Malloc

CS 241

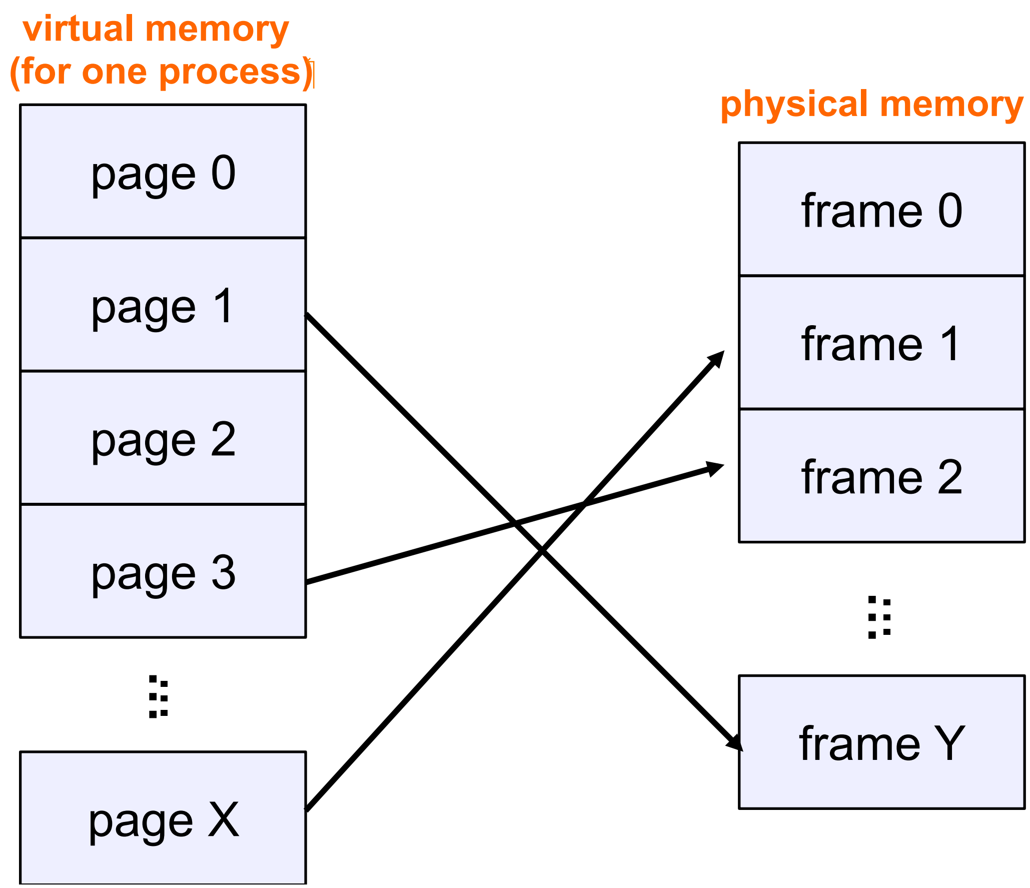
February 3, 2012

Announcements

- There is only one announcement today

Review: Paging

- OS solves the external fragmentation problem by using **fixed-size chunks** of virtual and physical memory
  - Virtual memory unit called a **page**
  - Physical memory unit called a **frame** (or sometimes **page frame**)



Definitions

- External fragmentation
  - Unused chunks of memory **between** allocated chunks
  - Can't use for large contiguous allocations

