Dynamic Memory Allocation

CPSC/ECE 3220

Mark Smotherman

Explicit Dynamic Memory Allocation Characteristics

- Objects are explicitly created and destroyed
  - Dynamically-allocated object lifetimes are not linked to normal scope rules
    - Longer than lifetime of a stack-allocated object (which is duration of a function call)
    - Shorter than lifetime of a global or static object (which is duration of process)
      - It’s not practical to globally allocate all the dynamic objects that may be needed
  - Allocated in special segment of memory called the heap
  - Object is born on call to malloc(), dies on call to free()
  - Does not use garbage collection
- Allocator knows the object size but usually not the data type
- Allocated memory blocks typically cannot be relocated
  - Multiple pointers could exist related to each block

Allocator Goals

- Space efficiency
  - Keep track of releases to reuse released memory for new requests
  - Minimize fragmentation (wasted space)
- Time efficiency
  - O(1) complexity where possible
- As usual, these goals can conflict

Allocation Patterns

- Block lifetimes are not random
  - Ramp – allocations throughout program lifetime without releases
  - Plateau – allocations, then lengthy usage, then releases
  - Peaks – bursty behavior and short object lifetimes
- Block sizes are not random
  - Zorn and Grunwald 1992 study of six allocation-intensive C programs (e.g., gawk, PCB channel router, perl) found that the top two sizes accounted for 53-93% of requests
  - See http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.5186
- So, the allocator can attempt to exploit patterns
  - Allocate related blocks contiguously
  - Can infer relationships
    - Similar allocation times
    - Same object sizes

Allocator Actions

- Allocate
  - Identify a free block
  - Split the block?
    - Exact size of request, or
    - Nearest power-of-2 in Buddy system
      - Results in internal fragmentation
- Release
  - Coalesce contiguous free blocks?
    - Immediately on release, or
    - Deferred (since small blocks may be immediately reallocated)

Data Structure Choices for the Set of Free Blocks

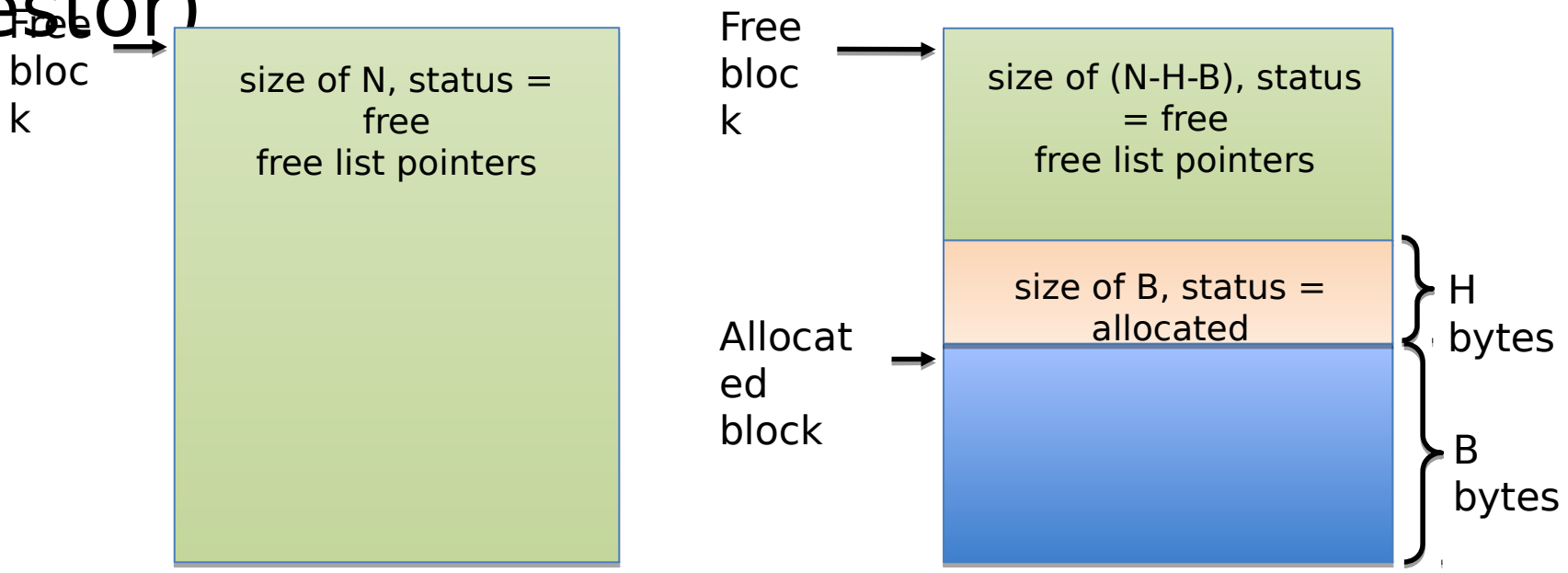
- Linked list of variable-size free blocks
  - Location
    - External list (but where to get the space for nodes in the free list as it gets longer?)
    - Integrated list (use the space within the free blocks to hold the links!)
  - Ordering
    - Sorted by address
    - Sorted by block size
    - LIFO
    - FIFO
- Array of linked lists (“segregated free lists”)
  - Multiple free lists, each with a fixed block size or a specified range of block sizes
- Tree
- Bitmap for fixed-size free blocks
- Simple free pointer for a region

