KERNEL MEMORY ALLOCATORS, PART 1

CS124 – Operating Systems

Winter 2016-2017, Lecture 16

Kernel Memory Allocators

- Kernels often keep complex data structures in memory…
  - Process control blocks, file descriptors, filesystem inodes, …
- Also, kernels need buffer space for device interactions…

- The kernel must include a dynamic memory allocator to service various allocation and deallocation requests
- Two kinds of memory allocation in the kernel:
  1. Add pages to the virtual address space of the kernel, or to the virtual address space of a specific user process
  2. Allocate a chunk of memory for use by the kernel (e.g. for a process control block, for use by a system call, etc.)
- Second kind is handled by the **kernel memory allocator**

Kernel Memory Allocators (2)

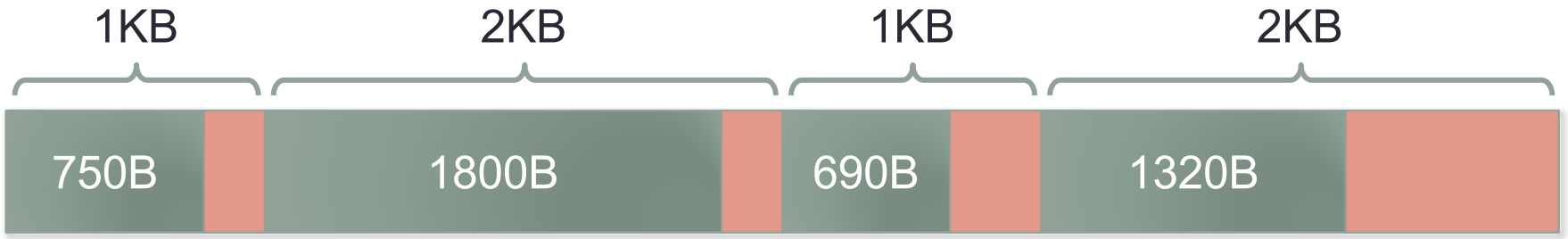
- Most parts of the kernel use the kernel allocator, so a primary requirement is that it must be fast
- Kernel memory allocator is called from interrupt context as well as process context
- Example: kernel allocator called from interrupt context, when the system has little available memory
  - System can page out some unnecessary pages to free up room…
  - …but that would involve putting the requestor to sleep ☹
- Generally, kernel allocators have at least two modes:
  - Allocation request is in interrupt context – cannot sleep; allocator may fail to satisfy request
    - Kernels try to avoid failing in this situation by having extra space around
  - Allocation request is in process context – can sleep; allocator will eventually succeed (barring disk-full errors, etc.)

Memory Fragmentation

- Kernels need to use memory as efficiently as possible
  - Don’t want the OS to impose a substantial resource overhead on the overall system
- Another primary concern is **memory fragmentation**
- Two kinds of fragmentation issues can occur:
- **External fragmentation** is when allocated blocks and free blocks become interspersed
- Example: allocate blocks of varying sizes
  - Then the two 1KB blocks are freed
  - 2KB is free, but not in a contiguous memory region
  - A 2KB allocation request will fail

Memory Fragmentation (2)

- **Internal fragmentation** is caused by constraints imposed by the allocation strategy itself
- Example: allocator only hands out memory in blocks that are multiples of 1KB in size
- Allocation requests:
  - 750 bytes (block size = 1024 bytes; waste = 274 bytes)
  - 1800 bytes (block size = 2048 bytes; waste = 248 bytes)
  - 690 bytes (block size = 1024 bytes; waste = 334 bytes)
  - 1320 bytes (block size = 2048 bytes; waste = 728 bytes)
- Total loss due to internal fragmentation: 1584 bytes



