Memory Allocation

Memory allocation within 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

Memory allocation within a process

- Solution (stack): stack data structure
  - 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

Allocation example

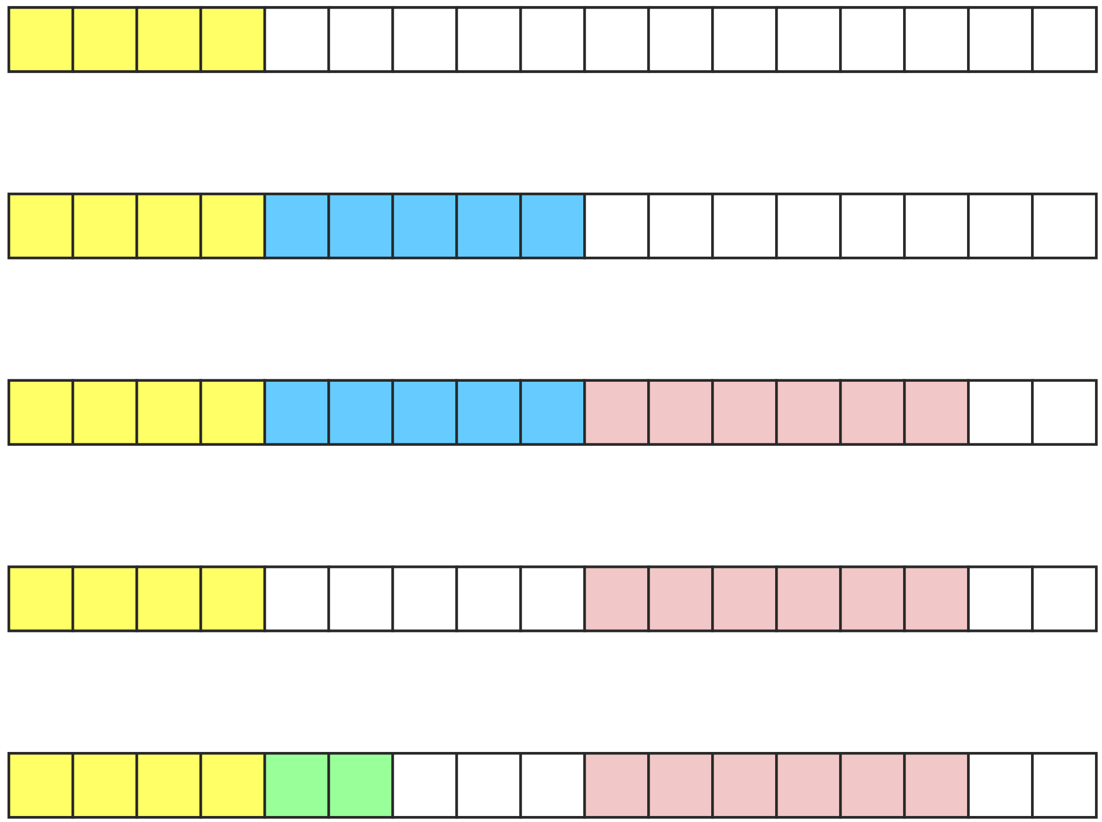
`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

`p4 = malloc(2)`



`malloc` Constraints

- Applications
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - `free` request must be to a `malloc`'d block

`malloc` Constraints

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

Goal 1: Speed

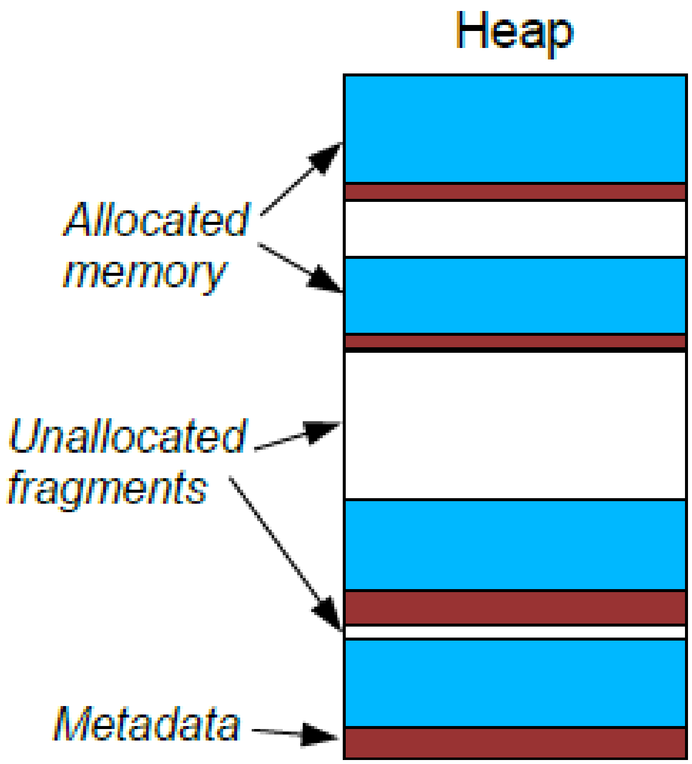
- Allocate fast!
  - Minimize overhead for both allocation and deallocation
- Maximize throughput
  - Number of completed `malloc` or `free` requests per unit time
  - Example
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds

Goal 1: Speed

- BUT
  - A fast allocator may not be efficient in terms of memory utilization
  - Faster allocators tend to be "sloppier"
    - Example: don't look through every free block to find the perfect fit

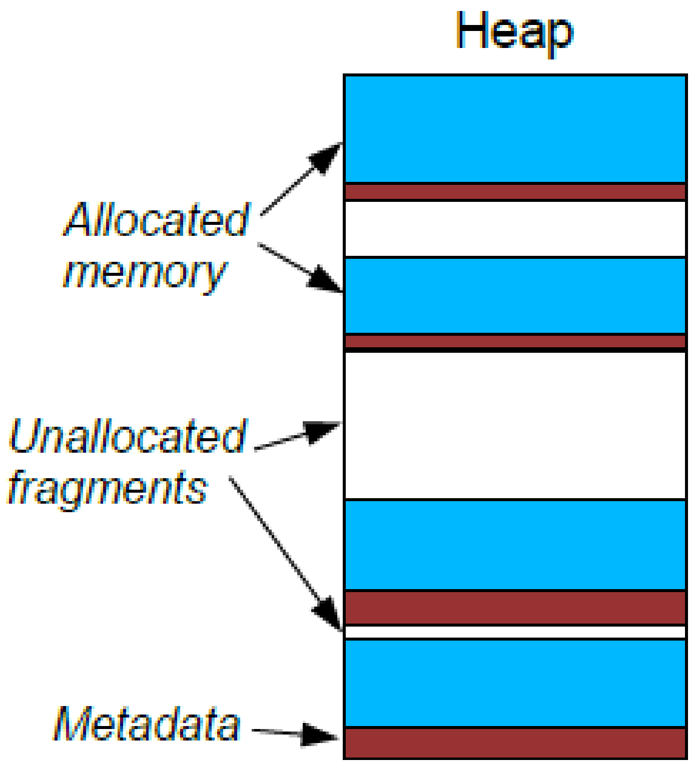
Goal 2: Memory Utilization

- 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)



Goal 2: Memory Utilization

- 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%

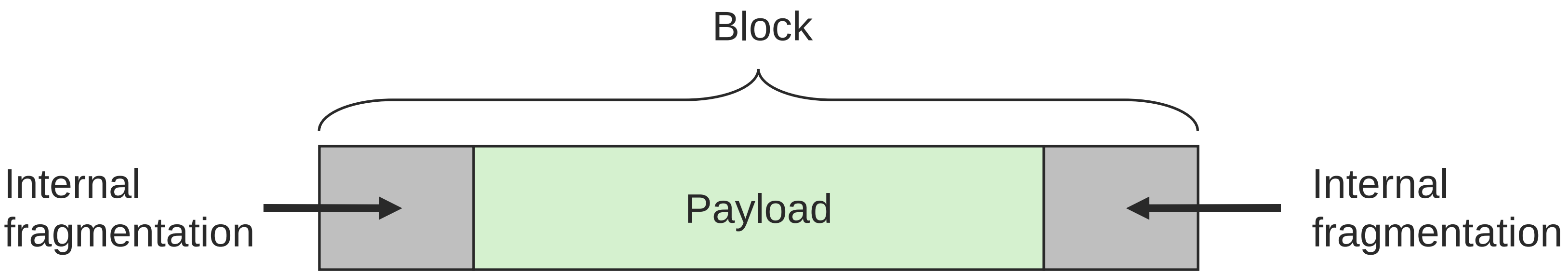


Fragmentation

- Poor memory utilization caused by unallocatable memory
  - internal fragmentation
  - external fragmentation
- OS fragmentation
  - When allocating memory to processes
- `malloc` fragmentation
  - When allocating memory to applications

Internal fragmentation

- 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)

Experiment

- Does `libc`'s `malloc` have internal fragmentation? How much?
- How would you test this?
  - 1. Close Facebook
  - 2. Preheat oven to 375°

