Dynamic Memory Allocation

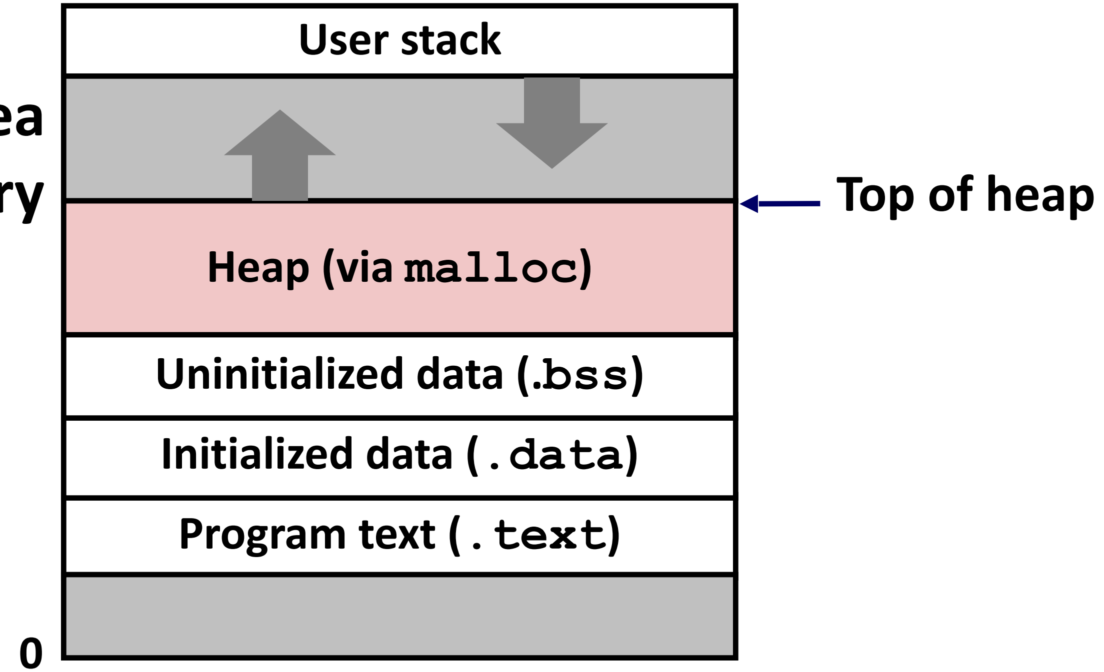
Today

- **Basic concepts**
- Implicit free lists

Dynamic Memory Allocation

- Programmers use ***dynamic memory allocators*** (such as `malloc`) to allocate memory at run time.

- Dynamic memory allocators manage an area of process virtual 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 space
    - E.g., `malloc` and `free` in C
  - ***Implicit allocator:*** application allocates, but does not free space
    - E.g. garbage collection in Java

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

malloc

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

`void *malloc(size_t size)`

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
- 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`**

Other functions

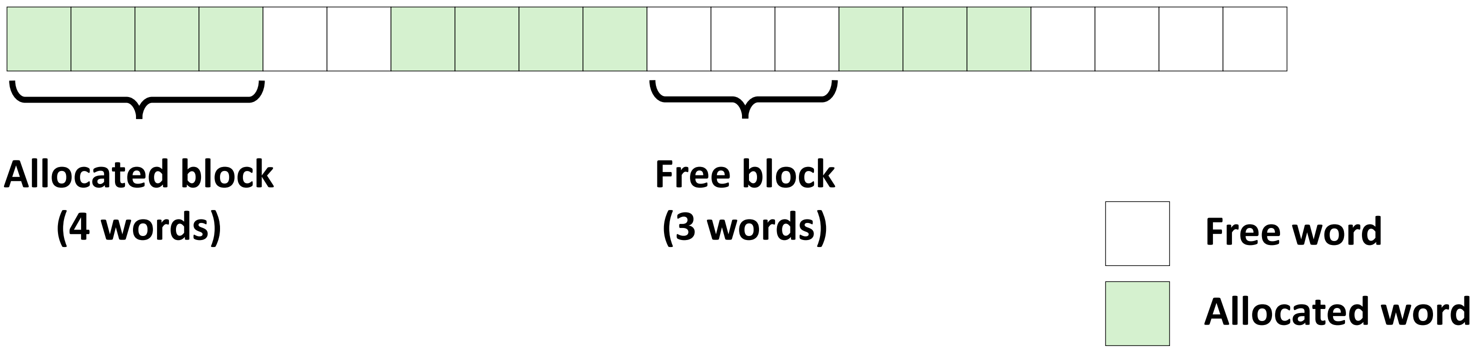
- **`calloc`**: Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`**: Changes the size of a previously allocated block.
- **`sbrk`**: Used internally by allocators to grow or shrink the heap

malloc Example

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

void foo(int n) {
    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;


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

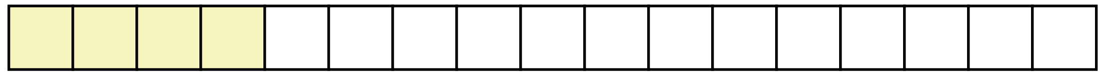
Assumptions Made in This Lecture

- Memory is word addressed.
- Words are int-sized.