Identifying a Variable-Length Free Block

- Sequential fit
  - Limited search
    - First fit – but small fragments will typically build up at front of list
    - Next fit – start next search where last search stopped
  - Exhaustive search
    - Best fit – try to leave the smallest fragments
    - Worst fit – try to leave the largest fragments
- Note that the array-of-lists approach can be similar to best-fit but without exhaustive searches

Header for an Allocated Block

- Some methods use a header or tag located above an allocated block of B bytes (invisible to requestor)



Header for an Allocated Block (2)

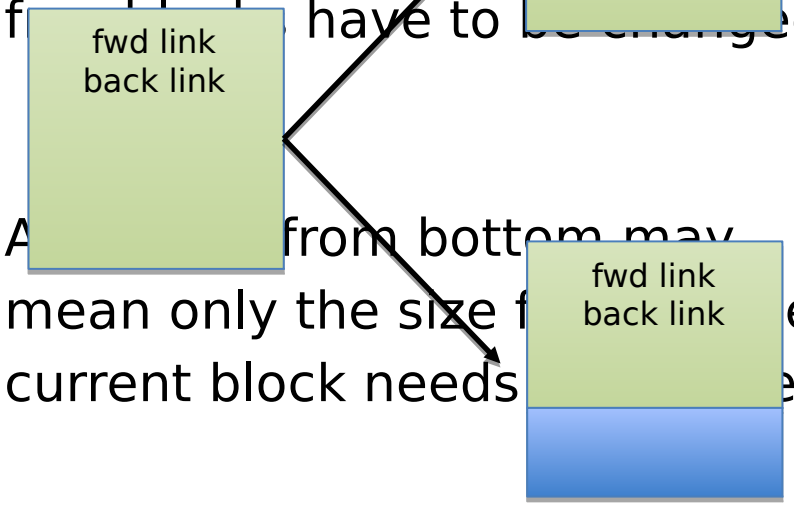
- Using a header means splitting (H+B) bytes out of a large free block, not just B bytes
- Boundary tag method uses both header and trailer
- Tags may include the status of the adjacent block in memory as well as the size and status of the current block

Splitting a Variable-Length Block for Exact Size

- From top or bottom of free block?