Run Example

fragtest

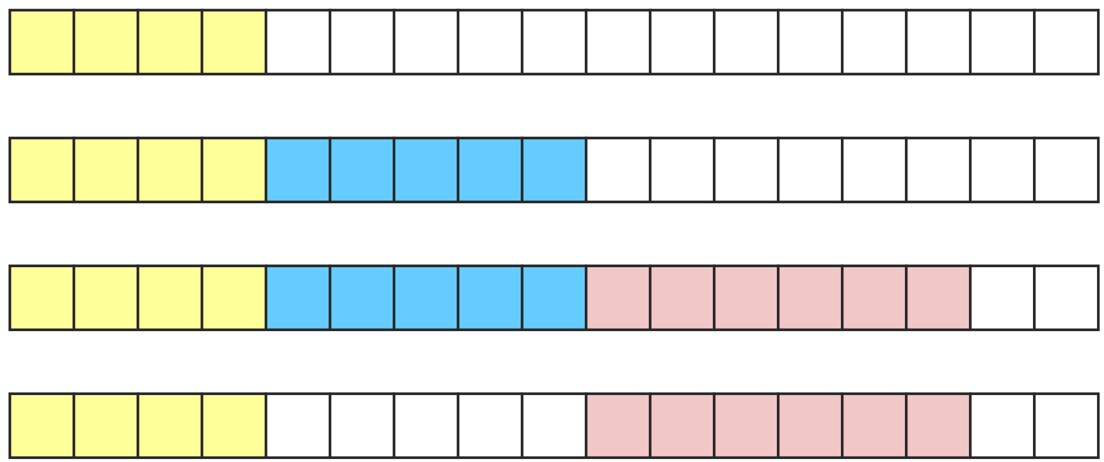
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
   int* a = (int*) malloc(1);
   int* b = (int*) malloc(1);
   int* c = (int*) malloc(100);
   int* d = (int*) malloc(100);

   printf("a = %p\nb = %p\nc = %p\nd = %p\n",
   a,b,c,d);
}
```

What output would you expect?

External Fragmentation

- There is enough aggregate heap memory, but no single free block is large enough

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(6)
```



Oops! (what would happen now?)

Depends on the pattern of future requests
Difficult to plan for

Conflicting performance goals

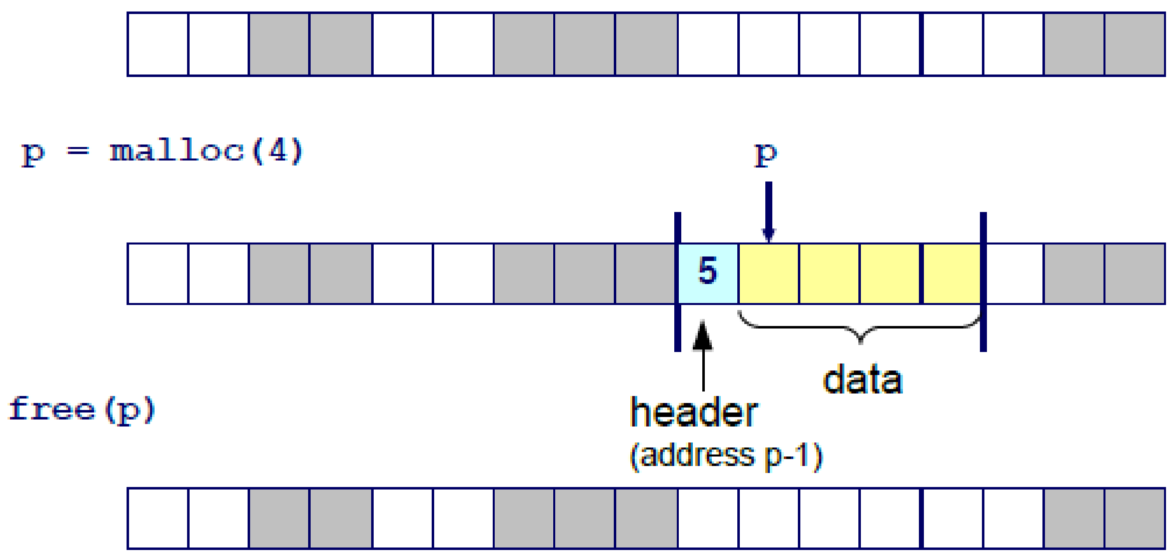
- Throughput vs. Utilization
  - Difficult to achieve simultaneously
- Speed vs. Efficiency
  - Faster allocators tend to be “sloppier” with memory usage
  - Space-efficient allocators may not be very fast
    - Tracking fragments to avoid waste generally results in longer allocation times

Implementation issues you need to solve!

- How do I know how much memory to free just given a pointer?