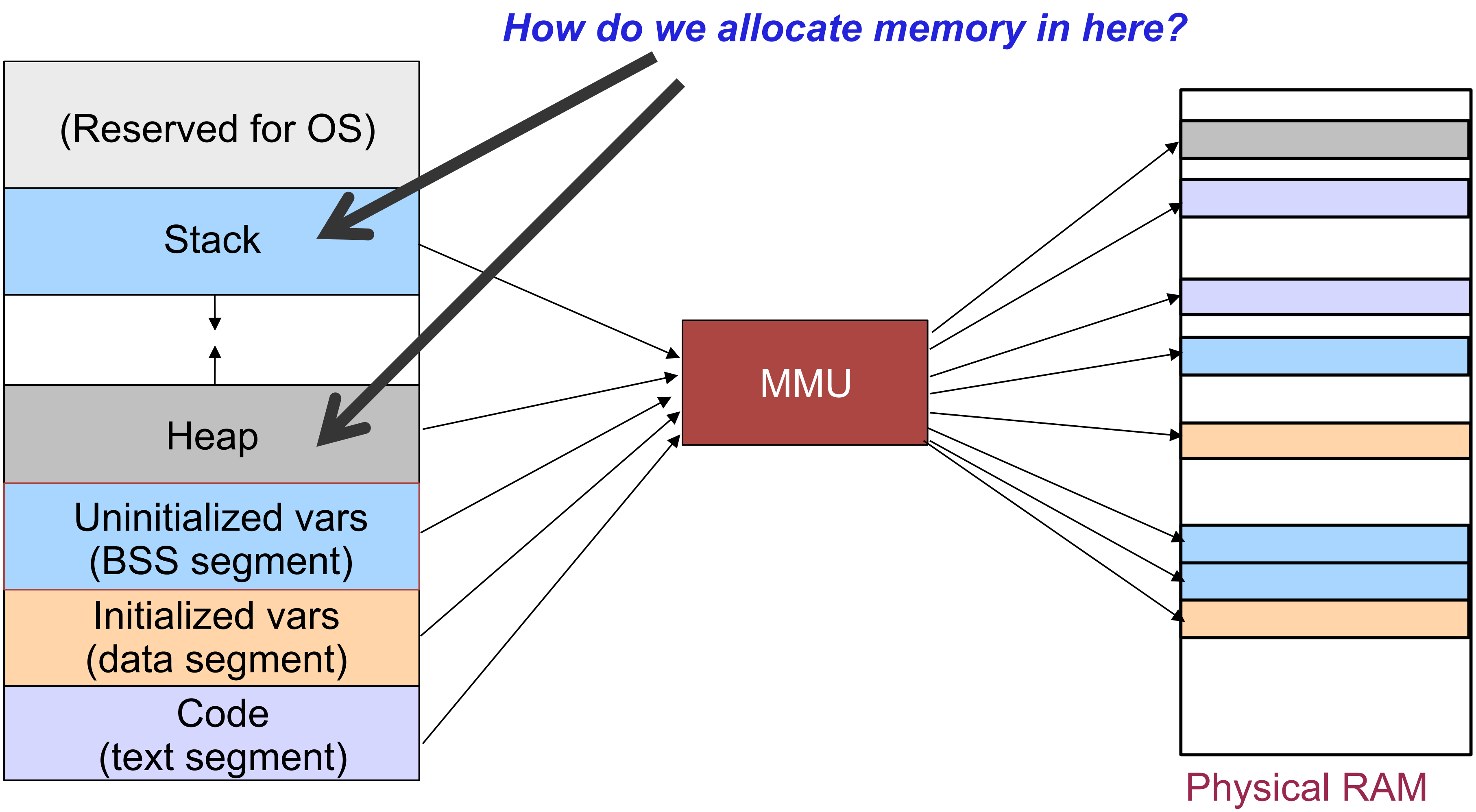
- Internal fragmentation
  - Unused memory **within** allocated regions
  - Because we allocated more than the requested size

- How does paging affect these?
  - Zero external fragmentation: all requests and fragments are the same size
  - Some internal fragmentation: requested size gets rounded up to next integer multiple of page size

Review: Advantages of Paging

- Simplifies physical memory management
  - OS maintains a free list of physical page frames
  - To allocate a physical page, just remove an entry from this list
- No external fragmentation!
  - Virtual pages from different processes can be interspersed arbitrarily in physical memory
  - No need to allocate pages in a contiguous fashion
- Allocation of memory can be performed at a (relatively) fine granularity
  - Only allocate physical memory to those parts of the address space that require it
  - Can swap unused pages out to disk when physical memory is running low
  - Idle programs won't use up a lot of memory (even if their address space is huge!)

Is paging enough?



Memory allocation w/in a process

- What happens when you declare a variable?
  - Allocating a page for every variable wouldn't be efficient
  - Allocations within a process are much smaller
  - Need to allocate on a finer granularity

- Solution (stack): stack data structure (duh)
  - Function calls follow LIFO semantics
  - So we can use a stack data structure to represent the process's stack – no fragmentation!

- Solution (heap): **malloc**
  - This is a much harder problem
  - Need to deal with fragmentation

Challenges of heap allocation

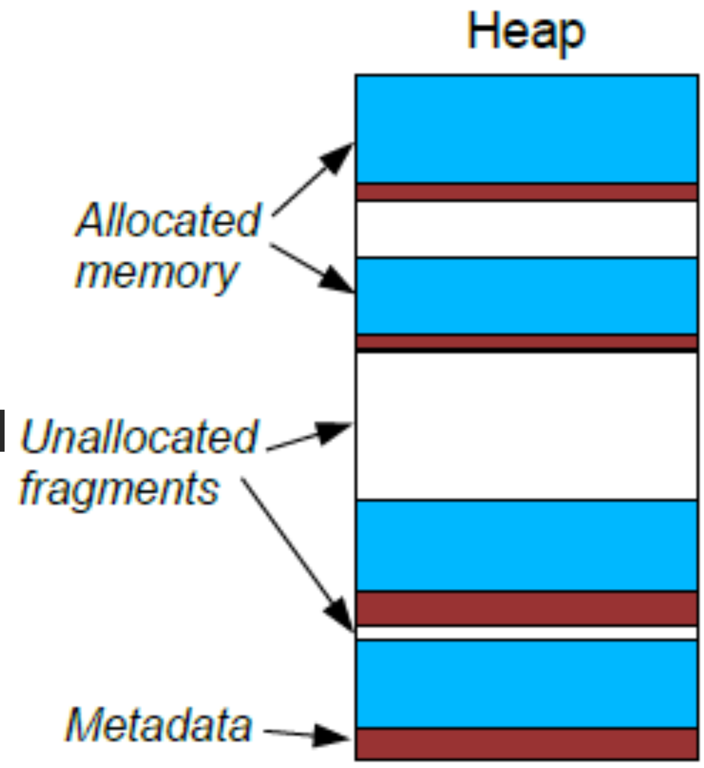
- Can’t control number or size of requested blocks
- Must respond immediately to all allocation requests
  - i.e., can’t reorder or buffer requests
- Must allocate blocks from free memory
- Must align blocks so they satisfy all alignment requirements
  - 8 byte alignment for GNU malloc (libc malloc) on Linux boxes
- Can only manipulate and modify free memory
- Can’t move the allocated blocks once they are allocated
  - i.e., compaction is not allowed (why not?)

