Nathan Tung

CS111 Chapter 6.2 + Paper Outline #12

Chapter 6, Sections 6.2-6.2.1 (pages 321-323). Dynamic Storage Allocation: A Survey and Critical Review. Sections 1.1-1.3 and section 2-21 (pages 6-13) and sections 3.3 and 3.4 (pages 42-45).

- Multilevel Memories
  - Using digital memory devices (RAM chip or magnetic disk) to prevent I/O bottlenecks
    - Tradeoffs in varying capacities, costs, and speeds; sometimes we need only one type of memory
    - Using both means we have to decide when to use the small/fast or large/slow memory
      - Also increases maintenance effort if memory configuration changes
  - Problems with memory that is both large and fast
    - Practical: increase in memory size leads to increase in problem sizes
      - Amount of data we use also grows
    - Intrinsic: memory has tradeoff between latency and size
      - Smaller size means few memory cells can fit close by, so the far ones have high latency
    - Memory chips have varying latency, so it's hard to maximize performance when put together
- Memory characterization
  - Capacity (bits or bytes): RAM has around 10 MB, magnetic disks have around 100+ GB
  - Average random latency (seconds or processor clock cycles): RAM is measured in ns
  - Cost (currency per storage): RAM has cents/MB, magnetic disks have dollars/GB
  - Cell size (bits/bytes transferred by one READ/WRITE): RAM cell size is 4, 8, or 16 bytes; disks, 512+ bytes
  - Throughput (bits/s): RAM transfers in GB/s, magnetic disks transfer in 100+ MB/s
- Memory Allocators/Allocation
  - Poorly-designed or implemented allocators lead to a huge waste of main/cache memory
  - Some OS are now supplying better allocators
  - Programs that run for long times might degrade due to allocators; but these are also the most important
- Allocator's Task
  - Track what parts of memory are in use and what parts are free
    - Minimize wasted space without time cost
  - Conventional allocator cannot control number of live blocks nor compact memory (freeing contiguous memory, etc.)
  - Responds immediately to a request for space; cannot change decision, memory becomes inviolable
  - Allocator: online algorithm that responds to requests sequentially, immediately, and irrevocably
  - When application frees blocks in any order, holes will form between live objects
    - Leads to fragmentation: numerously small holes cannot satisfy future large requests
  - No provably good allocation algorithms, but any allocator is bad in some situation
  - Worst case fragmentation: M log2(n), where M is live data amount, n is ratio of smallest/largest sizes
  - Some allocator's are decent or fairly good in practice; if you have a bad one, replace it with a better one
  - Allocators exploit regularities in program behavior
- Strategies, placement policies, and splitting and coalescing
  - Placement choice (keeps fragmentation under control): choosing where in free memory to put block
    - Splitting blocks to satisfy smaller requests
    - Coalescing free blocks to yield larger blocks
  - An allocator algorithm implements a placement policy, motivated by a strategy to lower fragmentation
  - Strategy, policy, and mechanism
    - Strategy tries to exploit regularities in request stream or program behavior
    - Policy is implementable decision procedure for placing blocks into memory
    - Mechanism is algorithm and their data structures that implement the policy
  - Ideal strategy: put blocks where fragmentation won't occur later
  - Working strategies:
    - Avoid letting small long-lived objects prevent you from reclaiming larger contiguous free area
      - Best fit policy is very concrete
    - If you have a split block and potentially waste left overs, minimize size of wasted part
  - Complete policy: allocate objects in smallest block needed to hold them; when block sizes are equal, use lowest address

        - Example mechanism:
            - Use linear list or ordered tree structure to record addresses/sizes of free blocks
            - Tree search or list search to find the one our policy dictates
    - Often times, strategy/policy or policy/mechanism are not well defined
    - Splitting and coalescing support a range of placement policies
        - Allocator can split larger blocks into sub-blocks to fulfill requests
        - Allocator can coalesce, or merge, free blocks to form a single more useful large free block
        - Splitting and coalescing might waste efforts if they weren't really needed
            - Deferred coalescing: avoid splits/coalesces except occasionally to avoid fragmentation
- Internal and External Fragmentation
    - External: free blocks are available for allocation, but can't actually hold objects of requested size
        - Free blocks are too small OR allocator doesn't want to split large blocks into smaller ones
        - Remedy: coalesce adjacent free blocks so they can hold larger objects
    - Internal: large free block allocates an object, but it's bigger than needed
        - Wastes some space, causing internal fragmentation (since it's inside allocated block)
        - Remedy: split block and allocate part of it
        - Might occur due to implementation constraints OR as a way to prevent external fragmentation
- Basic (Allocator) Mechanisms
    - Sequential Fits (first, next, best, worst fit)
    - Segregated Free Lists (simple segregated storage/fits)
    - Buddy Systems
    - Indexed Fits
    - Bitmapped Fits
- Sequential Fits
    - Classic allocator algorithms are often based on one linear list holding free blocks of memory
    - Sequential fit algorithms usually use Knuth's boundary tag technique and doubly-linked list
    - Strategy and policy issues are important, since implementations might not scale well
    - Best Fit
        - Don't waste space by finding fragments as small as possible
            - If the fit is too good, our plan backfires and the space wasted becomes unusable
        - Generally exhaustive search, so this doesn't scale well
        - Generally good memory usage (scales well with balanced trees, etc.); bad worst case
    - First Fit
        - Search from the beginning for any block that's big enough; split if necessary
            - We end up with a lot of small "splinters" at the front, increasing search time
        - Many variations; LIFO frees fast, but allocation requires sequential search
        - Address-ordered first fit can encode adjacency of free blocks for less fragmentation
        - FIFO pushes freed blocks to the end to create a queue-like structure; better than LIFO as well
        - Acts like Best Fit over time
    - Next Fit
        - Use a roving pointer with First Fit to hold where the last search was satisfied
            - Next search begins at this point to save time and prevent splinters
        - Unfortunately, roving pointer cycles through memory regularly, interspersing objects throughout memory
        - Bad locality characteristics; may affect locality of allocated program also (scattering objects)
        - Causes more fragmentation than Best Fit or address-ordered First Fit