Keep the length of the block in the header preceding the block

Requires an extra word for every allocated block

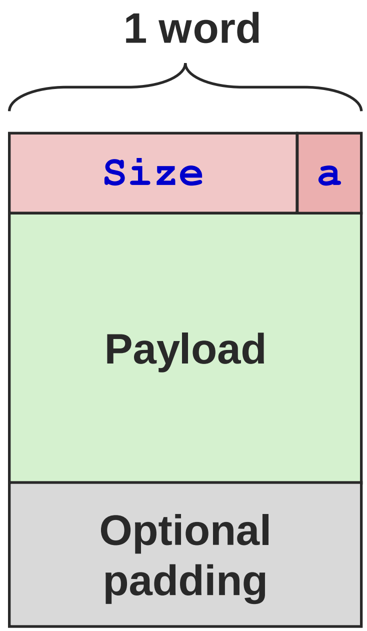


Keeping Track of Free Blocks

- 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

Implicit Free Lists

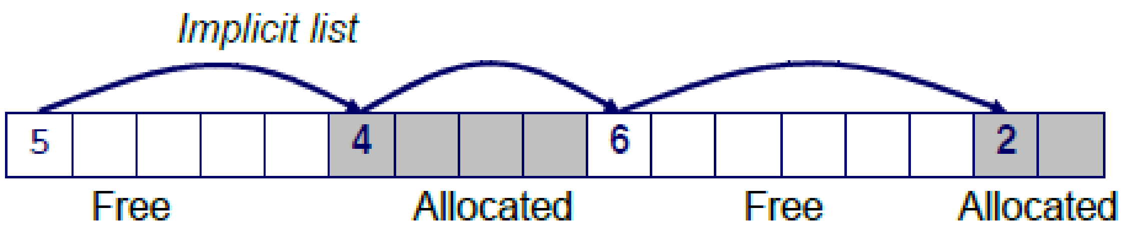
- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, low-order address bits are always 0
  - Why store an always-0 bit? Use it as allocated/free flag!
  - When reading size word, must mask out this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



`a = 1`: Allocated block
`a = 0`: Free block
`Size`: block size

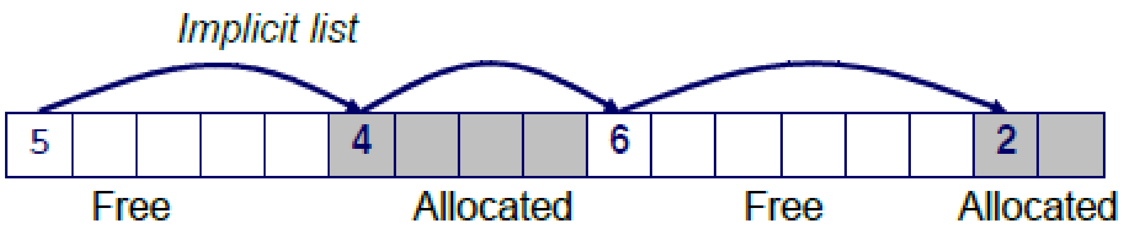
Payload: application data (allocated blocks only)

Implicit Free Lists



- No explicit structure tracking location of free/allocated blocks.
  - Rather, the size word (and allocated bit) in each block form an implicit "block list"

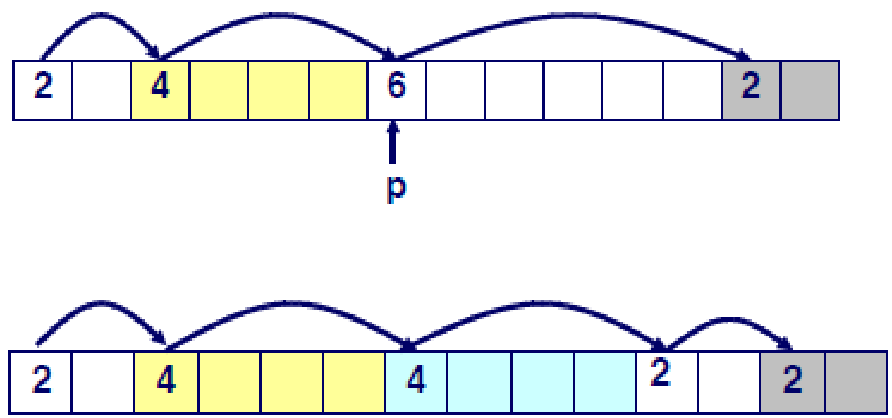
Implicit Free Lists: Free Blocks



- 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 Free Lists: Allocating Blocks

- Splitting free blocks
  - Allocated space might be smaller than free space,
  - May need to split the free block



