structures , e.g. linker symbol table. If all storage must be reserved in advance (statically), then it will be used inefficiently (enough will be reserved to handle the worst possible case).
- OS doesn't know how many jobs there will be or which programs will be run.

Need dynamic memory allocation both for main memory and for file space on disk.

Two basic operations in dynamic storage management:

- Allocate
- Free.

Dynamic allocation can be handled in one of two general ways:

- Stack allocation (hierarchical): restricted, but simple and efficient.
- Heap allocation: more general, but less efficient, more difficult to implement.

Stack organization: memory allocation and freeing are partially predictable (as usual, we do better when we can predict the future). Allocation is hierarchical: memory is freed in opposite order from allocation. If alloc(A) then alloc(B) then alloc(C), then it must be free(C) then free(B) then free(A).

- Example: procedure call. X calls Y calls Y again. Space for local variables and return addresses is allocated on a stack.
- Stacks are also useful for lots of other things: tree traversal, expression evaluation, top-down recursive descent parsers, etc.

A stack-based organization keeps all the free space together in one place. Advantage of hierarchy: good both for simplifying structure and for efficient implementations.

Heap organization : allocation and release are unpredictable (this is **not** the same meaning of heap as in the data structure used for sorting). Heaps are used for arbitrary list structures, complex data organizations. Example: payroll system. Don't know when employees will join and leave the company, must be able to keep track of all them using the least possible amount of storage.

- Memory consists of allocated areas and free areas (or holes). Inevitably end up with lots of holes. Goal: reuse the space in holes to keep the number of holes small, their size large.
- Fragmentation: inefficient use of memory due to holes that are too small to be useful. In stack allocation, all the holes are together in one big chunk.
- Refer to Knuth volume 1 for detailed treatment of what follows.
- Typically, heap allocation schemes use a *free list* to keep track of the storage that is not in use.

  Algorithms differ in how they manage the free list.
- Best fit : keep linked list of free blocks, search the whole list on each allocation, choose block that comes closest to matching the needs of the allocation, save the excess for later. During release operations, merge adjacent free blocks.
- First fit : just scan list for the first hole that is large enough. Free excess. Also merge on releases. Most first fit implementations are rotating first fit.
- Best fit is not necessarily better than first fit. Suppose memory contains 2 free blocks of size 20 and 15.
  - Suppose allocation ops are 10 then 20: which approach wins?
  - Suppose ops are 8, 12, then 12: which one wins?

First fit tends to leave "average" size holes, while best fit tends to leave some very large ones, some very small ones. The very small ones can't be used very easily. Knuth claims that if storage is close to running out, it will run out regardless of which scheme is used, so pick easiest or most efficient scheme (first fit).

- Bit Map : used for allocation of storage that comes in fixed-size chunks (e.g. disk blocks, or 32-byte chunks). Keep a large array of bits, one for each chunk. If bit is 0 it means chunk is in use, if bit is 1 it means chunk is free. Will be discussed more when talking about file systems.
- Pools : keep a separate allocation pool for each popular size. Allocation is fast, no fragmentation. But may get some inefficiency if some pools run out while other pools have lots of free blocks: get shuffle between pools.

Reclamation Methods: how do we know when dynamically-allocated memory can be freed?

- It's easy when a chunk is only used in one place.
- Reclamation is hard when information is shared : it can't be recycled until all of the sharers are finished. Sharing is indicated by the presence of *pointers* to the data. Without a pointer, can't access (can't find it).

Two problems in reclamation:

- Dangling pointers: better not recycle storage while it's still being used.
- Memory leaks: Better not "lose" storage by forgetting to free it even when it can't ever be used again.

Reference Counts : keep track of the number of outstanding pointers to each chunk of memory. When this goes to zero, free the memory. Example: Smalltalk, file descriptors in Unix. Works fine for hierarchical structures. The reference counts must be managed carefully (by the system) so no mistakes are made in incrementing and decrementing them. What happens when there are circular structures?

Garbage Collection : storage isn't freed explicitly (using free operation), but rather implicitly: just delete pointers. When the system needs storage, it searches through all of the pointers (must be able to find them all!) and collects things that aren't used. If structures are circular then this is the only way to reclaim space. Makes life easier on the application programmer, but garbage collectors are difficult to program and debug, especially if compaction is also done. Examples: Lisp, capability systems.

How does garbage collection work?

- Must be able to find all objects.
- Must be able to find all pointers to objects.
- Pass 1: mark. Go through all statically-allocated and procedure-local variables, looking for pointers. Mark each object pointed to, and recursively mark all objects it points to. The compiler has to cooperate by saving information about where the pointers are within structures.
- Pass 2: sweep. Go through all objects, free up those that aren't marked.

Garbage collection can be expensive, but recent advances are encouraging.