Goal 1: Speed

- Want our memory allocator to be fast!
  - Minimize the overhead of both allocation and deallocation operations.
- Maximize throughput: number of completed alloc or free requests per unit time
  - E.g., if 5,000 malloc calls and 5,000 free calls in 10 seconds, throughput is 1,000 operations/second.
- A fast allocator may not be efficient in terms of memory utilization
  - Faster allocators tend to be “sloppier”
  - E.g., don’t look through every free block to find the perfect fit

Goal 2: Memory Utilization

- Allocators usually waste some memory
  - Extra metadata or internal structures used by the allocator itself
  - (example: keeping track of where free memory is located)
  - Chunks of heap memory that are unallocated (**fragments**)
- **Memory utilization =**
  - The **total amount of memory allocated to the application** divided by the total **heap size**
- Ideal: utilization = 100%
- In practice: try to get close to 100%



Fragmentation

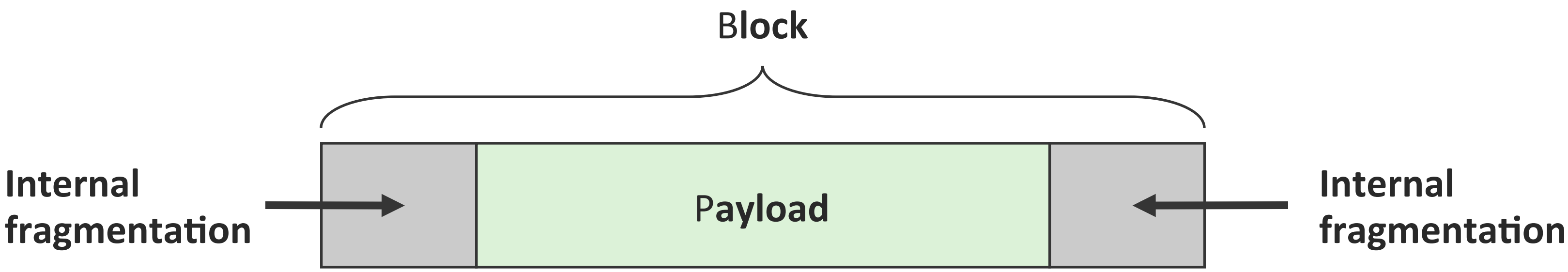
- Poor memory utilization caused by *fragmentation*
  - ***internal*** fragmentation
  - ***external*** fragmentation

- We saw: OS encounters fragmentation when allocating memory to processes

- Now: malloc encounters fragmentation when allocating memory to applications

Internal fragmentation

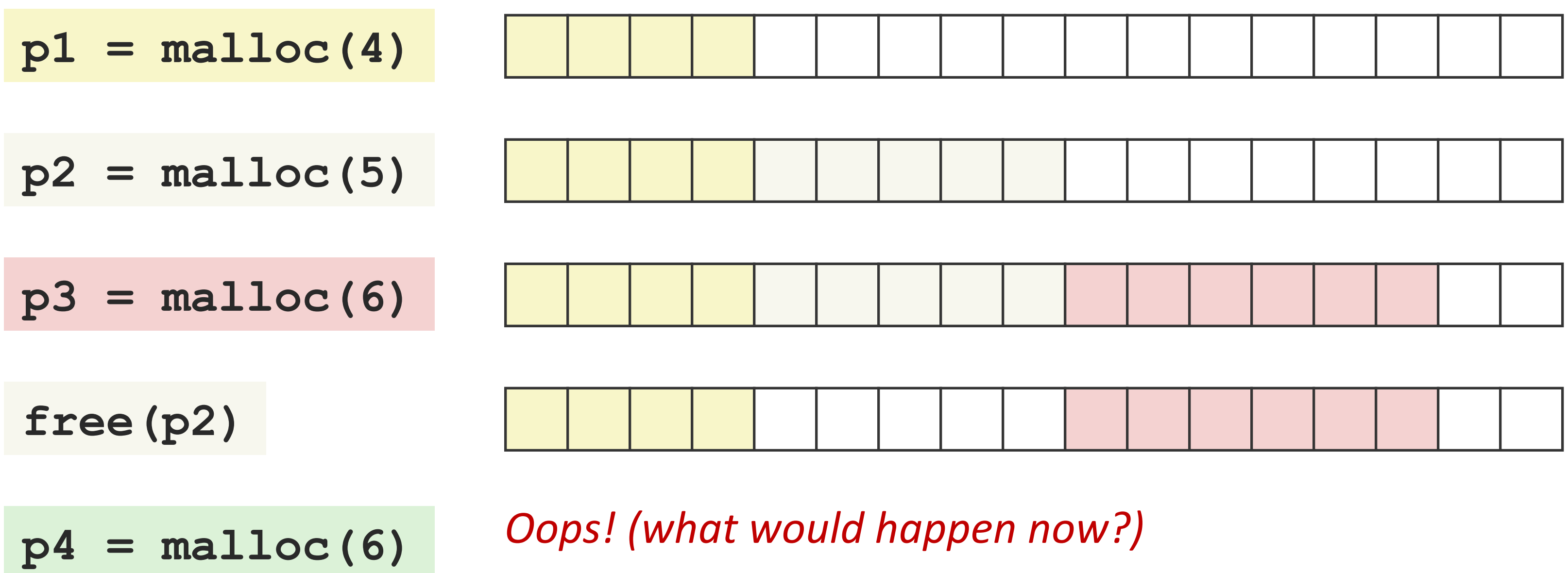
- For a given block, *internal fragmentation* occurs if payload is smaller than block size



- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions
    (e.g., to return a big block to satisfy a small request)

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough



- Depends on the pattern of future requests
  - Thus, difficult to plan for

Conflicting performance goals

- Good throughput and good utilization are difficult to achieve simultaneously

- A fast allocator may not be efficient in terms of memory utilization
  - Faster allocators tend to be “sloppier” with their memory usage.

- Likewise, a space-efficient allocator may not be very fast
  - To keep track of memory waste (i.e., tracking fragments), the allocation operations generally take longer time

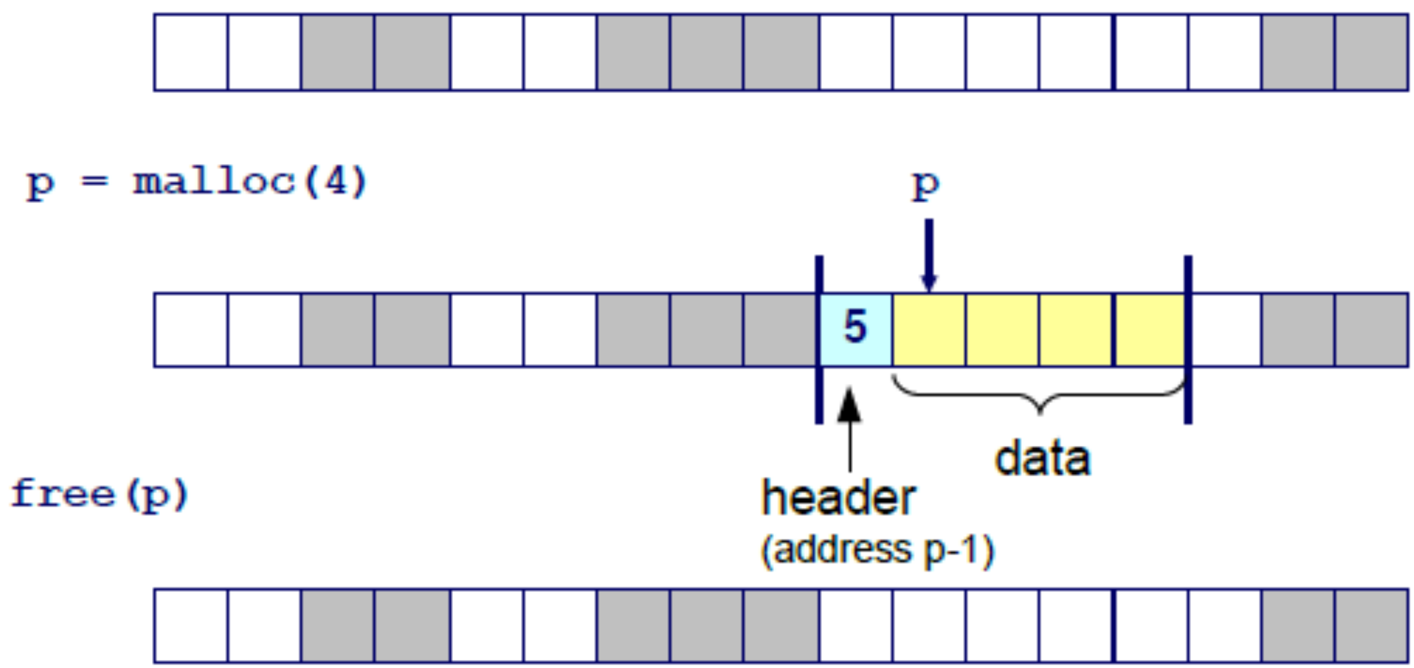
- Trick is to balance these two conflicting goals

Implementation Issues

- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a memory block that is smaller than the free block it is placed in?
- How do we pick which free block to use for allocation?

Knowing how much to free

- Standard method
  - Keep the length of the block in the header preceding the block
  - Requires an extra word for every allocated block

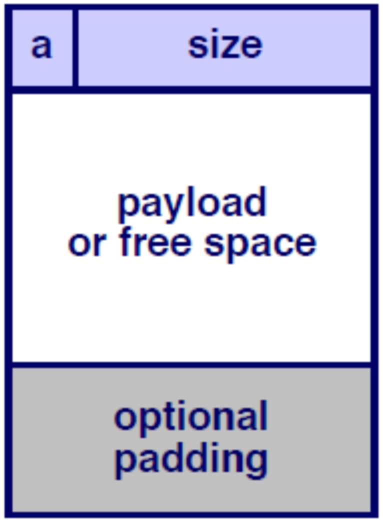


Keeping Track of Free Blocks

- One of the biggest jobs of an allocator is knowing where the free memory is
- The allocator's approach to this problem affects:
  - Throughput – time to complete a malloc() or free()
  - Space utilization – amount of extra metadata used to track location of free memory
- There are many approaches to free space management
  - Next, we will talk about one: **Implicit free lists.**

Implicit Free List

- Idea: Each block contains a header with some extra information.
- Allocated bit indicates whether block is allocated or free.
- Size field indicates entire size of block (including the header)
- Trick: Allocation bit is just the low-order bit of the size word
- For this lecture, let's assume the header size is 1 byte.
- Makes the pictures that I'll show later on easier to understand.
- This means the block size is only 7 bits, so max. block size is 127 bytes (2^7-1).
- Clearly a real implementation would want to use a larger header (e.g., 4 bytes).