Kernel Allocator and Virtual Memory

- When the kernel needs to increase its available memory size, it receives memory in units of virtual memory pages
  - e.g. on IA32, virtual pages are 4KiB in size
- A common scenario:
  - Allocator tries to satisfy request from existing free space, but can't
  - Allocator requests one or more new pages from the virtual memory system
- Want to maximize the use of each virtual page to minimize paging, and also to minimize internal fragmentation

- Good kernel memory allocators are always tuned to work efficiently with the virtual memory system

Kernel Allocation Characteristics

- Kernel allocation patterns are generally well understood by kernel developers
  - Can design the allocator to satisfy requests efficiently
- Generally, allocation requests have a maximum size
  - e.g. for larger I/O buffers
  - Larger requests tend to be less common; smaller requests are extremely common
- Also common to have many requests for a particular size
  - e.g. for process control blocks, file descriptors, etc.
  - Leads to an **allocate-free-reallocate** usage pattern:
    - A specific kind of object is allocated, used, then released
    - Soon, the OS will need to reallocate that kind of object again
- Better kernel memory allocators are designed to maximize performance in the context of these patterns

Resource Map Allocator

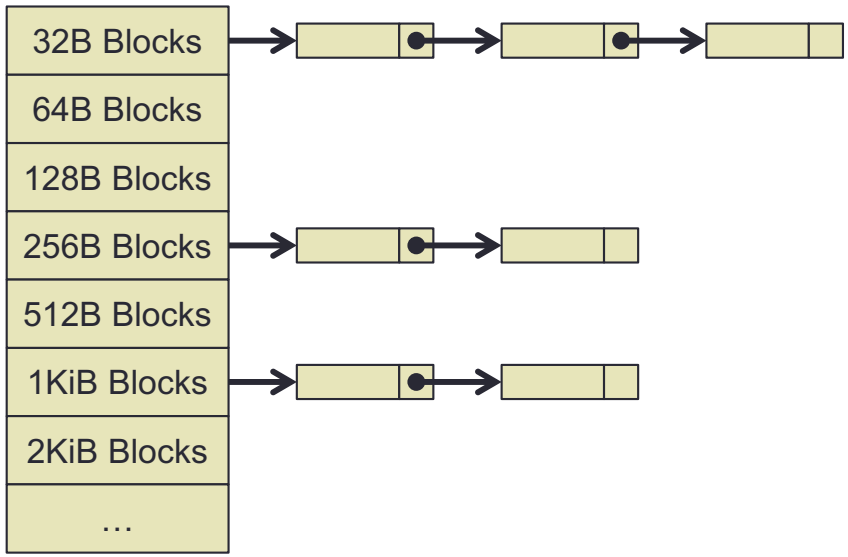
- Simplest kernel allocator is the **resource map allocator**
  - (Also called **sequential fits** or **explicit free list** allocator)
- Maintains a simple list of free regions in the memory pool
  - <*base*, *size*> pairs, where *base* is the starting address
  - (Allocated regions have a header to record their size)
- If list is maintained in order of increasing *base* address:
  - Easy to coalesce adjacent free regions
- If list is maintained in order of increasing *size* value:
  - Faster to find a region of the appropriate size for an allocation
- Benefits:
  - Very simple to implement
  - Almost no internal fragmentation at all
  - Can implement first-fit, best-fit, next-fit policies very easily

Resource Map Allocator (2)

- Drawbacks:
- Very prone to external fragmentation issues
  - External fragmentation becomes an issue as a program runs for a longer time…
  - The kernel runs as long as the system is in use
- Much slower than other kernel allocators
  - Even with optimizations, allocation/deallocation can be very slow
  - (e.g. if resource map is maintained in some particular order…)
- Resource map allocator doesn't take virtual memory system into account hardly at all
  - Some systems have used resource map allocator at the page level, not at the level of individual allocations