Allocating from top
Original in the cur
free block move
f
have to be
s in two other
have to be changed

Use the space inside a free block to hold free list links

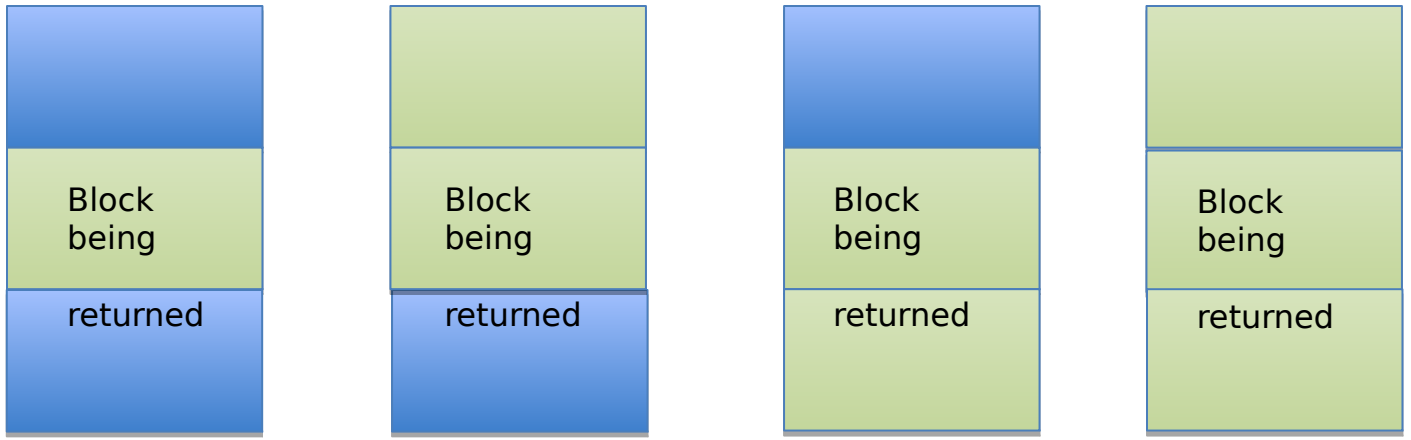


A from bottom may
mean only the size f e
current block needs e

Coalescing Variable-Length Free Blocks

- Four cases based on status of adjacent blocks

Neither is free Above is free Below is free Both are free



Must insert a new free block in the free list

Must merge two existing free blocks __ update an existing free block __/ in the free list

Boundary Tag (Knuth, 1962)

- Designed for variable-sized allocations, bursty patterns, and ease of coalescing
- Single free list, doubly-linked, LIFO or FIFO order
- Header and trailer words in each block
  - Allows O(1) access to adjacent blocks to inspect their status and guide coalescing
- void *allocate(size_t size)
  - Search free list (e.g., first-fit)
  - If a split is needed, set up two new tags within the identified free block
- void release(void *ptr)
  - Compare tags of returned block with tags in adjacent blocks in memory
  - If no coalescing possible, add new free block to free list (front or back) – O(1)

Buddy System (Markowitz, 1963)

- Designed for variable-sized allocations, bursty patterns, and ease of coalescing
- Table of free list head pointers, with each list containing free blocks of 2^n in size
  - there are limits on the high and low values for n
- void *allocate(size_t size)
  - Round up a request size to 2^n size
  - If free block of that size is not available, identify a larger block and recursively split it in half until a suitably-sized block is available to satisfy the request
- release(...)
  - Design choice 1: pass back both address and size
    - No header/tag needed for allocated block
    - Flip bit for rounded size in the returned block address to construct the address of the “buddy” block
    - Search in free list of that size for the buddy; if found, coalesce and recursively repeat
  - Design choice 2: pass back only address
    - Use a header word with status and size to provide information for coalescing
    - Alternatively, use a hierarchical set of bitmaps
- Later extensions are Fibonacci buddy, weighted buddy, and double buddy

BSD 4.2 malloc() (Kingsley, 1982)

- Designed to be extremely fast
  - Tradeoff is fragmentation (internal as well as unused blocks of wrong size)
- Array of 30 free lists, each of specified size $2^{(i+3)}$ (i.e., 8 bytes to 4 GiB)
- void *malloc(size_t size)
  - Round up the requested size, plus a 4-byte header, to a power-of-2
  - No splitting of blocks from free lists with larger block size
  - Instead, if the appropriate free list is empty, attempt to get the block via mmap()
    - For block sizes less than a page, request a page from mmap() and then divide the page into multiple blocks
- void free(void *ptr)
  - Index field in header determines the free list on which the block is placed in LIFO order
  - No coalescing
- See ftp://web.mit.edu/freebsd/head/libexec/rtld-elf/malloc.c
- BSD 4.3 kernel memory allocator paired Kingsley's power-of-2 approach for 16 bytes to 2 KiB with another approach using multiples of page size for requests larger than 2KiB

dlmalloc() (Doug Lea, 1987)