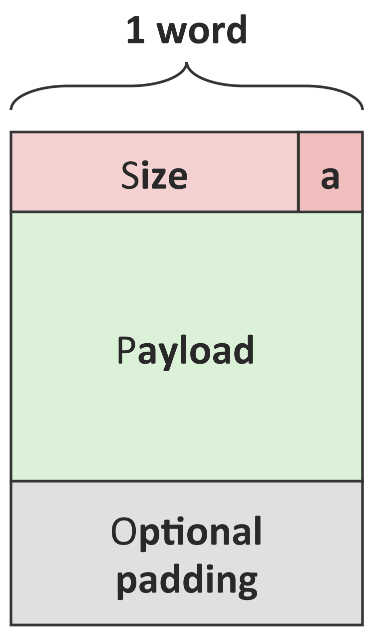
a = 1: block is allocated
a = 0: block is free

size: block size

payload: application data

Implicit free list

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, low-order address bits are always 0
  - Why store an always-0 bit? Use it as allocated/free flag!
  - When reading size word, must mask out this bit



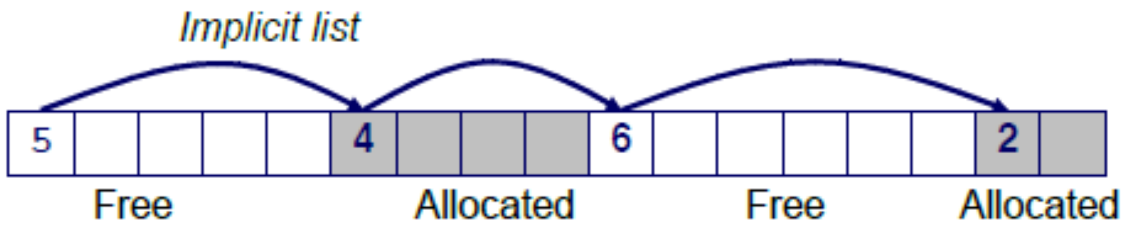
Format of allocated and free blocks

a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

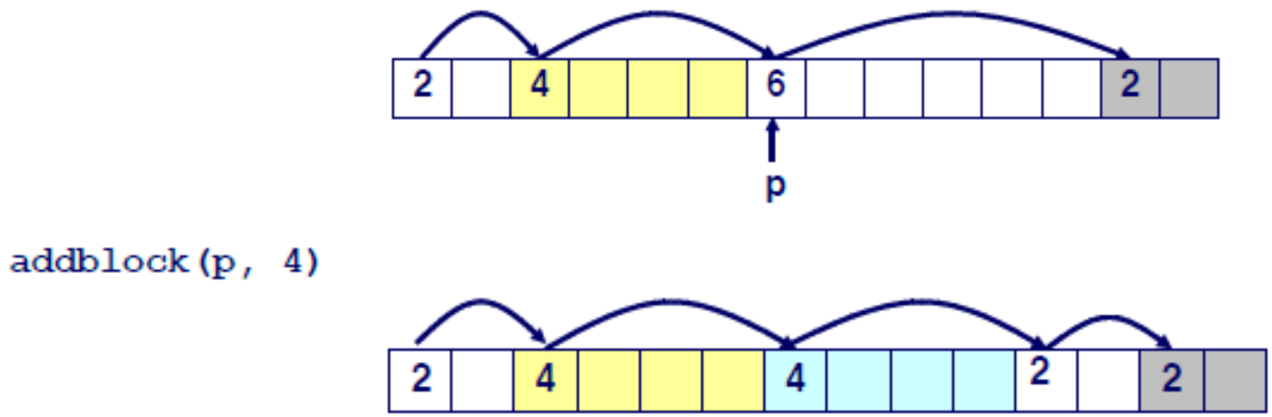
Implicit free list



- No explicit structure tracking location of free/allocated blocks.
  - Rather, the size word (and allocated bit) in each block form an implicit “block list”
- How do we find a free block in the heap?
- Start scanning from the beginning of the heap.
- Traverse each block until (a) we find a free block and (b) the block is large enough to handle the request.
- This is called the first fit strategy.
  - Could also use next fit, best fit, etc

