Dynamic Memory Allocation

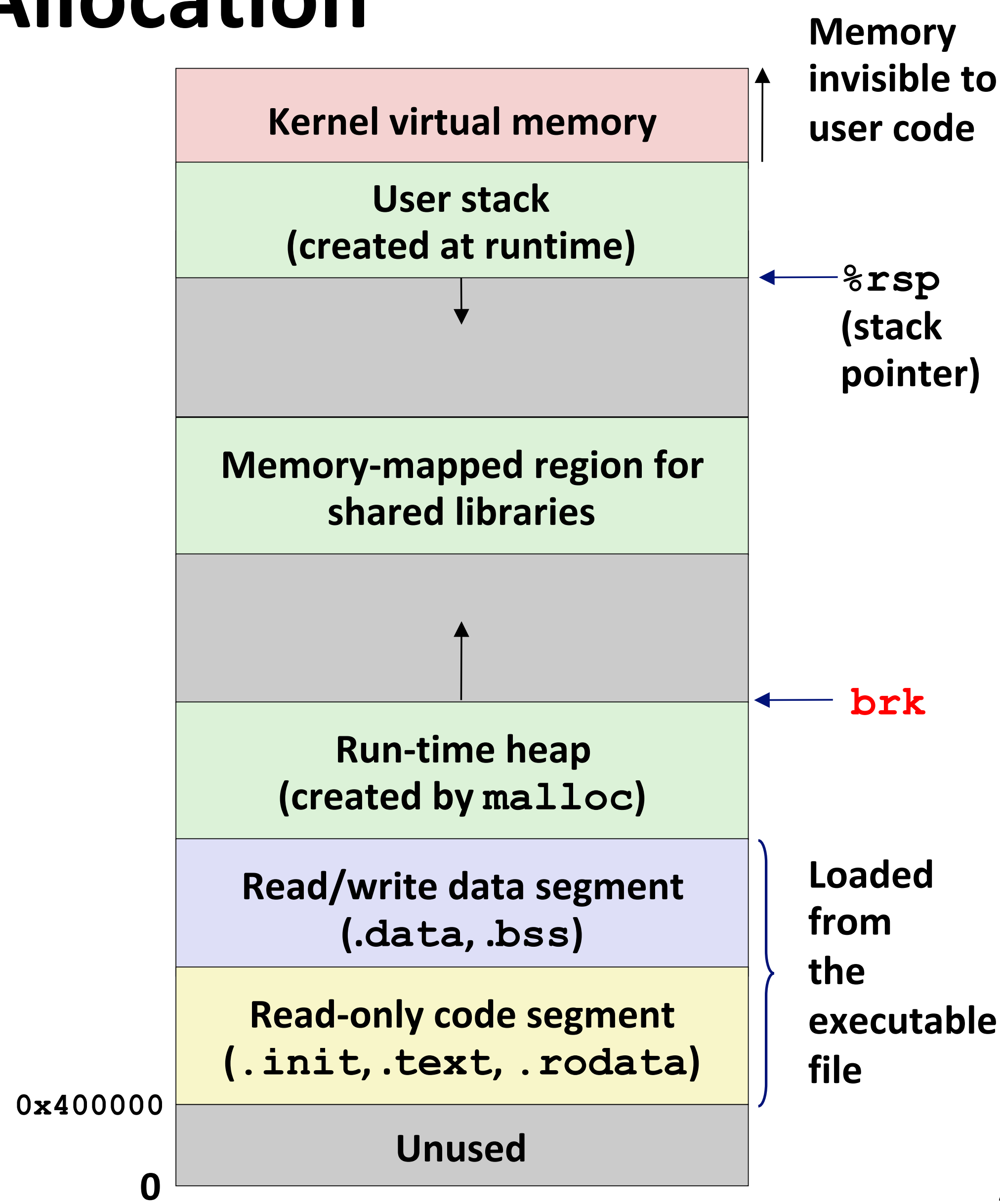
B&O Readings: 9.9 and 9.11

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Dynamic Memory Allocation

- **Programmers use *dynamic memory allocators* (like `malloc`) to acquire memory at run time.**
  - For data structures whose size is only known at runtime
- **Dynamic memory allocators manage an area of process *memory* known as the *heap*.**



Dynamic Memory Allocation

- **Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free***
- **Types of allocators**
  - ***Explicit allocator*:** application allocates and frees
    - E.g., `malloc` and `free` in C
  - ***Implicit allocator:*** application allocates, but does not free
    - E.g. garbage collection in Java, ML, and Lisp

- **Will discuss simple explicit memory allocation today**

The `malloc` Package

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

```
void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes (typically) aligned to 8-byte boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
void free(void *p)
```

- Returns the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`** (or **`realloc`** or **`calloc`**)

Other functions

- **`calloc`**: initializes allocated block to zero
- **`realloc`**: changes size of a previously allocated block
- **`sbrk`**: used internally by allocators to grow or shrink heap

`malloc` Example

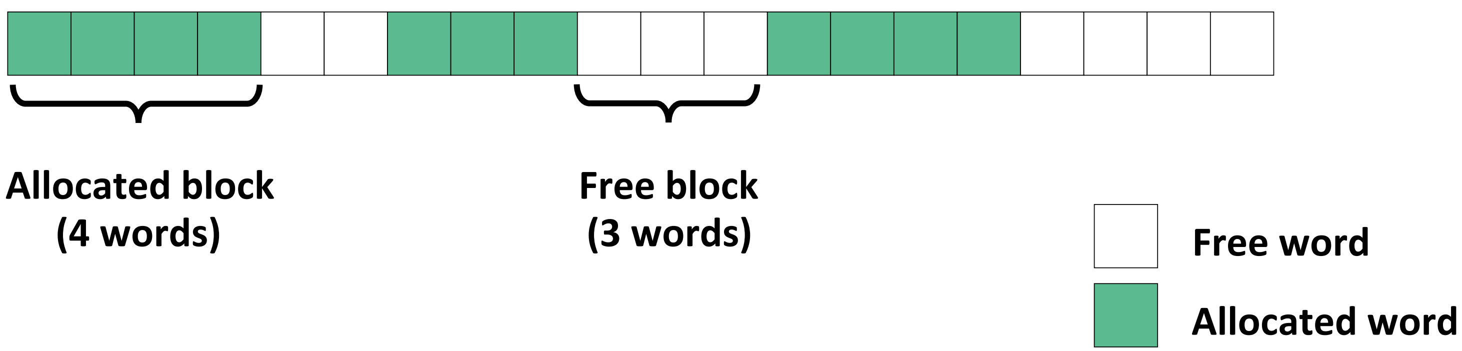
```
void foo(int n, int m) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;

    do_stuff(p);

    /* Return p to the heap */
    free(p);
}
```

Why Dynamic Memory Allocation?

- Don't always know the size needed until runtime
- Allows your data structure to grow and shrink dynamically
- Need the objects to be alive across function invocations

Assumptions Made in This Lecture

- Allocated block is always multiple of 4 bytes (word size)
- Each allocation needs to be aligned by double-word boundary.