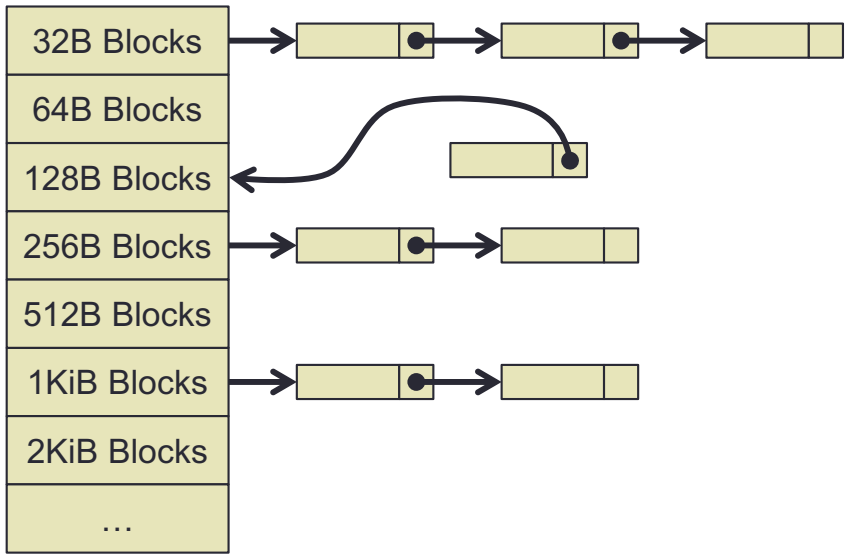
Power-of-Two Free List Allocators

- **Power-of-two free list allocators** round up allocation requests to the next power of two
  - Or, if a request spans multiple virtual pages, rounds up to a whole number of virtual pages
  - e.g. a request for 58 bytes is stored in a block of 64 bytes
- Instead of maintaining one free list, keep an array of free lists, each for a different size
  - Free blocks in each list hold a pointer to next block in the list
- Allocation is usually very fast:
  - Round up request size to nearest power of 2 (or whole # of pages)
  - Go to corresponding list and get the first block of that size



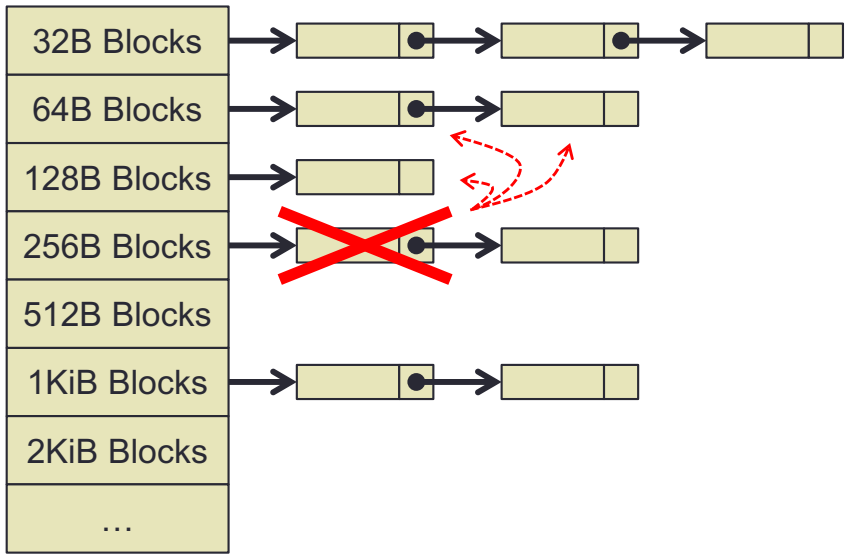
Power-of-Two Free List Allocators (2)

- Freeing is usually equally fast:
  - Find the free list that the block being deallocated should be stored into, and add it to that list
- Problem: how do we tell what list corresponds to the allocated block? (i.e. how do we know the block's size?)
- A simple solution:
  - One word of allocated blocks is used to store the free-list that the block corresponds to
- Problem: now blocks can only hold slightly less than their size
  - If kernel requests sizes that are powers of 2, 50% of the block's space will be lost!
  - (In practice, this happens often ☹)