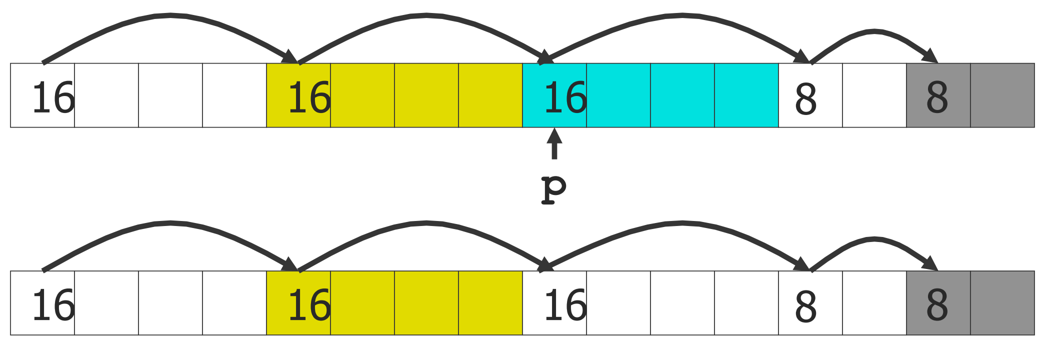
Implicit list: Allocating a Block

- Splitting free blocks
  - Since allocated space might be smaller than free space, we may need to split the free block that we're allocating within



Implicit List: Freeing a Block

- Simplest implementation:
  - Only need to clear allocated flag
  - `void free_block(ptr p) { *p = *p & ~1; }`
- But can lead to "false fragmentation"



`free(p)`

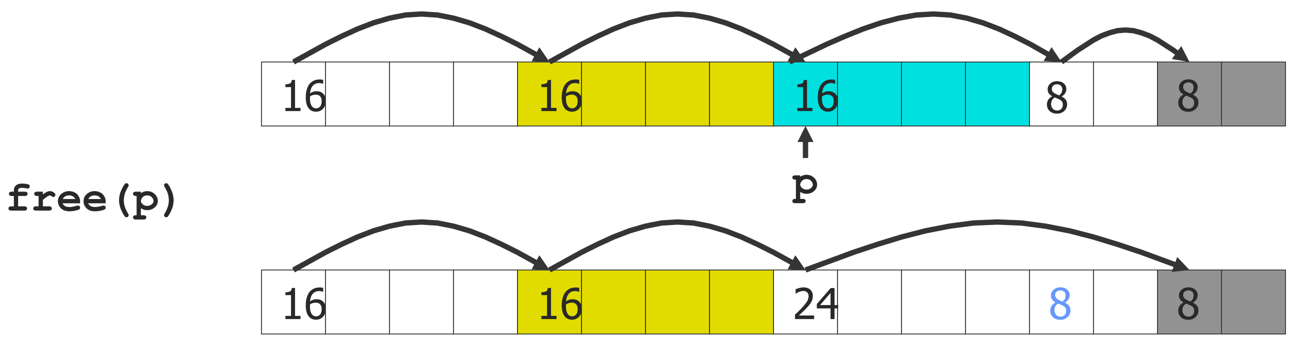
`malloc(20)`

Oops!

- There's enough free space, but allocator won't find it!

Implicit List: Coalescing

- Join (coalesce) with next and previous block if they are free
  - Coalescing with next block



- But how do we coalesce with previous block?

Implicit Lists: Summary

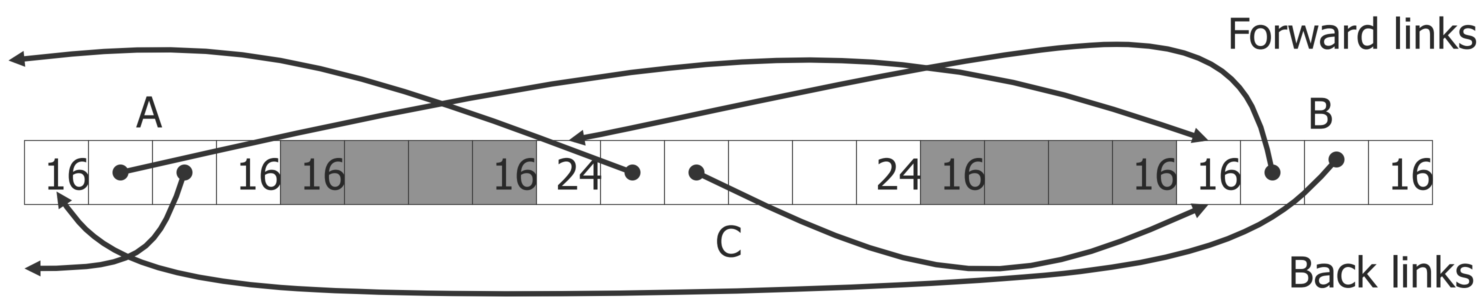
- Implementation: very simple
- Allocate: linear-time worst case
- Free: constant-time worst case—even with coalescing
- Memory usage: will depend on placement policy
  - First, next, or best fit

- Not used in practice for malloc/free because of linear-time allocate, but used in some special-purpose applications

- However, concepts of splitting and boundary tag coalescing are general to *all* allocators

Alternative: Explicit Free Lists

- Use data space for link pointers
  - Typically doubly linked
  - Still need boundary tags for coalescing



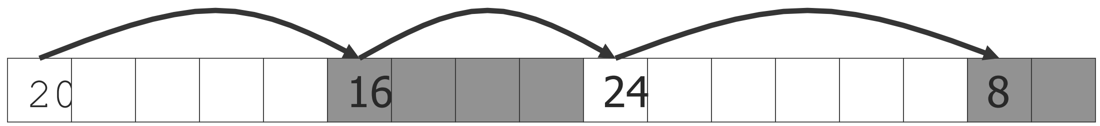
- Links aren't necessarily in same order as blocks! Advantage?