- In practice, on x86-64, libc return block that is 16-byte aligned. (For 32-bit it's 8-byte aligned.)

Allocation Example

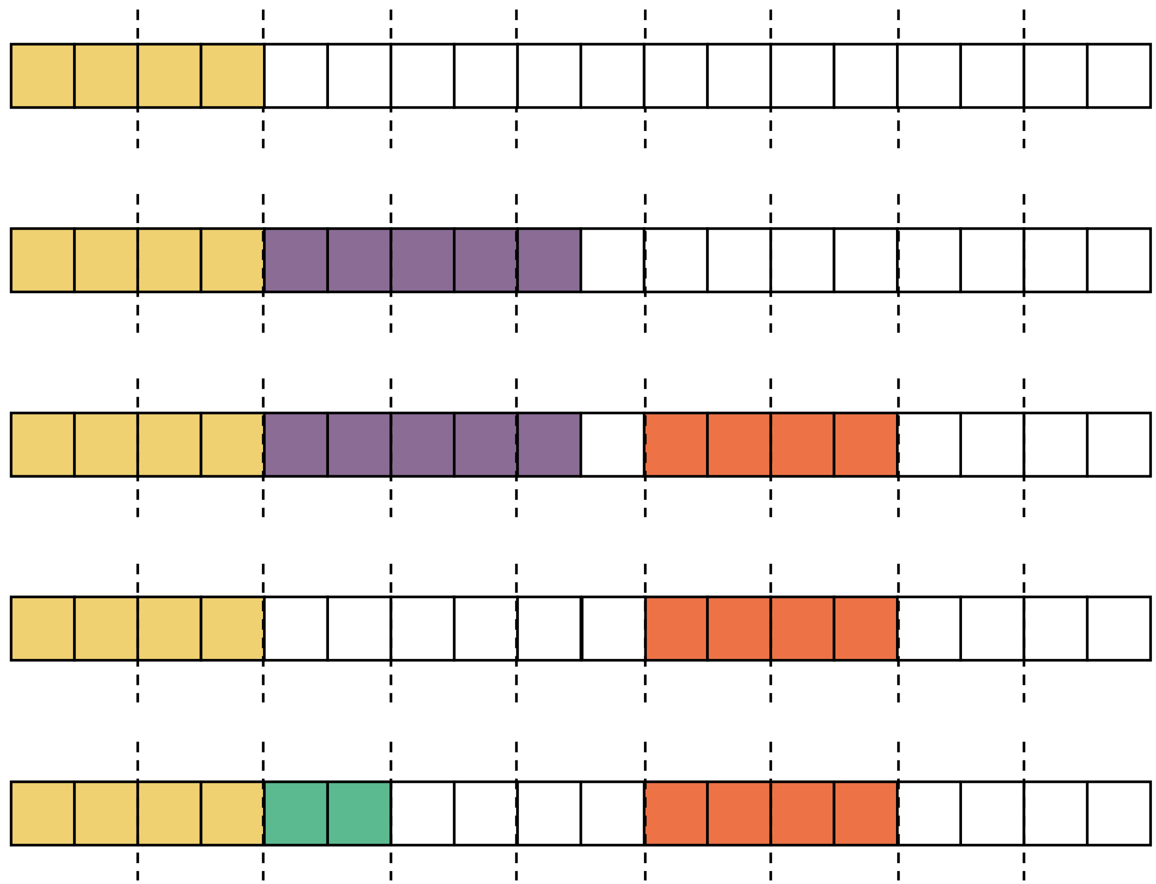
`p1 = malloc(16)`

`p2 = malloc(20)`

`p3 = malloc(16)`

`free(p2)`

`p4 = malloc(8)`



Note: this is not an accurate depiction.

Constraints

- **Applications**
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`** request must be to a **`malloc`**’d block (or **`realloc`** or **`calloc`**)
- **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
    - *i.e.*, can only place allocated blocks in free memory
  - 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?
  - Must align blocks so they satisfy all alignment requirements
    - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes

Implementation Issues: the 5 Questions

1. Given just a pointer, how much memory do we free?

2. How do we keep track of the free blocks?

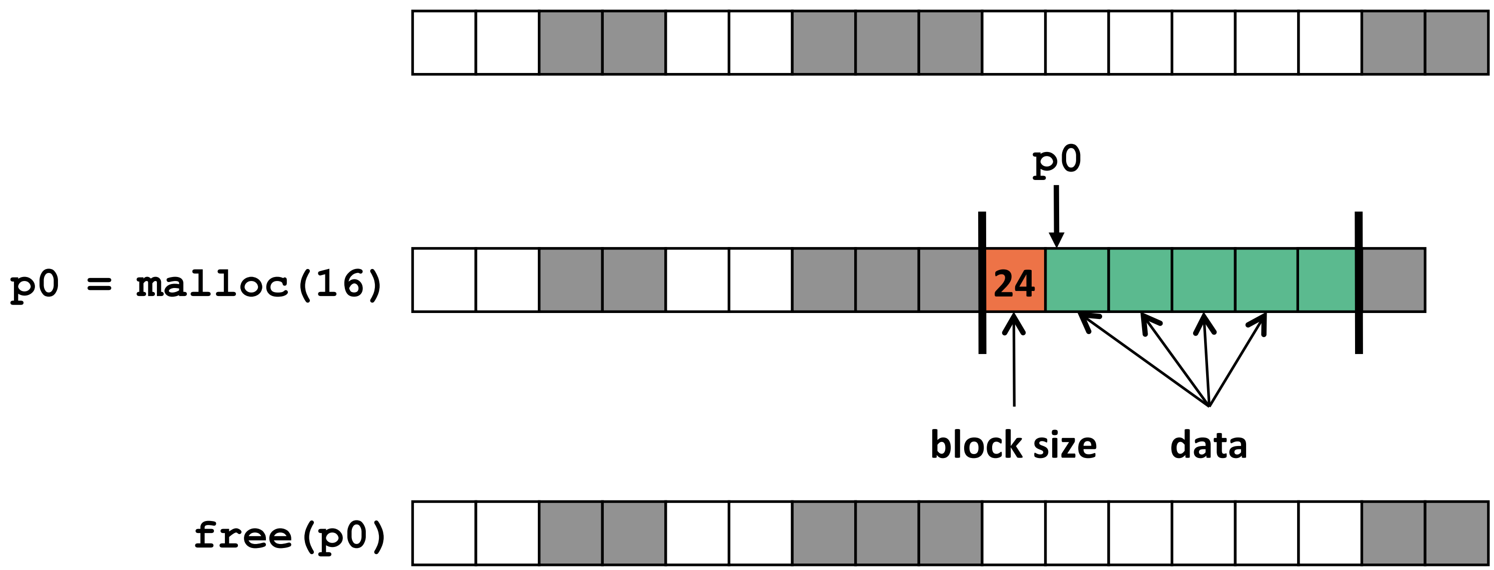
3. How do we pick a block to use for allocation? (if a few work)