Power-of-Two Free List Allocators (3)

- Many options to handle the case when blocks of the required size aren't available
  - e.g. a request of 48 bytes, requiring a 64 byte block
- A very common solution: take a larger block, and split it repeatedly into smaller blocks
  - e.g. to get a 64B block, split a 256B block into a 128B block and two 64B blocks
- Problem:
  - Splitting large blocks makes it more difficult to coalesce smaller blocks into larger ones
  - Similarly, reclaiming entire pages becomes difficult; page may have blocks across multiple free lists

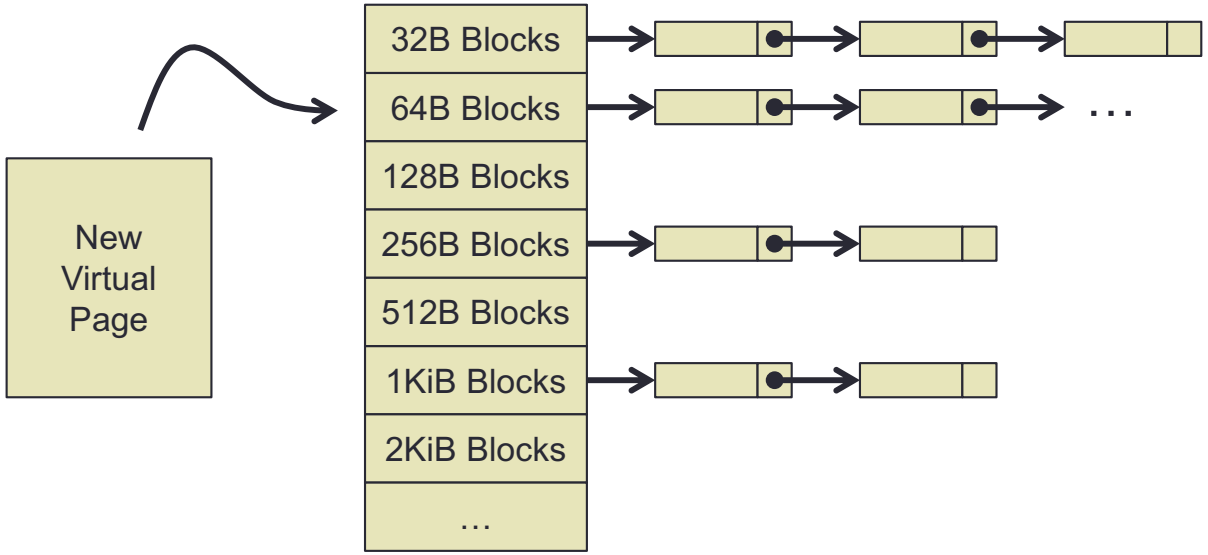


Power-of-Two Free List Allocators (4)

- Benefits:
  - Very fast allocation and deallocation
  - Very good at reusing previous allocations of a given size
  - Very resilient against external fragmentation – power-of-two blocks pack together into pages extremely well
- Drawbacks:
  - Suffers badly from internal fragmentation issues
    - Research results show an expected loss from internal fragmentation around 28% (e.g. Buddy Systems by Peterson and Norman, 1977)
  - Internal fragmentation is magnified by the fact that allocated blocks lose a word to remember their corresponding free-list
  - Coalescing and page reclamation tends to be very difficult without imposing additional time or space overhead
    - e.g. sort free-lists by address to facilitate coalescing, or use boundary tags, etc.