Freeing with Explicit Free Lists

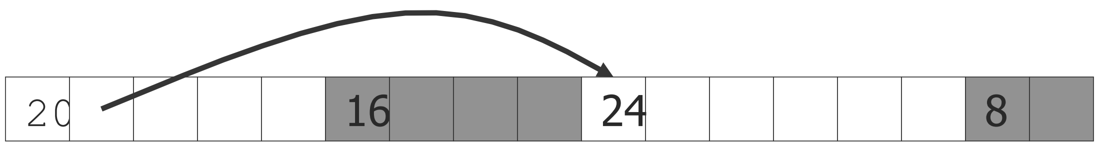
- **Insertion policy**: Where in free list to put newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at beginning of free list
    - Pro: simple, and constant-time
    - Con: studies suggest fragmentation is worse than address-ordered
  - Address-ordered policy
    - Insert freed blocks so list is always in address order
      - i.e. addr(pred) < addr(curr) < addr(succ)
    - Con: requires search (using boundary tags)
    - Pro: studies suggest fragmentation is better than LIFO

Keeping Track of Free Blocks

- *Method 1*: Implicit list using lengths -- links all blocks



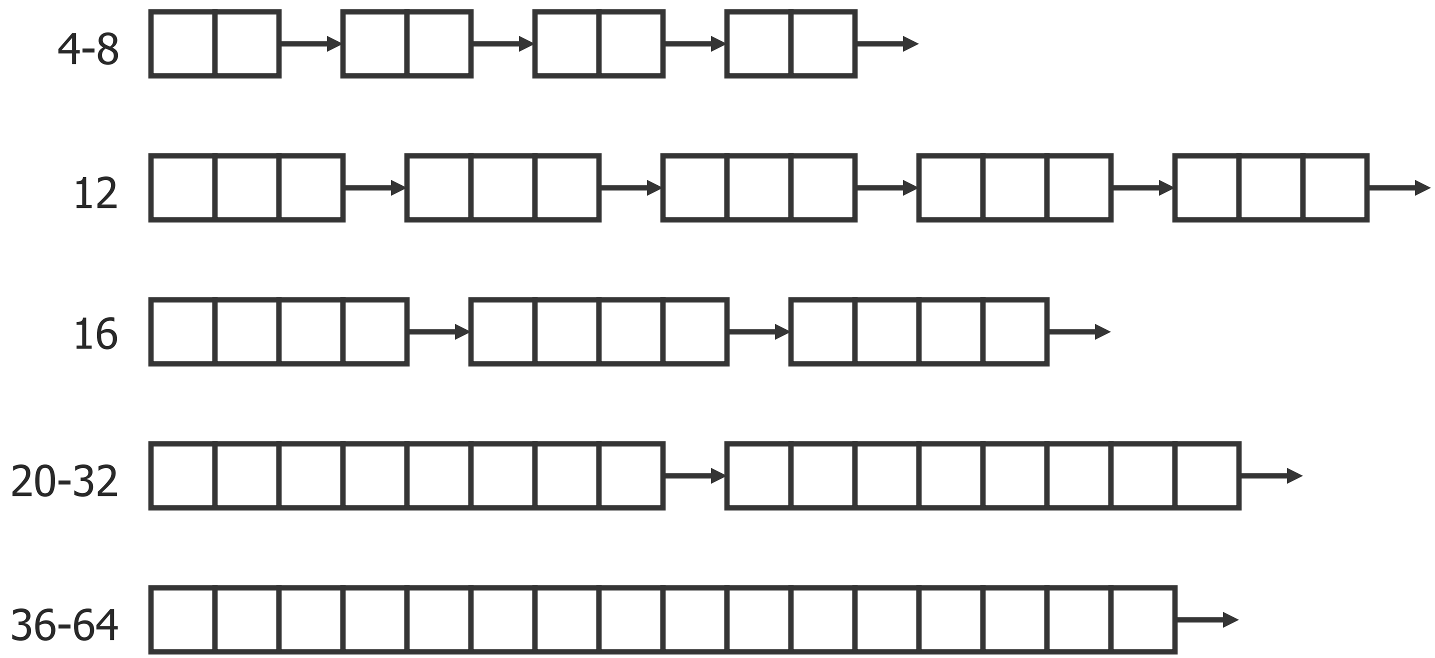
- *Method 2*: Explicit list among the free blocks using pointers within the free blocks



- *Method 3*: Segregated free list
  - Different free lists for different size classes
  - We'll talk about this one next

Segregated Storage

- Each *size class* has its own collection of blocks



  - Often separate size class for every small size (8, 12, 16, …)
  - For larger, typically have size class for each power of 2