4. When allocating a structure that is smaller than the free block it is placed in, what do we do with the extra space?

5. How do we reinsert freed block?

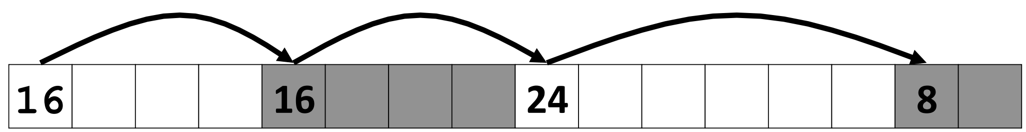
Q1: Knowing How Much to Free

- **Standard method**
  - Keep the length of a block in the word preceding the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block

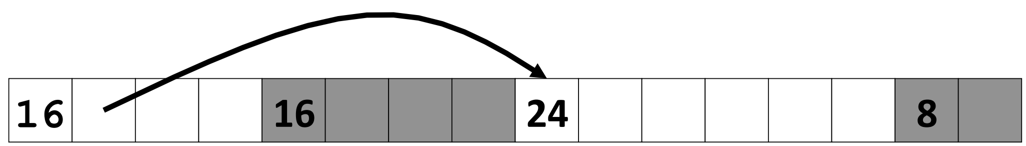


Q2: Keeping Track of Free Blocks

- **Method 1: *Implicit list* using length—links all blocks**



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



- **Method 3: *Segregated free list***
  - Different free lists for different size classes

- **Method 4: *Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Performance Metric: Throughput

- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- **Throughput:**
  - Number of completed requests per unit time
  - Example:
    - 5,000 **`malloc`** calls and 5,000 **`free`** calls in 10 seconds
    - Throughput is 1,000 operations/second

Performance Metric: Memory Utilization

- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

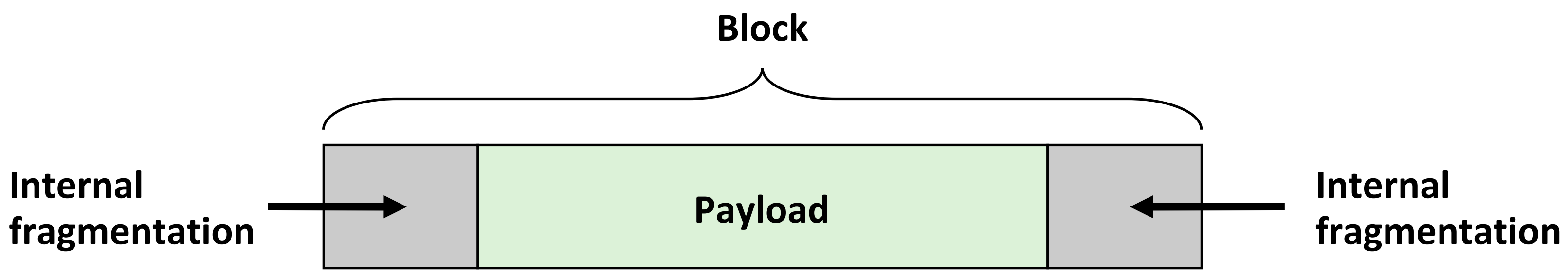
- **Goal: Want to maximize memory utilization (the ration between memory used for actual data versus overall allocation)**

Main Issue: Fragmentation

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

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)
- **Depends only on the pattern of *previous* requests**
  - Thus, easy to measure and optimize for

External Fragmentation

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

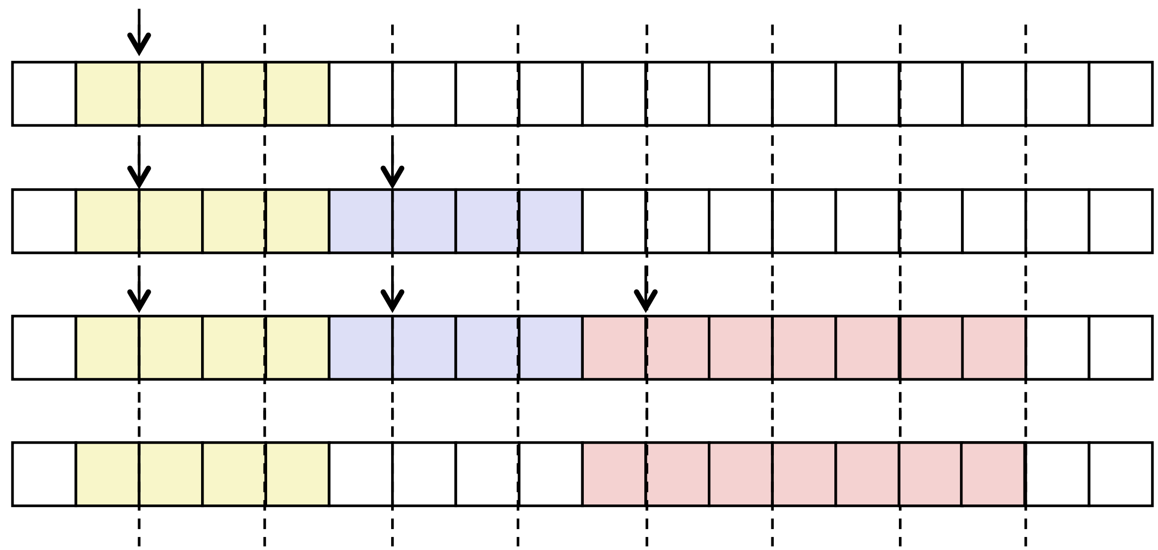
`p1 = malloc(12)`

`p2 = malloc(12)`

`p3 = malloc(16)`

`free(p2)`

`p4 = malloc(20)`



Oops! (what would happen now?)

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

Performance Metric: Memory Utilization

- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- ***Def: Aggregate payload P_k***
  - `malloc(p)` results in a block with a ***payload*** of `p` bytes
  - After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads
- ***Def: Current heap size H_k***
  - Assume H_k is monotonically nondecreasing
    - i.e., heap only grows when allocator uses `sbrk`
- ***Def: Peak memory utilization after k+1 requests***
  - $U_k = (\max_{i<=k} P_i) \;/\; H_k$
  - *Peak memory utilization: when* aggregate payload was closest to size of the heap
- ***Goal: Maximize peak memory utilization***

Summary: Performance Goals

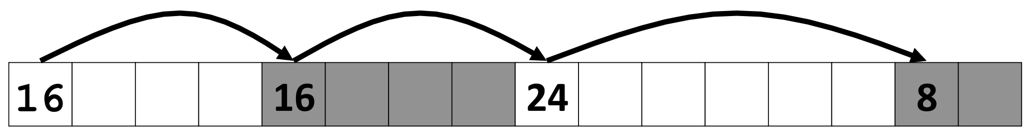
- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- **Maximize Throughput**
- **Maximize Peak Memory Utilization:**
  - When was aggregate payload closest to size of the heap?
  - Poor memory utilization caused by *fragmentation*