Power-of-Two Free List Allocators (5)

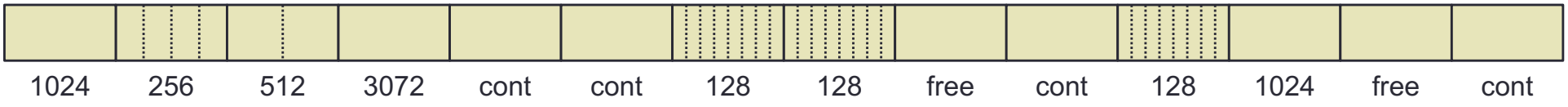
- Another option when more free blocks are needed:
  - Request a new virtual page, and split the entire page into blocks of the same size
  - Example: for 48-byte request, get a new 4KiB virtual page, split it into 64 64B blocks, and add all blocks to the 64B free-list



- If all blocks in a virtual page are the same size, why waste space recording what free-list each block corresponds to?
  - Can record this detail at the page level instead

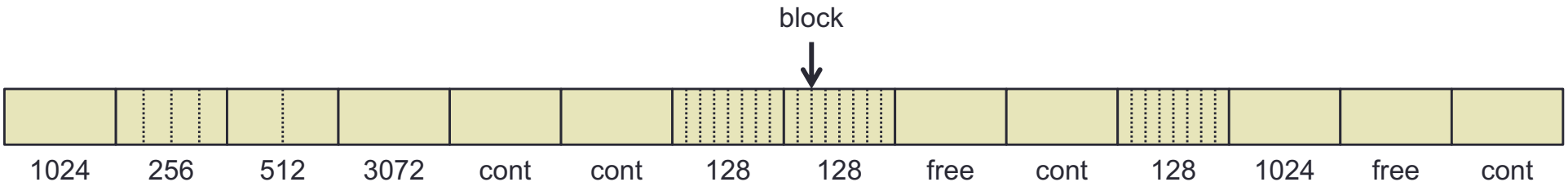
The McKusick-Karels Allocator

- The McKusick-Karels allocator is a variant of power-of-two allocator that maintains block sizes at the page level
  - Used in the 4.3BSD UNIX kernel
- Major benefit: allocated blocks no longer lose a word to the block size (or associated free-list pointer)
- Instead, the kernel allocator maintains an array of details for every page in the allocator's pool
  - 4.3BSD allocator calls this **`kmemsizes`**
  - Allocator also manages a free-list per block size, as before
- Array entries specify the allocation block size, or "free," or a "continuation of the previous allocation"
- Example from McKusick/Karels paper (1KiB page size):



The McKusick-Karels Allocator (2)

- Easy to find a block's size from its address: just access array-entry corresponding to page number of block
  - *entry* = *address* **div** *pagesize*
  - (since *pagesize* is a power of 2, can do this with a shift-right)
- Example: block address = 7424, page size = 1024
  - 7424 **div** 1024 = 7
  - **`kmemsizes[7]`** = 128, so the block size is 128 bytes
- Can store other details in page-level metadata, e.g. how many blocks in a page are allocated
  - Allows pages to be released very easily



The McKusick-Karels Allocator (3)

- The McKusick-Karels allocator is better than the simple power-of-two allocator, but still not great
- Internal fragmentation is reduced by factoring block-size details out of individual blocks, and into page-level details
  - But, still high due to the constraint of power-of-two block sizes

Binary Buddy Allocators

- **Buddy allocators** are similar to the previous allocators
- **Binary buddy allocators** always work with regions that are a power of 2 in size
  - (Also Fibonacci buddy allocators, weighted buddy allocators, etc.)
- Every block is of a particular **size order** (or **size index**)
  - The minimum size block is order 0, and is a power of 2
  - A block of order i has size (order 0 block size)$\times 2^i$
- Example: minimum block size = 32 bytes
  - Order 0 blocks are $32 \times 2^0 = 32$ bytes
  - Order 1 blocks are $32 \times 2^1 = 64$ bytes
  - Order 2 blocks are $32 \times 2^2 = 128$ bytes
  - etc.
- As before, a free list is maintained for each block size

Binary Buddy Allocators (2)