`addblock(p, 4)`

Implicit Free Lists: Freeing a Block

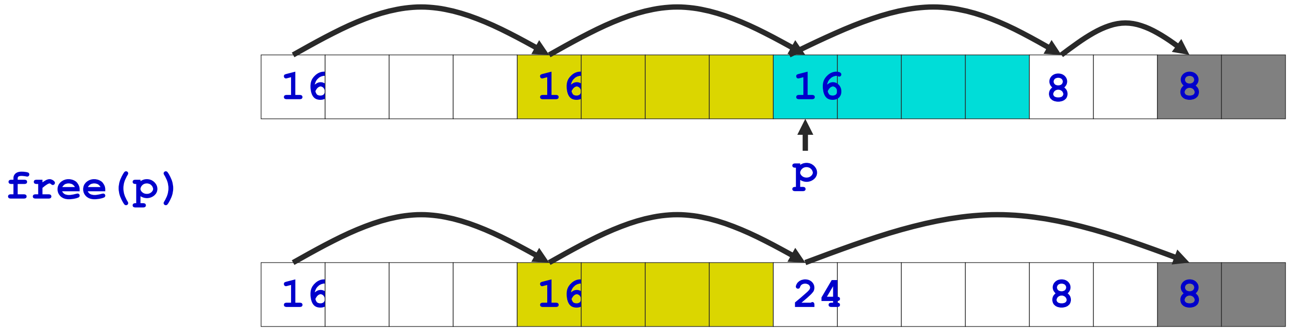
- Simplest implementation:
  - Only need to clear allocated flag

```
void free_block(ptr p) { *p = *p & ~1; }
```

- Problem?

Implicit Free Lists: Coalescing Blocks

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



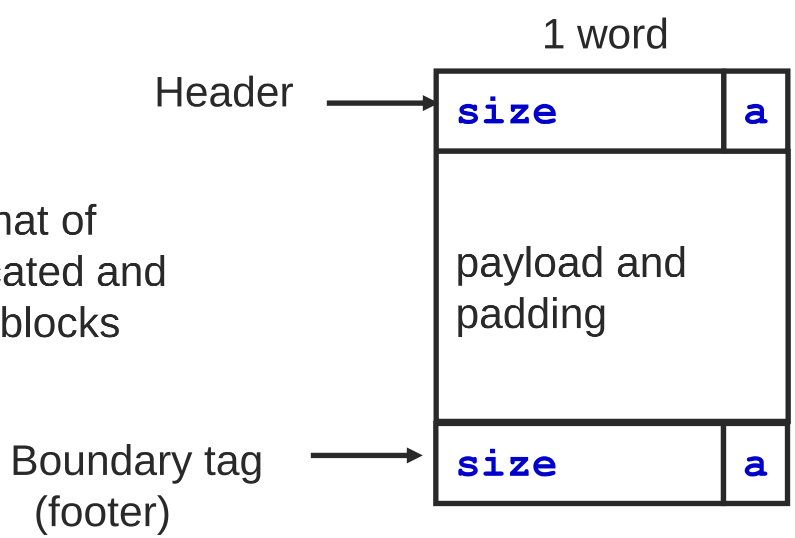
But how do we coalesce with previous block?

Implicit Free Lists: Bidirectional Coalescing

- Boundary tags [Knuth73]
  - Replicate size/allocated word at tail end of all blocks
  - Lets us traverse list backwards, but needs extra space
  - General technique: doubly linked list

Implicit Free Lists: Bidirectional Coalescing

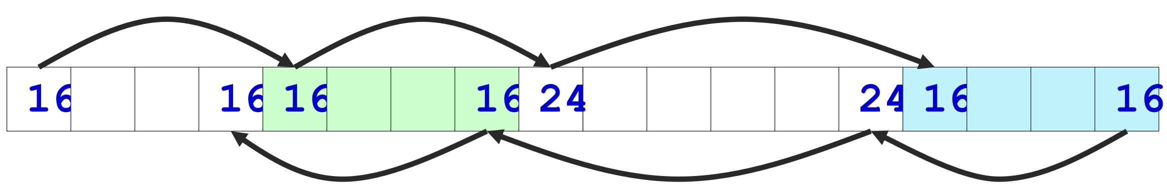
- Boundary tags [Knuth73]



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 Lists: Summary

- Implementation
  - Very simple
- Allocation
  - linear-time worst case
- Free
  - Constant-time worst case—even with coalescing
- Memory usage
  - Will depend on placement policy
  - First, next, or best fit