Maximizing throughput and peak memory utilization = HARD

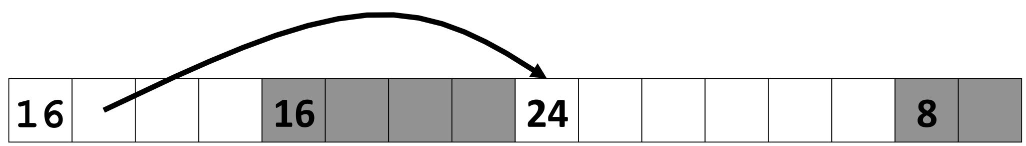
- These goals are often conflicting

Q2: Keeping Track of Free Blocks

- **Method 1:** ***Implicit list*** **using length—links all blocks**



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



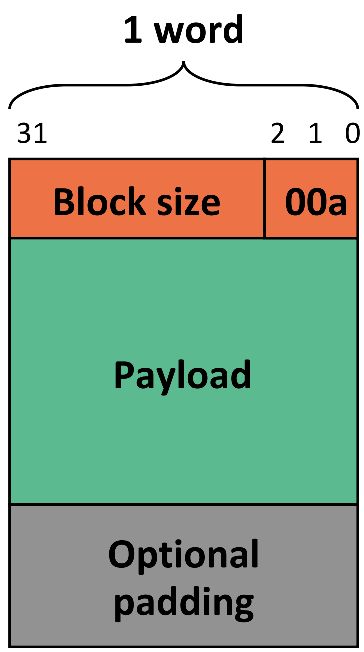
- **Method 3:** ***Segregated free list***
  - Different free lists for different size classes

- **Method 4:** ***Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Method 1: Implicit 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, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks

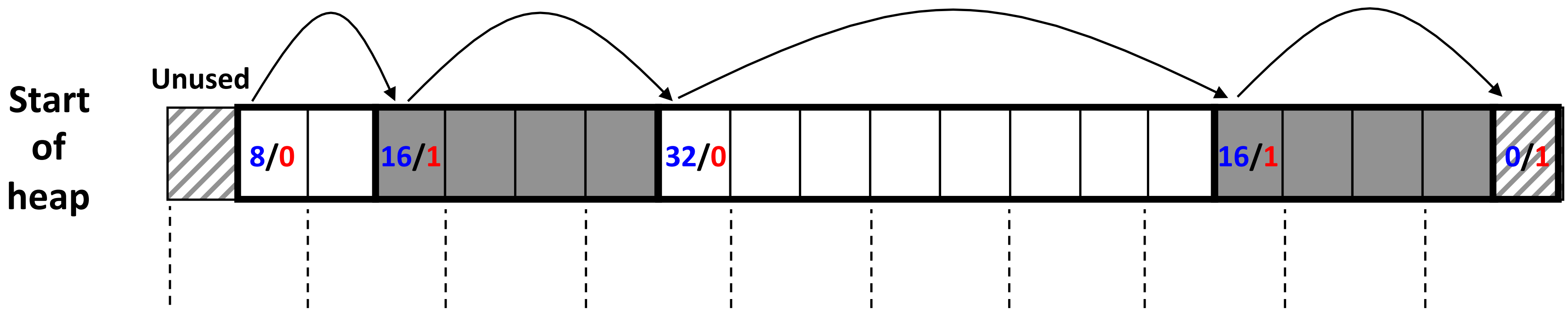


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



Double-word aligned

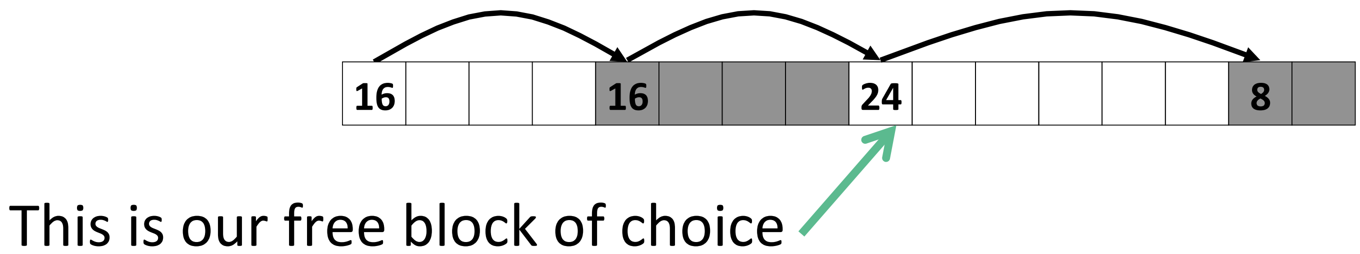
Allocated blocks: shaded grey
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

Q3: Implicit List: Finding a Free Block

- **First fit:**
  - Search list from beginning, choose ***first*** free block that fits:
  - Linear time in total number of blocks (allocated and free)
  - Can cause "splinters" (of small free blocks) at beginning of list
- **Next fit:**
  - Like first fit, but search list starting where previous search finished
  - Often faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- **Best fit:**
  - Search list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Typically runs slower than first fit

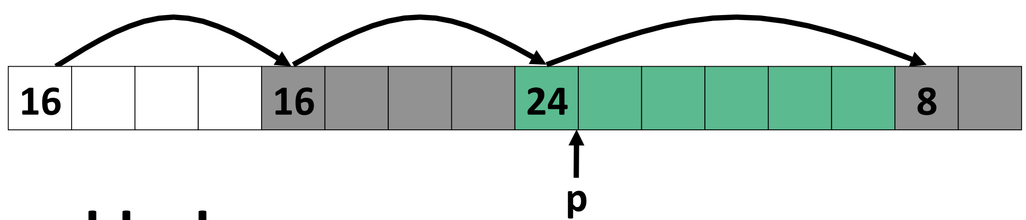
Q4: Implicit List: Allocating in Free Block

Suppose we need to allocate 3 words

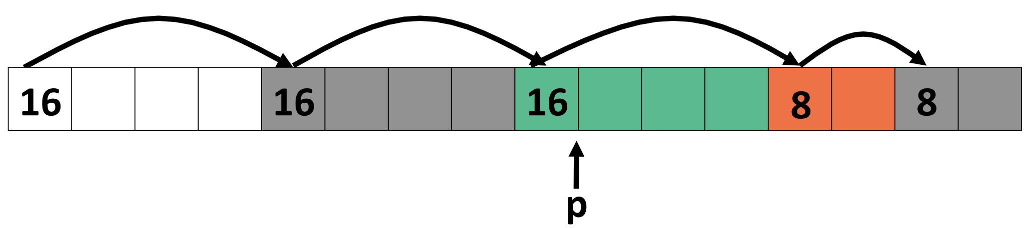


Two options:

1. **Allocate the whole block (wasted space!)**

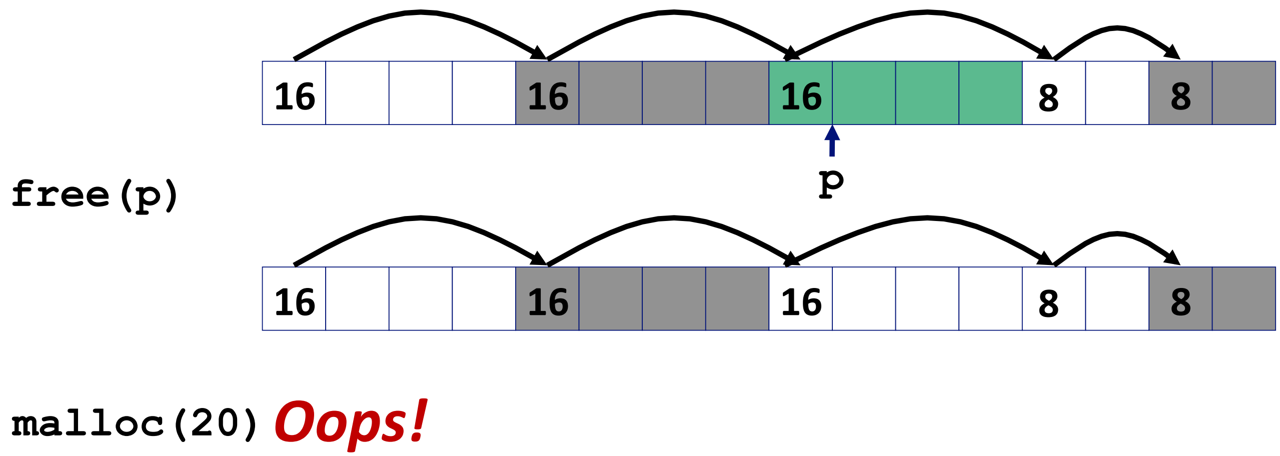


2. **Split the free block**



Q5: Implicit List: Freeing a Block

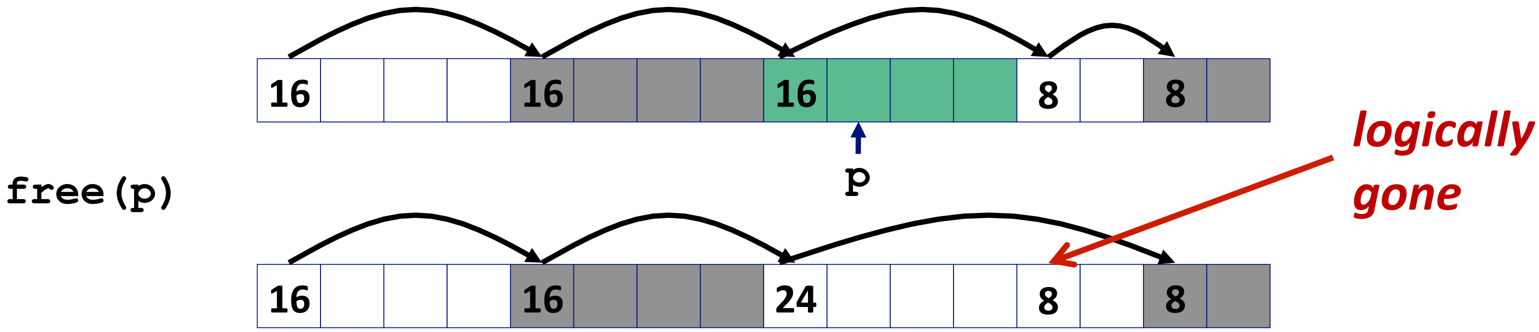
- **Simplest implementation: clear the "allocated" flag**
  - But can lead to "false fragmentation"



There is enough free space, but the allocator won't be able to find it

Implicit List: Coalescing

- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block

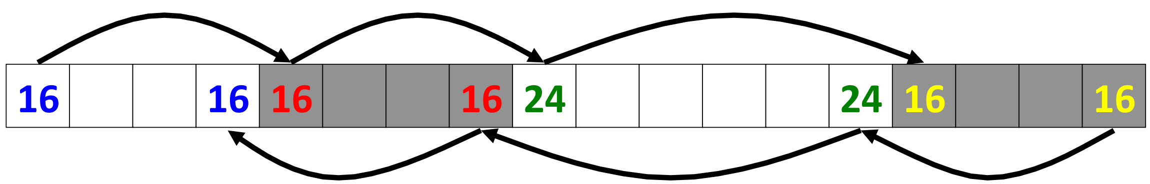


How do we coalesce with *previous* block?

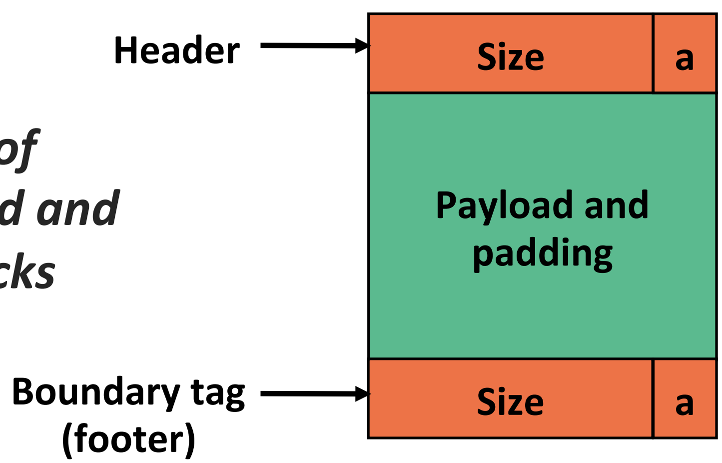
We don't know its size nor whether it's free or not unless we traverse the free list.

Implicit List: Bidirectional Coalescing

- ***Boundary tags*** [Knuth73]
  - Replicate size/allocated word at “bottom” (end) of blocks
  - Allows us to traverse the “list” backwards, but requires extra space
  - Important and general technique!