- Designed for balance across a number of goals; various versions since 1987
- Includes heuristics, e.g, trying to allocate a “designated victim”, caching
- Combination of three techniques
  - Table of free lists for fixed-size blocks with payload sizes 8, 16, 24, …, 248 bytes (“small bins”)
    - MRU order for locality
    - Splitting when a particular free list is empty
  - Table of free lists supporting a bitwise trie for variable-sized blocks with payloads of 256 bytes to 256 KiB (“large bins”)
    - LRU order to reduce fragmentation
    - Blocks arranged in descending size order
  - Use mmap() for requests greater than 256 KiB
- Header contains size and status of current block as well as status of adjacent block
- Trailer is written into free blocks, containing size of current block
- See ftp://gee.cs.oswego.edu/pub/misc/malloc.c

Region / Arena (Hanson, 1988)

- Designed for variable-sized allocations where related groups of blocks are released all at once
- A region consists of a linked list of large “arenas”
  - No need to keep a free list within an arena or to use headers in the allocated/free blocks
  - Instead, rely on a simple free pointer in each arena
- void *allocate(region_t region, size_t size)
  - If there is enough free space in the current arena, make a copy of the free pointer value to return and increment the free pointer by the size amount
  - Otherwise, add a new arena
- void release(region_t region)
  - Release all but first arena from the region and reset the free pointer in the first arena
  - Cannot release an individual block
- See ftp://ftp.cs.princeton.edu/techreports/1988/191.pdf

Slab Allocation (Bonwick, 1994)

- Designed for fixed-size, pre-initialized allocations with bursty patterns in an operating system kernel
  - E.g., a control block containing a lock can be initialized once and later used and reused without incurring create/destroy lock overhead on each reuse
- “Cache” of fixed-sized objects
  - Multiple “slabs” in each cache, each capable of containing multiple objects
  - Different caches for different objects
- Two-level hierarchy of bitmaps to speed search
  - Root-level: empty, partial, full bitmaps (each with one bit per slab)
  - Second-level: bitmap of free blocks within slab (one bit per block)
- void *kmem_cache_alloc(struct kmem_cache *cp, int flags)
- void kmem_cache_free(struct kmem_cache *cp, void *ptr)
- See http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.4759

GNU glibc malloc()

- Derived from ptalloc(), which was derived from dlmalloc()
- Uses arenas to support multithreaded applications
  - Switch to a separate arena if the malloc() call encounters an arena that is locked by another thread (using pthread_mutex_trylock())
  - Total arenas typically limited to eight times the number of processor cores
- See https://sourceware.org/glibc/wiki/MallocInternals

GNU glibc Obstacks – Object Stacks

- Can define separate obstacks
  - Each obstack is a linked list of “chunks”
  - Default chunk size is 4 KiB
- void *obstack_alloc(struct obstack *obstack_ptr, int size)
  - You must specify how memory is obtained, e.g., using malloc()
- void obstack_free(struct obstack *obstack_ptr, void *object)
  - You must specify how memory is freed, e.g., using free()
  - Nested behavior - within a given obstack, freeing one object automatically frees all other objects allocated more recently
  - Passing a NULL object pointer frees all objects in that obstack

General References

- Paul Wilson, *et al*., “Dynamic Storage Allocation: A Survey and Critical Review,” Proc. Intl. Workshop on Memory Management, Scotland, Sept. 1995
- Valtteri Heikkilä, “A Study on Dynamic Memory Allocation Mechanisms for Small Block Sizes in Real-Time Embedded Systems,” University of Oulu, Department of Information Processing, Science Master's Thesis, Dec. 2012