Advanced Techniques

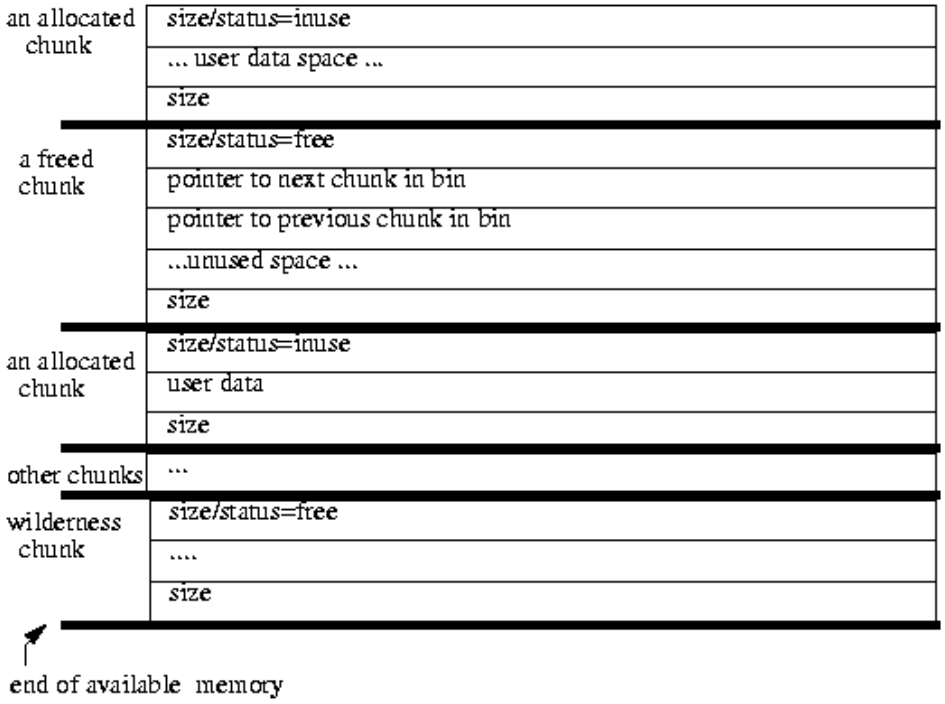
- [Ref: A Memory Allocator by Doug Lea. http://gee.cs.oswego.edu/dl/html/malloc.html]

- Also see Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, "Dynamic Storage Allocation: A Survey and Critical Review", in *International Workshop on Memory Management*, September 1995 for a comprehensive (if dated) survey paper.

- We will discuss the following enhancements.
  1. Boundary tags
  2. Binning
  3. Locality preservation
  4. Wilderness preservation
  5. Memory mapping
  6. Caching

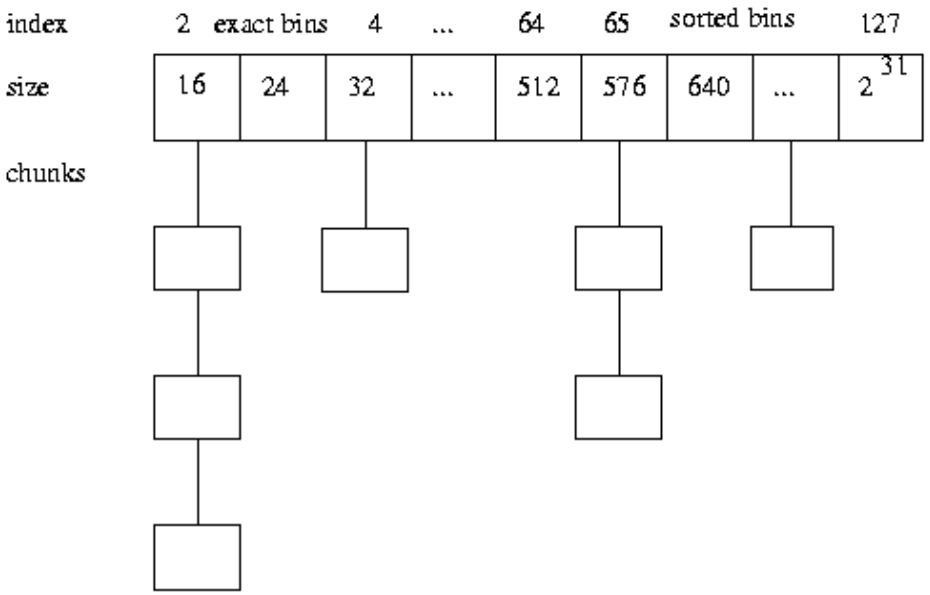
Technique #1: Boundary Tags

- Introduced by Knuth.
- Memory blocks (both free and allocated) contain size information fields at both ends (called headers and footers).
  - Allows for quick coalescing.
  - Allows for forward/backward traversal of *all* blocks starting from a given block.
- Possible optimization: Omit footers in allocated blocks.
  - Reduces overhead slightly.
  - Weakens error detection.