Format of allocated and free blocks

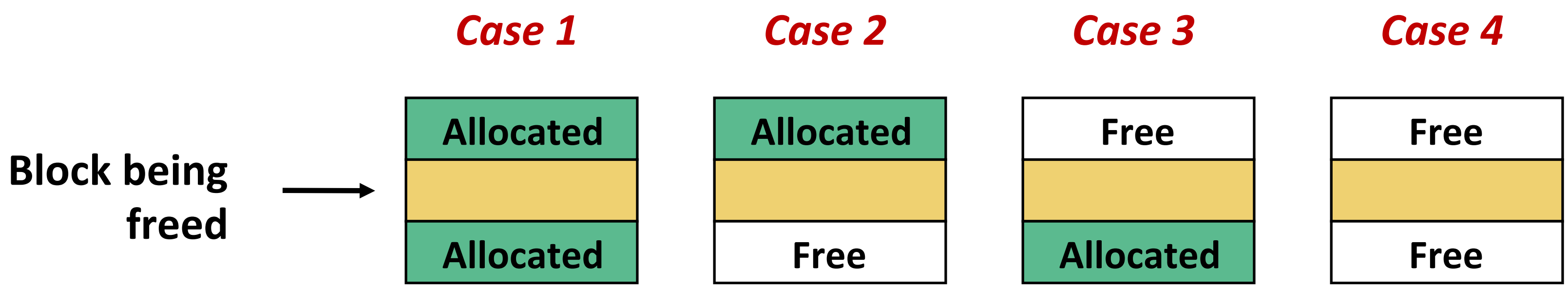


a = 1: Allocated block
a = 0: Free block

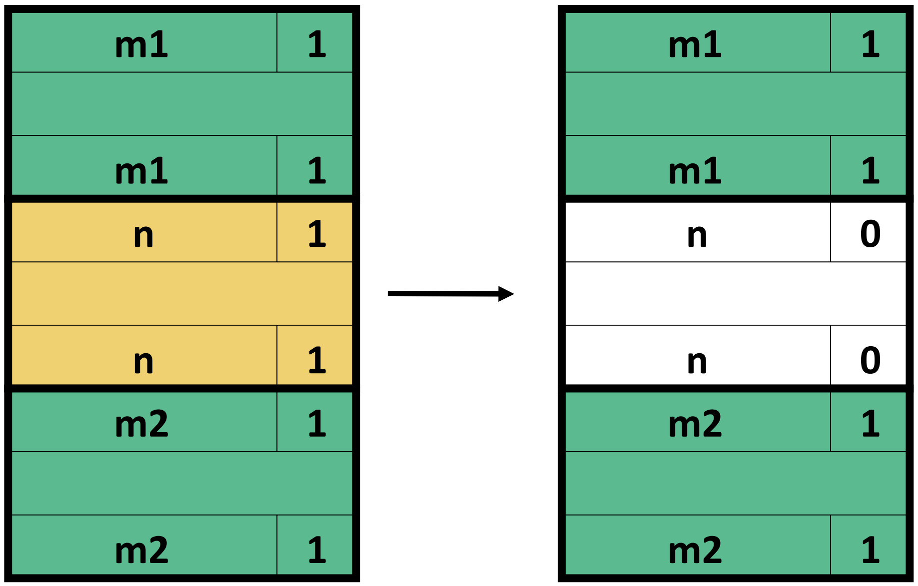
Size: Total block size

Payload: Application data (allocated blocks only)

Constant Time Coalescing

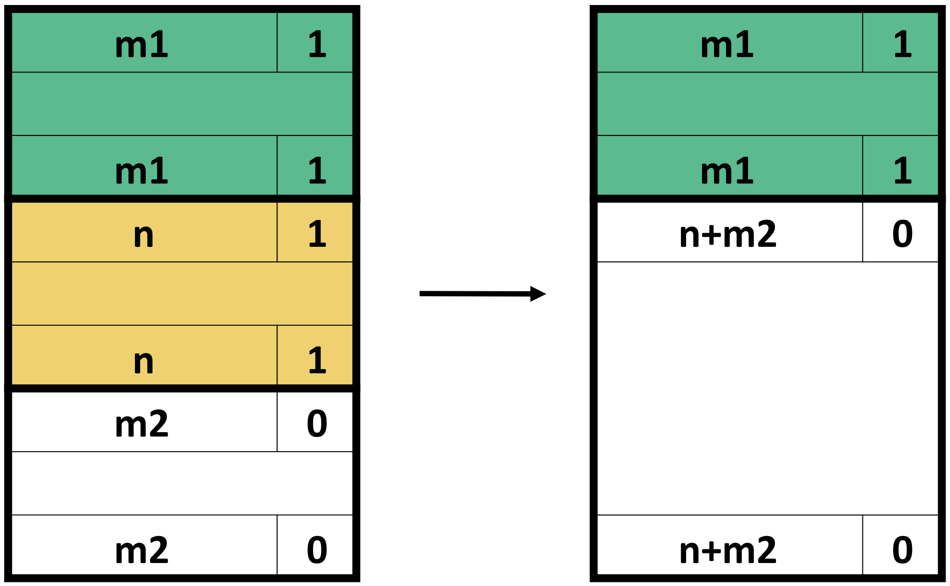


Constant Time Coalescing (Case 1)



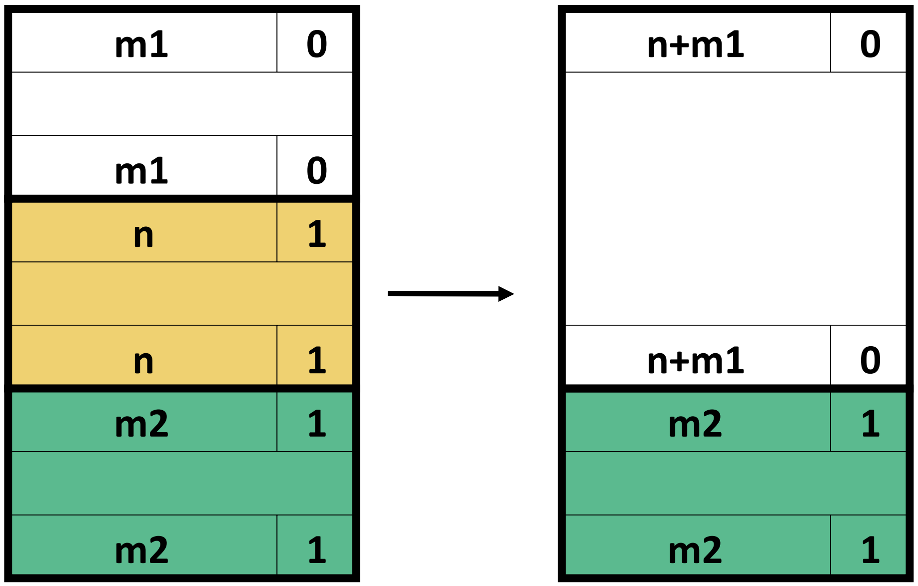
LSB with value '1' indicates that the block is in use.

Constant Time Coalescing (Case 2)



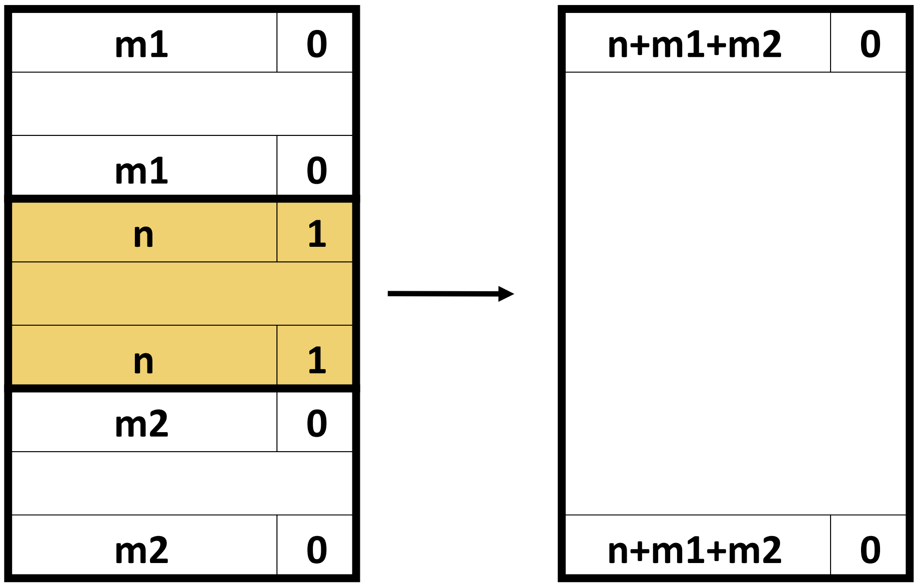
LSB with value '1' indicates that the block is in use.