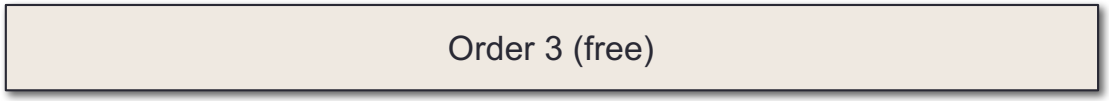
- Initially, the entire memory pool is a single free block of a maximum size order
  - Maximum is based on the memory pool size
- Example: order-0 block size = 32 bytes, pool size = 1MiB
  - Maximum order = 15; $32 \times 2^{15} = 1048576$ bytes
- A block of a given order i can be split into two adjacent blocks of order i – 1
  - These blocks are **buddies** of each other
  - Note: not all adjacent blocks of the same order are buddies!
- A free block of a given order i can be coalesced with its buddy block, if it is also free and of the same order
  - Forms a new block of order i + 1

Binary Buddy Allocators (3)

- When an allocation request is made:
  - As before, size is rounded up to nearest power of 2
  - The size order is computed
  - If a free block of the required size order is available, return it
- Otherwise, find the next highest size order that has an available free block
  - Required size order = a, smallest available size order = b, b > a
  - While b > a:
    - Split a free block of order b in half; new free blocks are now order b – 1
    - Add new blocks to free-list for order b – 1
    - Decrease b by 1
  - When loop is complete, there should be a free block of size-order a
  - Return the free block of the required size order

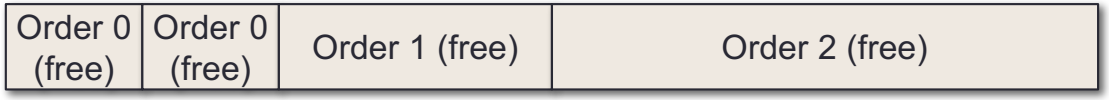
Binary Buddy Allocators (4)

- Example: order-0 block size = 32 bytes, max order = 3
- Initial state of memory pool:

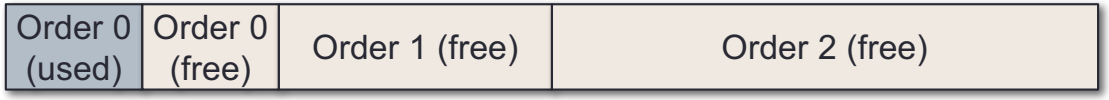


- Allocate a 20-byte block
  - Round up to 32 bytes, order 0
- No free block of that size
  - Split order-3 block into two order-2 blocks

  - Repeat with left block until we have an order-0 block

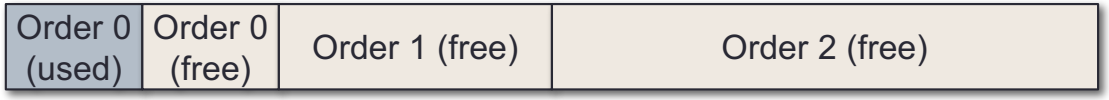


  - Finally, use an order-0 block for the allocation request



Binary Buddy Allocators (5)

- Memory pool state:



- Allocate a 40-byte block
  - Round up to 64 bytes, order 1
  - Already a free block of that size! Allocation satisfied immediately

- Allocate a 60-byte block
  - No more 64-byte blocks left. Must split order-2 block.

