Dynamic Storage Allocation (Topic 5)

홍 성 수

서울대학교 공과대학 전기 공학부
Real-Time Operating Systems Laboratory

Dynamic Storage Allocation (1)

- Static vs Dynamic:
  - why isn't static allocation sufficient for everything ?
  - Answer: Unpredictability.

    can't predict ahead of time how much memory, or in what from, will be needed.

Dynamic Storage Allocation (2)

- Example of memory request unpredictability:
  - 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).
  - For example, OS doesn't know how many jobs there will be or which programs will be run.

Dynamic Storage Allocation (3)

- Need dynamic memory allocation both for main memory and for file space on disk.
- Two basic operations in dynamic storage management:
  - Allocate and Free

Dynamic Storage Allocation (4)

- Dynamic allocation can be handled in one of two general ways:
  - Stack allocation (hierarchical):

    Restriced, but simple and efficient.
  - Heap allocation:

    More general, but less efficient.

    More difficult to implement.

Dynamic Storage Allocation (5): Stack

- Stack organization:
  - Memory allocation and freeing are partially predictable.
  - Allocation is hierarchical:

    Memory is freed in opposite order from allocation.

    (E.g.) alloc(A); alloc(B); alloc(C); free(C); free(B); free(A)
  - Examples:

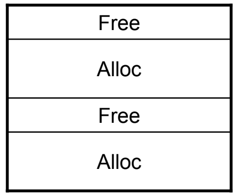
    Procedure call frames, tree traversal, expression evaluation, parsing.
  - A stack-based organization keeps all the free space together in one place.

Dynamic Storage Allocation (6): Heap

- Heap organization:
  - Allocation and release are unpredictable.
  - Heaps are used for arbitrary list structures, complex data organizations.
  - Examples: new in C++, malloc() in C.

Dynamic Storage Allocation (7): Heap

- Memory consists of allocated areas and free areas (or holes).



∗Inevitably end up with lots of holes

Dynamic Storage Allocation (8)

- 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.
  - No problem 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.

Dynamic Storage Allocation Algorithms (1)

- 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.
    - During release operations, merge adjacent free blocks.
  - First fit:
    - Just scan list for the first hole that is large enough.
    - Also merge on releases.
    - Most first fit implementations are rotating first fit.

Dynamic Storage Allocation Algorithms (2)

- 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.
  - Suppose ops are 8, 12, then 12.
- 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.

Dynamic Storage Allocation Algorithms (3)

- Heap implementation data structures:
  - Bit Map:
    - Used for allocation of storage that comes in fixed-size chunks.
    - Examples: disk blocks 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 bit means chunk is free.
  - Pools:
    - Keep a separate allocation pool for each popular size.
    - Allocation is fast, no fragmentation.
    - May get some inefficiency if some pools run out while other pools have lots of free blocks:
      - Get shuffle between pools.

Garbage collection (1)

- 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).

Garbage collection (2)

- Two problems in reclamation:
  - Dangling pointers: Better not recycle storage while it's still being used.
    - A reference that doesn't actually point at anything valid. Usually this happens because it formerly pointed to something that has moved or disappeared.
  - Memory leaks: Better not "lose" storage by forgetting to free it even when it can't ever be used again.

Garbage collection (3)

- Implementing reclamation:
  - 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.
    Problems with circular data structures.

Garbage collection (4)

- 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 data structures.
  Works with circular data structures.
  Makes life easier on the application programmer, but garbage collectors are difficult to program and debug, especially if compaction is also done.
  Examples: Lisp

Garbage collection (5)

- 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 structures.
  - Pass 2: Sweep.

    Go through all objects, free up those that aren't marked.

Garbage collection (6)

- Garbage collection is often expensive: 20% or more of all CPU time in systems that use it.
- 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 structures.

Garbage Collection (7)

- Pass 2: Sweep

  Go through all objects, free up those that aren't marked.
- Garbage collection is often expensive: 20% or more of all CPU time in systems that use it.
- Conservative garbage collection:
  - Treat all memory as pointers.
  - Works on C, C++