Technique #2: Binning

- Free blocks are maintained in bins, grouped by size.
  - 128 fixed-width bins, approximately logarithmically spaced in size.
  - Exact sizes for small objects.
  - Free blocks kept sorted by size within bins, with ties broken by an oldest-first rule.
  - Search for block selection is smallest-first, best-fit.
- Used in conjunction with coalescing.
- Leads to fixed management overhead per block.



Technique #3: Locality Preservation

- Objects allocated at about the same time by a program tend to have similar reference patterns and coexistent lifetimes.
  - Maintaining locality of such objects minimizes page faults and cache misses.
- If locality were the only goal, an allocator might always allocate each successive block as close to the previous one as possible.
  - However, this nearest-fit (often approximated by next-fit) strategy can lead to very bad fragmentation.
- A version of next-fit can be used only in a restricted context that maintains locality in those cases where it conflicts the least with other goals.
  - If a free block of the exact desired size is not available, the most recently split-off space is used (and re-split) if it is big enough; otherwise best-fit is used.
  - This restricted use eliminates cases where a perfectly usable existing block fails to be allocated, thus eliminating at least this form of fragmentation.

Technique #4: Wilderness Preservation

- The wilderness block represents the space bordering the topmost address allocated from the system.
  - Because it is at the border, it is the only chunk that can be arbitrarily extended (via `sbrk()`) to be bigger than it is.
- Simple strategy: Handle the wilderness block just like any other block.
  - While this simplifies and speeds up implementation, it can lead to some very bad worst-case space characteristics.
  - Among other problems, if the wilderness block is used when another available free block exists, the chances of a later request causing an otherwise preventable `sbrk()` increase.
- A better strategy uses a very simple fix.
  - Artificially treat the wilderness block as "bigger" than all others (since it can be made so, up to system limitations) and use it as such in a best-first scan.
  - This results in the wilderness block always being used *only if no other chunk exists*, further avoiding preventable fragmentation.

Technique #5: Memory Mapping

- In addition to extending general-purpose allocation regions via `sbrk()`, Linux supports system calls such as `mmap()/mfree()` that allocate and deallocate separate non-contiguous regions of memory for use by a program.
  - Requesting and returning a `mmap()`-ed block can further reduce downstream fragmentation, since a released memory map does not create a ``hole'' that needs to be managed.
  - However, because of built-in limitations and overheads associated with `mmap()`, it is only worth doing this in very restricted situations, e.g., the request is greater than a (large and dynamically adjustable) threshold size and the space requested is not already available in the existing arena, so it would have to be obtained via `sbrk()`.
- As this technique is not always applicable in most programs, one can also trim the main arena, which achieves one of the effects of memory mapping – releasing unused space back to the system.
  - When long-lived programs contain brief peaks where they allocate large amounts of memory, followed by longer valleys where the have more modest requirements, system performance as a whole can be improved by releasing unused parts of the wilderness block back to the system.
    - Releasing space allows the underlying operating system to cut down on swap space requirements and reuse memory mapping tables.
    - Often, `sbrk()` can be used with a negative argument to achieve this effect.
    - Should be attempted only if trailing unused memory exceeds a tunable threshold.

Technique #6: Caching

- Operations to split and to coalesce blocks take time. This time overhead can sometimes be avoided by using either of both of two caching strategies.
  - *Deferred coalescing*: Rather than coalescing freed blocks, leave them at their current sizes in hopes that another request for the same size will arrive soon. This saves a coalesce, a later split, and the time it takes to find a non-exactly-matching block to split.
  - *Pre-allocation*: Rather than splitting out new blocks one-by one, pre-split many at once. This is normally faster than doing it individually.
- The corresponding caching heuristics are easy to apply because the basic data structures in the allocator permit coalescing at any time.
  - The effectiveness of caching obviously depends on the costs of splitting, coalescing, and searching relative to the work needed to track cached blocks.
  - Less obviously, effectiveness depends on the policy used to decide when to cache versus when to coalesce.