  - Finally, satisfy the allocation request

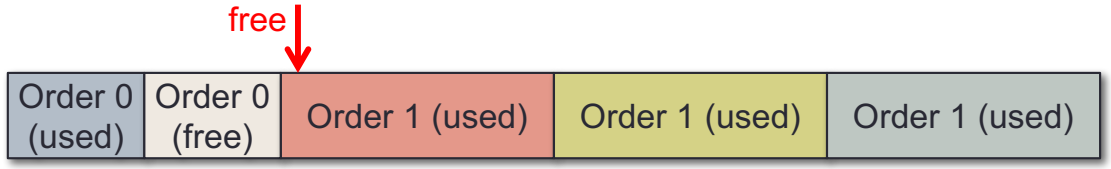
Binary Buddy Allocators (6)

- When a block is freed:
  - Mark the block as “free”
  - Set the “current block” to be the newly freed block
  - While the current block is not the highest order block in the pool:
    - (a = the current block’s order)
    - Find the block’s buddy
    - If block’s buddy is the same order as the current block, and is free, then coalesce the block and its buddy to form a new free block of order a + 1
    - Set the “current block” to be the newly coalesced free block

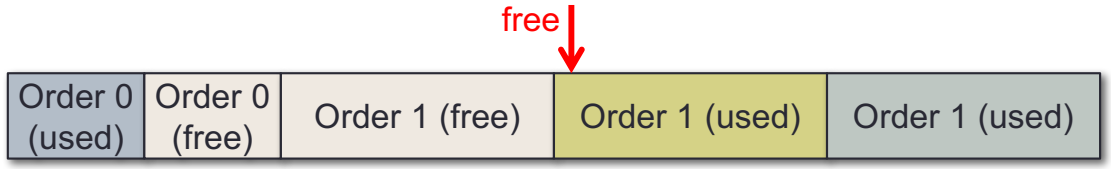
- To reiterate:
  - A block may only be coalesced with its buddy (i.e. split from the same parent block of the next higher size-order)

Binary Buddy Allocators (7)

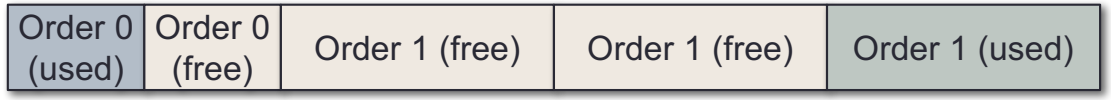
- Current memory pool state:



- Free first order-1 block
  - Doesn't have a buddy of the same size, so can't coalesce

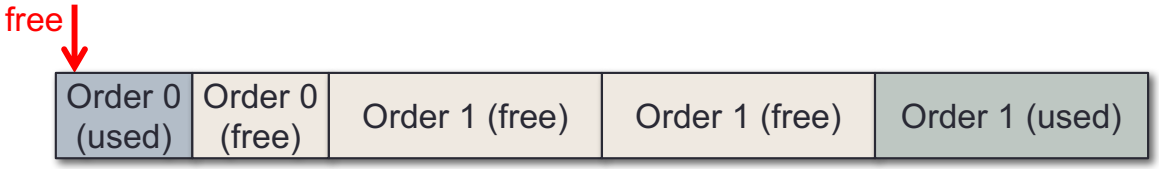


- Free second order-1 block
  - This block's buddy is to its right, not its left
  - Block's buddy is not free, cannot coalesce

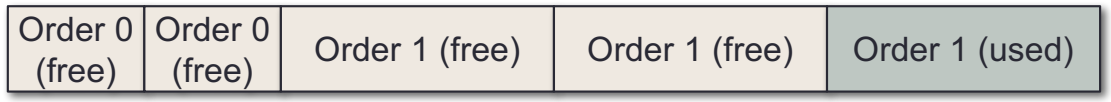


Binary Buddy Allocators (8)

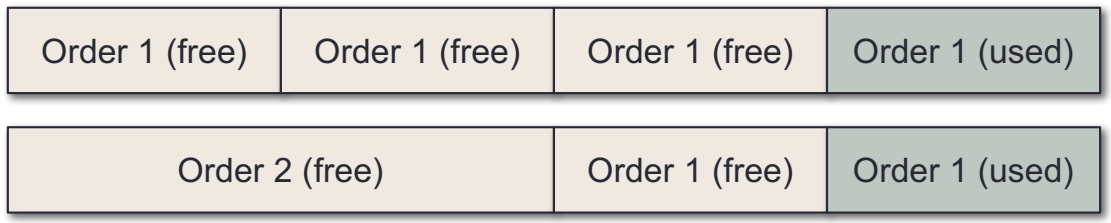
- Current memory pool state:



- Free first order-0 block
  - This block has a buddy of the same size order, and it's free!