Implicit Free Lists: Summary

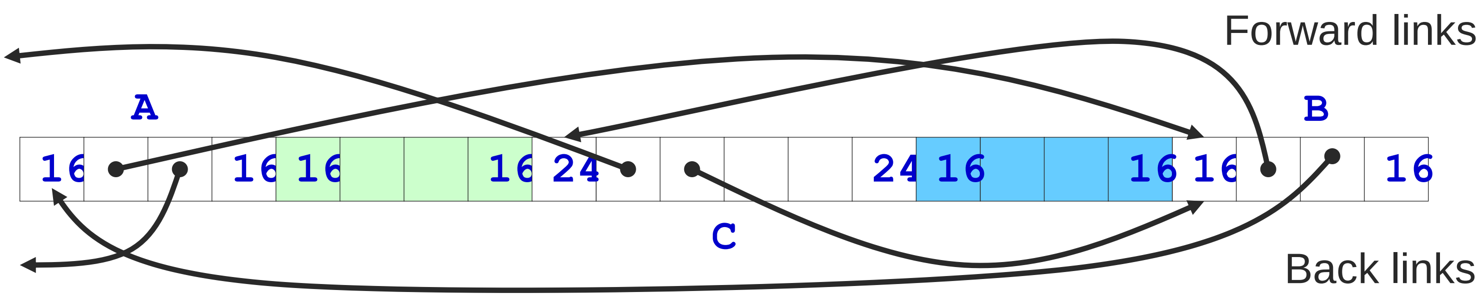
- Not used in practice for **`malloc/free`**
  - linear-time allocation is actually slow!
  - But used in some special-purpose applications
- However, concepts of splitting and boundary tag coalescing are general to all allocators

Alternative Approaches

- Explicit Free Lists
- Segregated Free Lists
  - Buddy allocators

Explicit Free Lists

- Linked list among free blocks
- 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?

Explicit Free Lists: Inserting Free Blocks

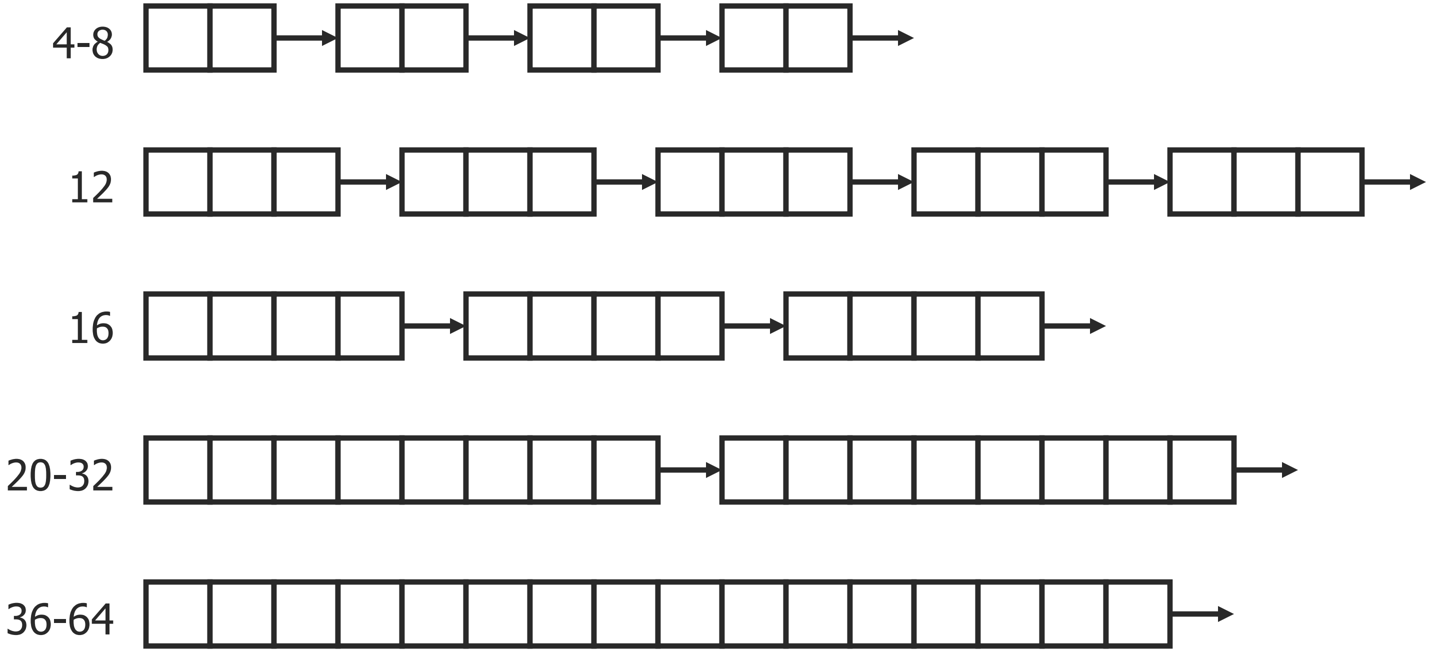
- Where in free list to put newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at beginning of free list
    - Pro
      - Simple, and constant-time
    - Con
      - Studies suggest fragmentation is worse than address-ordered

