

CS 414 : Operating Systems

UNIVERSITY OF VIRGINIA
Department of Computer Science

Spring 2008

Topic 11: Dynamic Storage Allocation

- Two aspects of memory allocation:
 - Static allocation: finding a slot in memory big enough to load a.out
 - Dynamic allocation: while executing the program, finding space for additional memory requests
- Why isn't static allocation sufficient for everything?
 - Recursive procedures: e.g., factorial
 - Complex data structures: e.g., symbol table
 - What is the problem if all storage must be reserved in advance (statically)?
- OS cannot predict when a process will come and ask for storage space: Need dynamic memory allocation both for main memory and for file space on disk.
- Two basic operations in dynamic storage management:
 - Allocate
 - Free
- In general, dynamic allocation can be handled in one of two general ways:
 - Stack allocation (hierarchical; LIFO): 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 (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)`.
- 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. Why is it important?
- Heap organization: allocation and release are unpredictable. 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.
 - 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. Merge on releases.
 - Is best fit better than first fit?

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.

- 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 1, it means chunk is free. When freeing, no need to merge.

Problems with BF, FF, and bitmap?

- Pools: keep a separate allocation pool for each popular size. Allocation is fast, no fragmentation. It 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.
- 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.
- Two approaches: reference counts and garbage collection
- Reference Counts: keep track of the number of outstanding pointers to each chunk of memory. When this goes to zero, free the memory.
- Garbage Collection: no explicit free operation. 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 incredibly difficult to program and debug, especially if compaction is also done.

- How does garbage collection work?
 - Must be able to find all pointers to objects.
 - Must be able to find all 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 is often expensive: 10% or more of CPU time used in systems that use it.