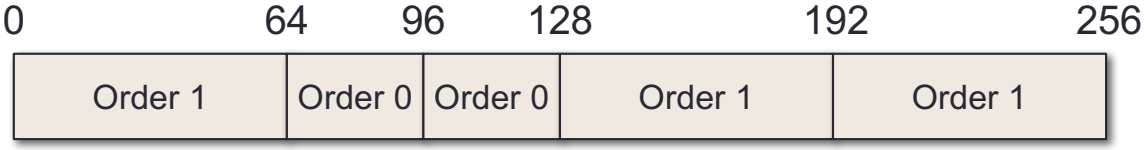
  - Coalesce adjacent buddy blocks until we can't continue



- Final state: one free order-2 block, one free order-1 block

Binary Buddy Allocators (9)

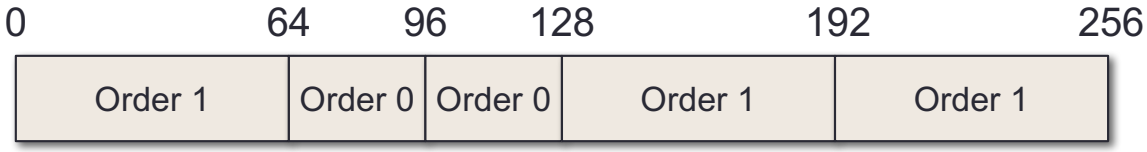
- Buddy allocator has an interesting characteristic:



- Can easily find a block's buddy by XORing the block's offset from the start of the memory pool, with the block's size
- Example: find buddy of second order-0 block
  - Offset of this block is 96 (0110 0000), size is 32 bytes (0010 0000)
  - XOR two values together, get 0100 0000 = 64 for buddy address
- Each bit in a block's offset from start of memory pool indicates whether it is the left or right buddy at each level
  - Offset 96 = 0110 0000
  - Topmost 0-bit indicates block is in left half of order-3 block
  - Second 1-bit indicates block is in right half of order-2 block
  - Third 1-bit indicates block is in right half of order-1 block

Binary Buddy Allocators (10)

- Buddy allocator has to keep track of details for each block



  - Allocated / free flag for the block
  - Size order of the block
- Can employ same technique as McKusick-Karels allocator
  - Maintain an array of details, at a resolution of minimum size order
    - e.g. for above memory pool, use 8 entries to track block details
  - Makes it easy to determine if a block's buddy is free or allocated, and also the size order of the buddy block

Binary Buddy Allocators (11)

- Buddy allocators are fast, nearly as fast as McKusick-Karels allocator
- Additionally, can coalesce space very easily
  - Coalescing is the extra time overhead of buddy allocators
- As with other power-of-two-based allocators, internal fragmentation is still a huge problem

- Frequently see buddy allocators employed at a virtual page level (i.e. for allocating groups of virtual pages)
  - Within each virtual page, other allocation mechanisms are used

Binary Buddy Allocators (12)

- Problem: repeated coalescing during deallocation is slow
- Common memory usage pattern: allocate-free-reallocate
  - Allocators usually want to retain free objects of a particular size for later use anyway…
  - Doesn't play well with buddy allocators that coalesce immediately
- Can choose to **defer coalescing** until needed
  - ("deferred coalescing," aka. "lazy coalescing")
- One strategy: identify when a specific free list has grown particularly long, and/or hasn't been used recently
  - Perform coalescing on blocks in that free-list until the number of free blocks is more reasonable

Next Time

- Continue discussion of kernel memory allocators with more advanced allocation mechanisms