Constant Time Coalescing (Case 3)



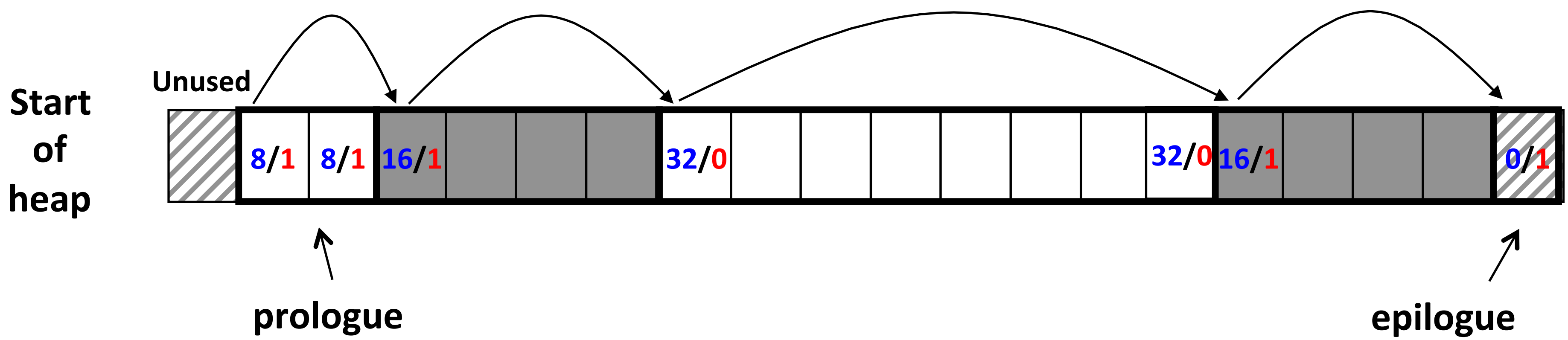
LSB with value '1' indicates that the block is in use.

Constant Time Coalescing (Case 4)



LSB with value '1' indicates that the block is in use.

Prologue and Epilogue: Treat The Beginning and End of Heap Similarly

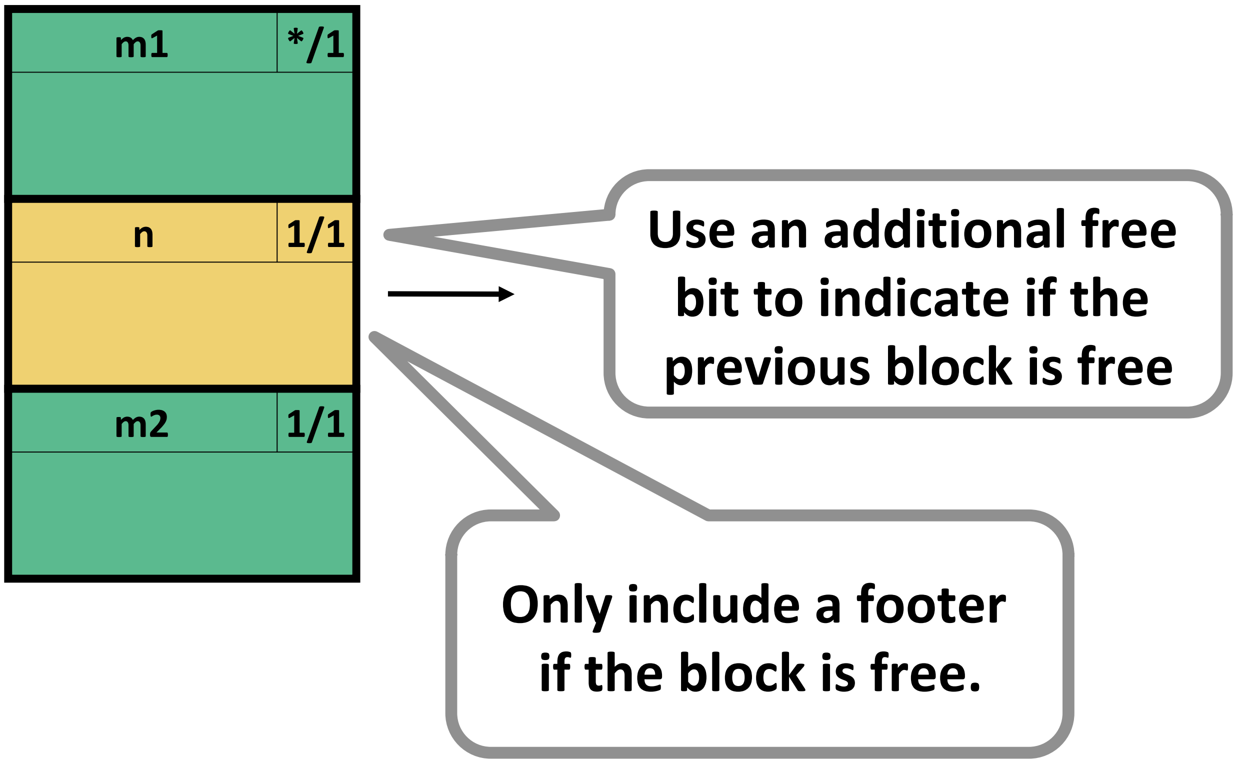


Disadvantages of Boundary Tags

- **Internal fragmentation (space used for bookkeeping)**
- **Can it be optimized?**
  - Which blocks need the footer tag?
  - What does that mean?

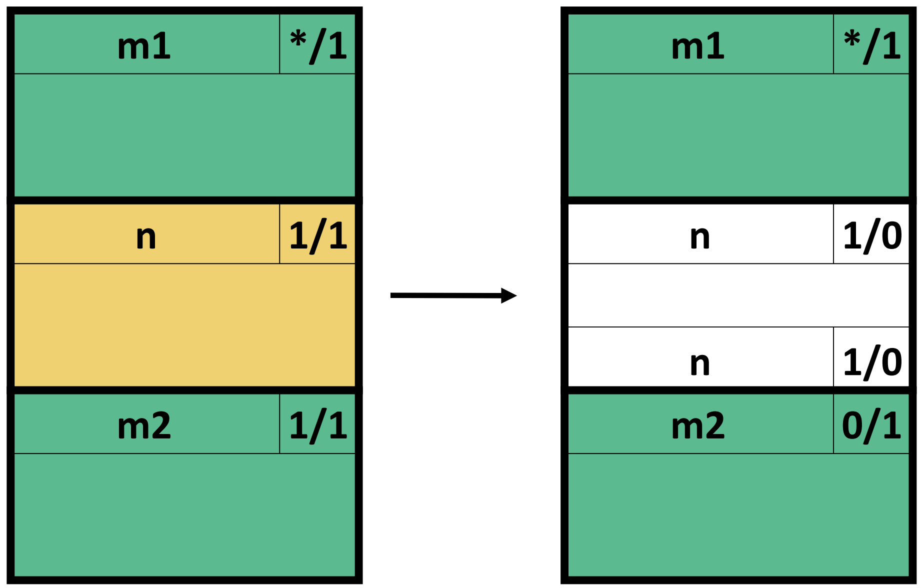
- **Observation:**
  - When a block is NOT free, you simply need to know that it is not free.
  - You need to know the size of a block ONLY when it is free so that you can coalesce with it.
- **Proposal: let's add footer for the free blocks only, and keep a bit in the header to indicate whether previous block is free or not!**

Constant Time Coalescing When Allocated Block Has No Footer (Case 1)



LSB with value '1/1' indicates that previous/this block is in use.