Buddy Allocators

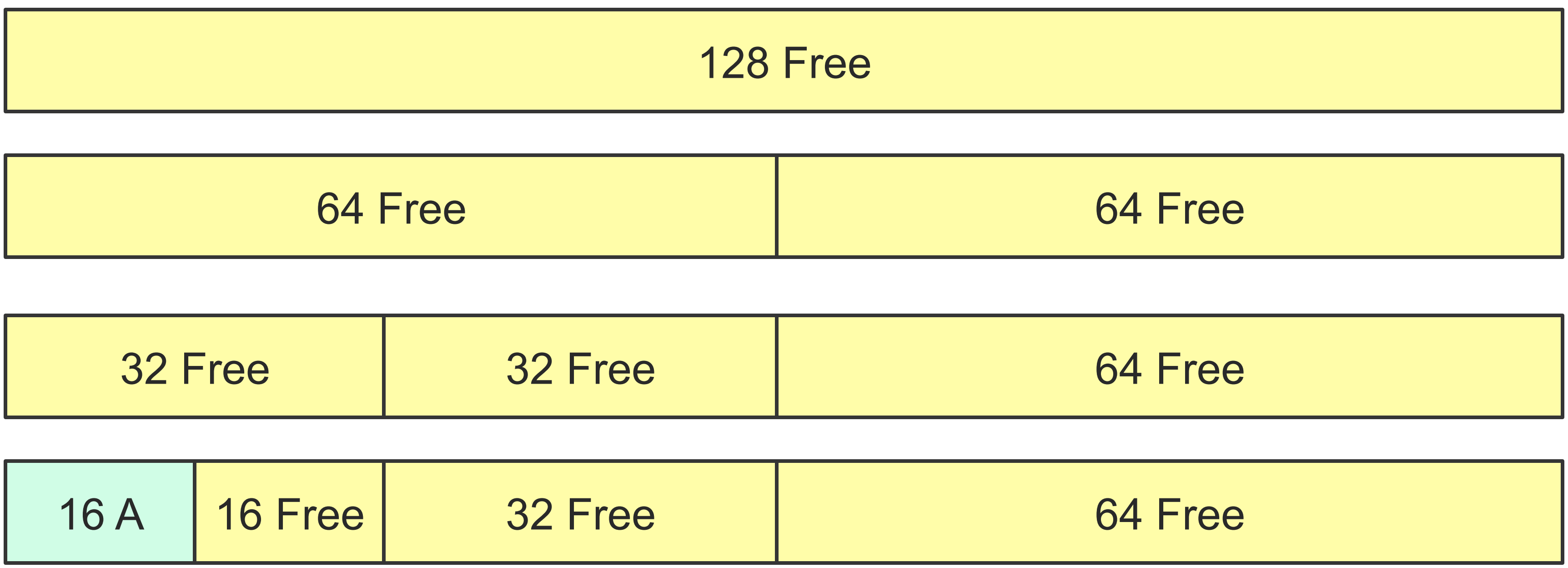
- Special case of segregated fits
- Basic idea:
  - Limited to power-of-two sizes
  - Can only coalesce with "buddy", who is other half of next-higher power of two
- Clever use of low address bits to find buddies
- Problem: large powers of two result in large internal fragmentation (e.g., what if you want to allocate 65537 bytes?)

Buddy System Example

128 Free

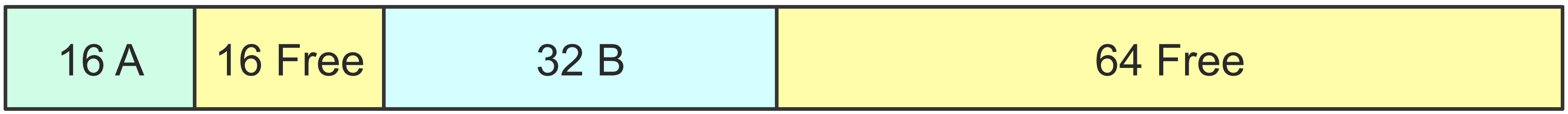
Buddy System Example

Process A requests 16



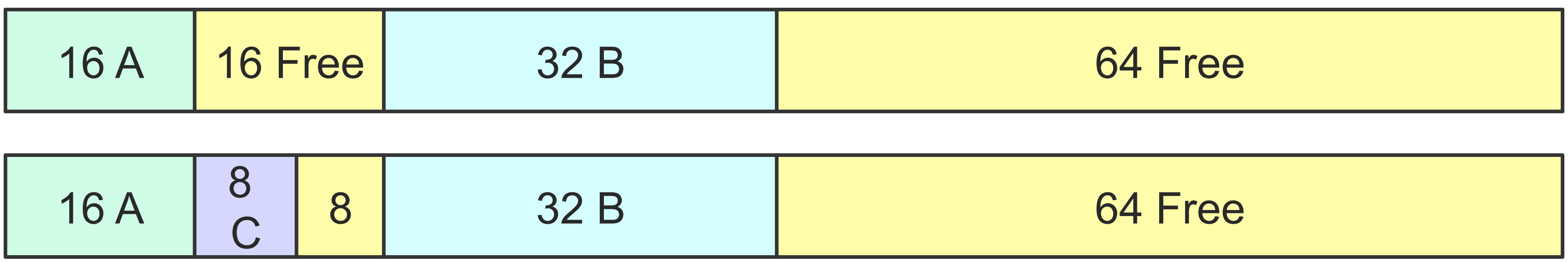
Buddy System Example

Process B requests 32



Buddy System Example

Process C requests 8



Buddy System Example

Process A exits

16 Free	8 C	8	32 B	64 Free

Buddy System Example

Process C exits

16 Free	8	8	32 B	64 Free

16 Free	16 Free	32 B	64 Free

32 Free	32 B	64 Free

- Advantages, disadvantages?
- Advantage: Minimizes external fragmentation
- Disadvantage: Internal fragmentation when not 2^n-sized request

So what should I do for MP2?

- Designs sketched here are reasonable
- Many other possible designs
- Implement anything you want!