Explicit Free Lists: Inserting Free Blocks

- Where in free list to put newly freed block?
  - Address-ordered policy
    - Insert so list is always in address order
      - i.e. `addr(pred) < addr(curr) < addr(succ)`
    - Con
      - Requires search (using boundary tags); slow!
    - Pro
      - studies suggest fragmentation is better than LIFO

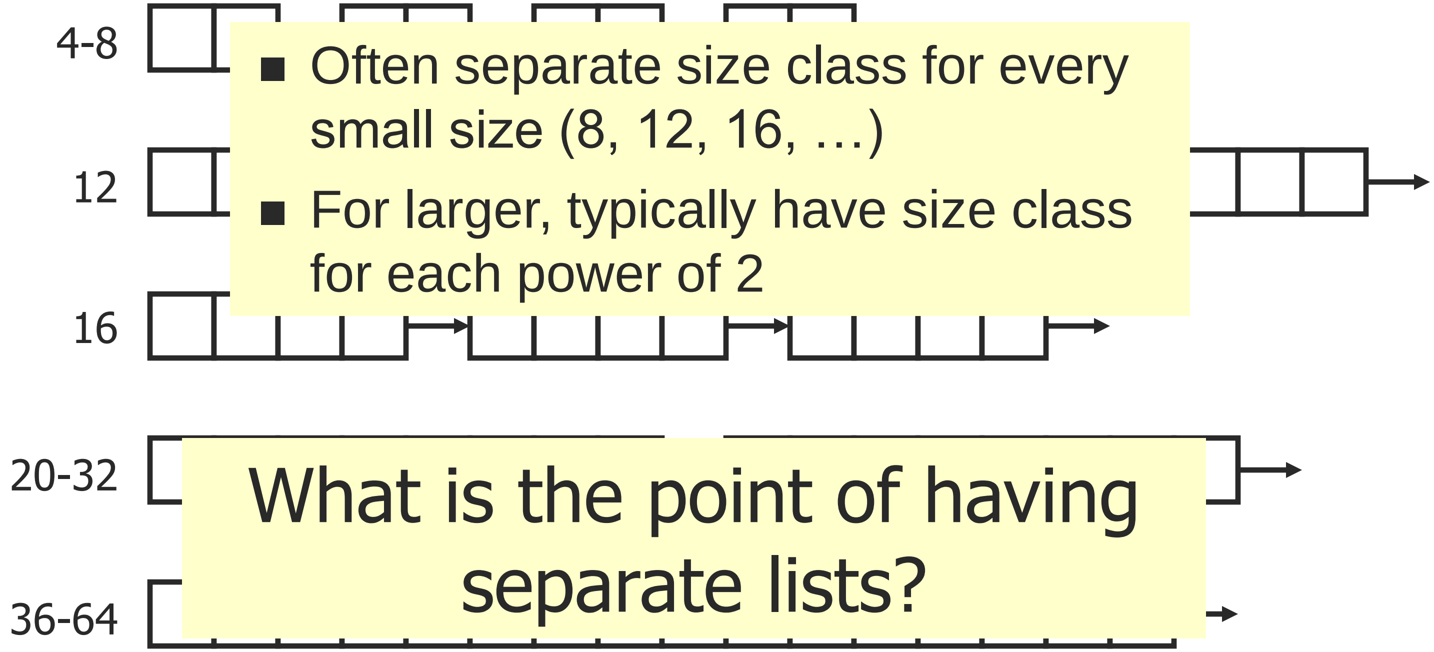
Segregated Free Lists

- Each size class has its own collection of blocks



Segregated Free Lists

- 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

What is the point of having separate lists?

Buddy Allocators

- Special case of segregated free lists
  - Limited allocations to 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