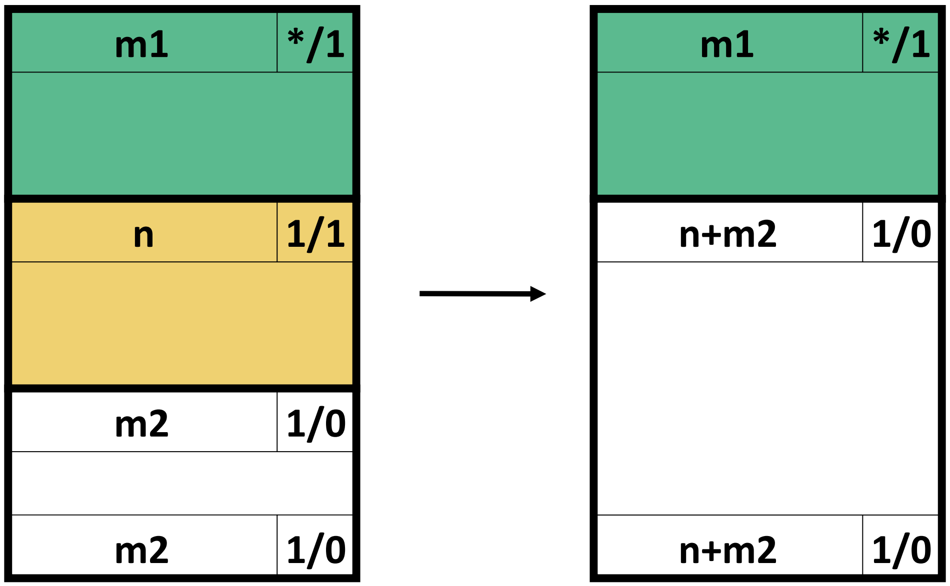
Constant Time Coalescing When Allocated Block Has No Footer (Case 1)



Need to modify next block's bit when allocate / free a new block.

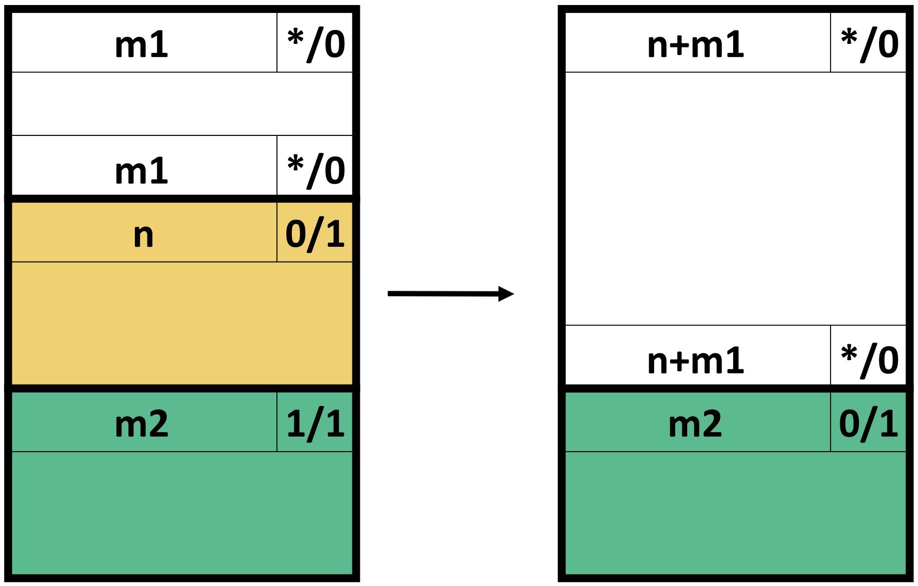
LSB with value '1/1' indicates that previous/this block is in use.

Constant Time Coalescing When Allocated Block Has No Footer (Case 2)

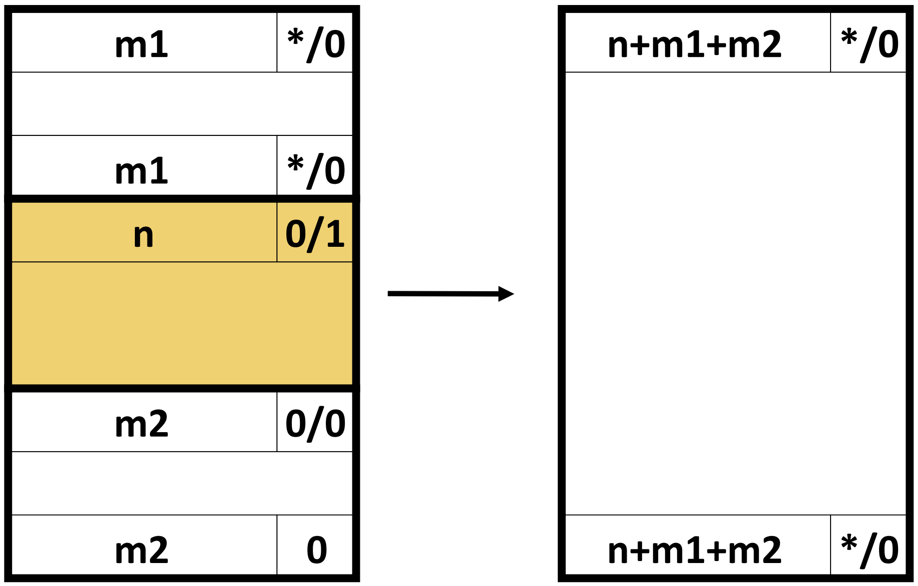


LSB with value '1/1' indicates that previous/this block is in use.

Constant Time Coalescing When Allocated Block Has No Footer (Case 3)



Constant Time Coalescing When Allocated Block Has No Footer (Case 4)



Practice Problem

Assuming that my memory allocator is using an implicit free list to keep track of free blocks. A free block contains both header and footer (4 bytes each) , and an allocated block contains a header. My memory allocator always round up block size to be multiple of 8.

What's the minimum block size (in bytes) does each of the call return?

malloc call	minimum block size
`malloc(12)`	
`malloc(5)`	
`malloc(1)`	

Practice Problem

Assuming that my memory allocator is using an implicit free list to keep track of free blocks. A free block contains both header and footer (4 bytes each) , and an allocated block contains a header. My memory allocator always round up block size to be multiple of 8.

What's the minimum block size (in bytes) does each of the call return?

malloc call	minimum block size
`malloc(12)`	16
`malloc(5)`	16
`malloc(1)`	8

Implicit Lists: Summary

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

- **Not used in practice for `malloc/free` (too slow)**
  - used in many special purpose applications

- **Concepts of splitting & coalescing are general to *all* allocators**