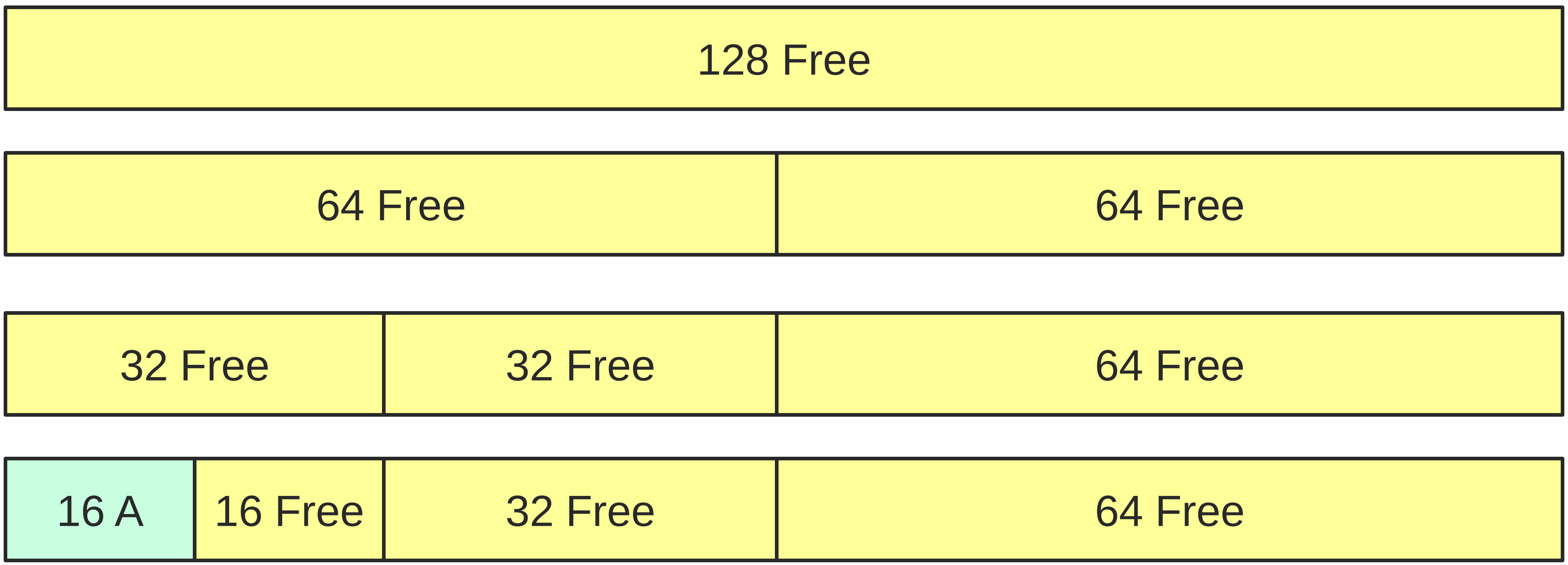
- Approach
  - Minimum allocation size = smallest frame
  - Use a bitmap to monitor frame use
  - Maintain freelist for each possible frame size
    - power of 2 frame sizes from min to max
  - Initially one block = entire buffer
  - If two neighboring frames (“buddies”) are free, combine them and add to next larger freelist

Buddy System Example

128 Free

Buddy System Example

Process A requests 16



Buddy System Example

Process B requests 32

16 A	16 Free	32 B	64 Free

Buddy System Example

Process C requests 8

16 A	16 Free	32 B	64 Free

16 A	8 C	8	32 B	64 Free

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

- Advantage
  - Minimizes external fragmentation
- Disadvantage
  - Internal fragmentation when not 2^n request

So what should I do for MP2?

- Designs sketched here are all reasonable
- But, there are many other possible designs
- So, implement anything you want!

Back to Paging

- On heavily-loaded systems, memory can fill up
- Need to make room for newly-accessed pages
  - Heuristic: try to move “inactive” pages out to disk
    - What constitutes an “inactive” page?
- Paging
  - Refers to moving individual pages out to disk (and back)
  - We often use the terms “paging” and “swapping” interchangeably
  - Different from context switching
    - Background processes often have their pages remain resident in memory

Demand Paging

- Never bring a page into primary memory until its needed
- Fetch Strategies
  - When should a page be brought into primary (main) memory from secondary (disk) storage.
- Placement Strategies
  - When a page is brought into primary storage, where should it be put?
- Replacement Strategies
  - Which page now in primary storage should be removed from primary storage when some other page or segment needs to be brought in and there is not enough room

Page Eviction: When?

- When do we decide to evict a page from memory?
  - Usually, at the same time that we are trying to allocate a new physical page
  - However, the OS keeps a pool of “free pages” around, even when memory is tight, so that allocating a new page can be done quickly
  - The process of evicting pages to disk is then performed in the background

Page Eviction: Which page?

- Hopefully, kick out a less-useful page
  - Dirty pages require writing, clean pages don't
  - Where do you write? To "swap space"
- Goal: kick out the page that's least useful
- Problem: how do you determine utility?
  - Heuristic: temporal locality exists
  - Kick out pages that aren't likely to be used again

Basic Page Replacement

- How do we replace pages?
  - Find the location of the desired page on disk
  - Find a free frame
    - If there is a free frame, use it
    - If there is no free frame, use a page replacement algorithm to select a *victim* frame
  - Read the desired page into the (newly) free frame. Update the page and frame tables.
  - Restart the process

Page Replacement Strategies

- Random page replacement
  - Choose a page randomly
- FIFO - First in First Out
  - Replace the page that has been in primary memory the longest
- LRU - Least Recently Used
  - Replace the page that has not been used for the longest time
- LFU - Least Frequently Used
  - Replace the page that is used least often
- NRU - Not Recently Used
  - An approximation to LRU.
- Working Set
  - Keep in memory those pages that the process is actively using.