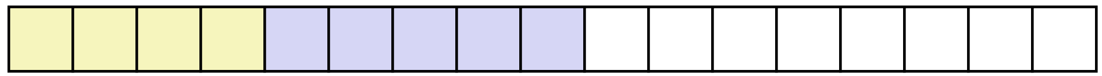


Allocation Example

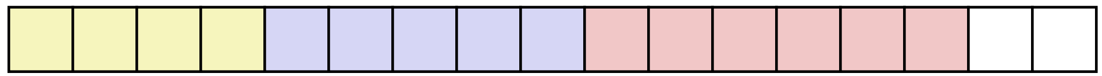
`p1 = malloc(4)`



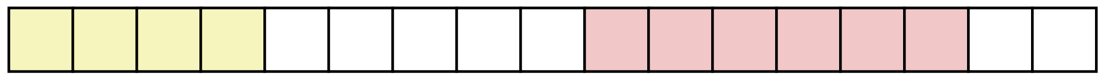
`p2 = malloc(5)`



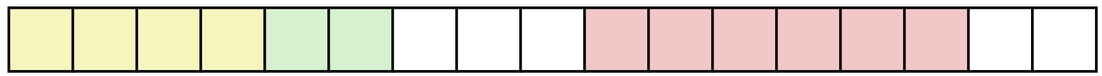
`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(2)`



Constraints

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

- **Allocators**
  - Must respond immediately to **`malloc`** requests
    - *i.e.*, can't reorder or buffer requests
  - Must align blocks so they satisfy all alignment requirements
    - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
  - Can't move the allocated blocks once they are **`malloc`**'d
    - *i.e.*, compaction is not allowed

Performance Goal

- **Goals: maximize <u>throughput</u> and <u>memory utilization</u>**
  - These goals are often conflicting

- **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 calls/second

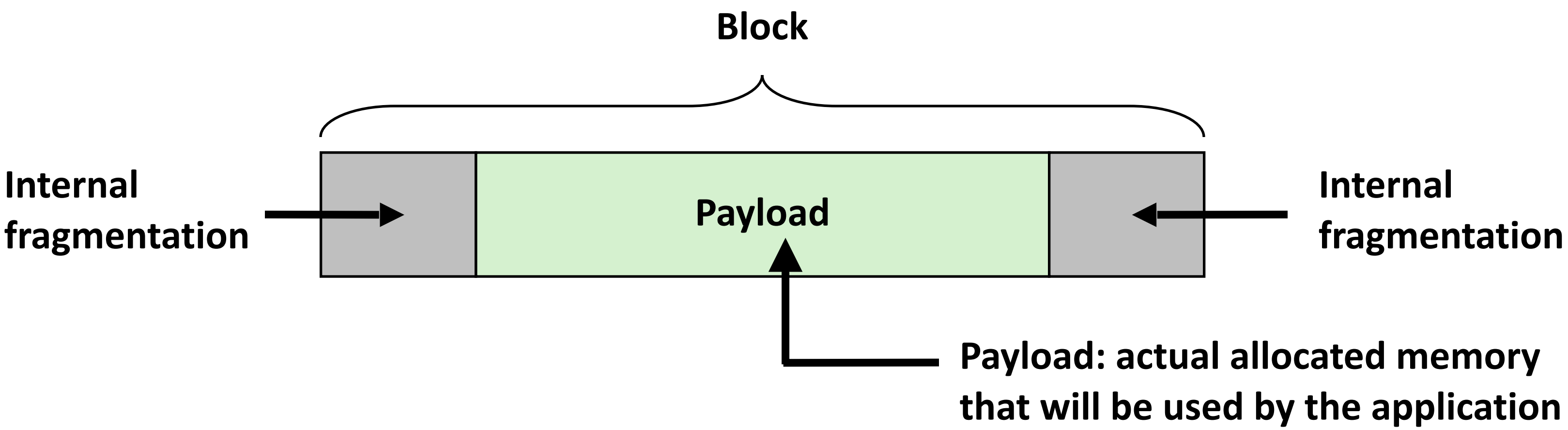
- **Memory utilization**
  - How much memory space is used to manage the heap?
  - How much memory can be used by the application?

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**



- **Depends only on the pattern of *previous* requests**
  - Thus, easy to measure (sum of differences between allocated blocks and payloads)

External Fragmentation

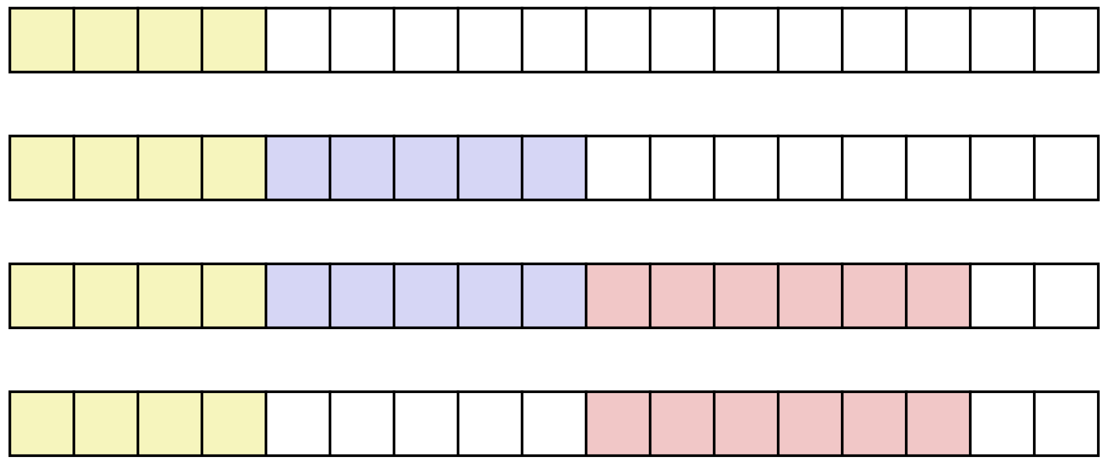
- **Occurs when there is enough aggregate heap memory, but no single free block is large enough**

`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`



`p4 = malloc(6)`

Oops! (what would happen now?)

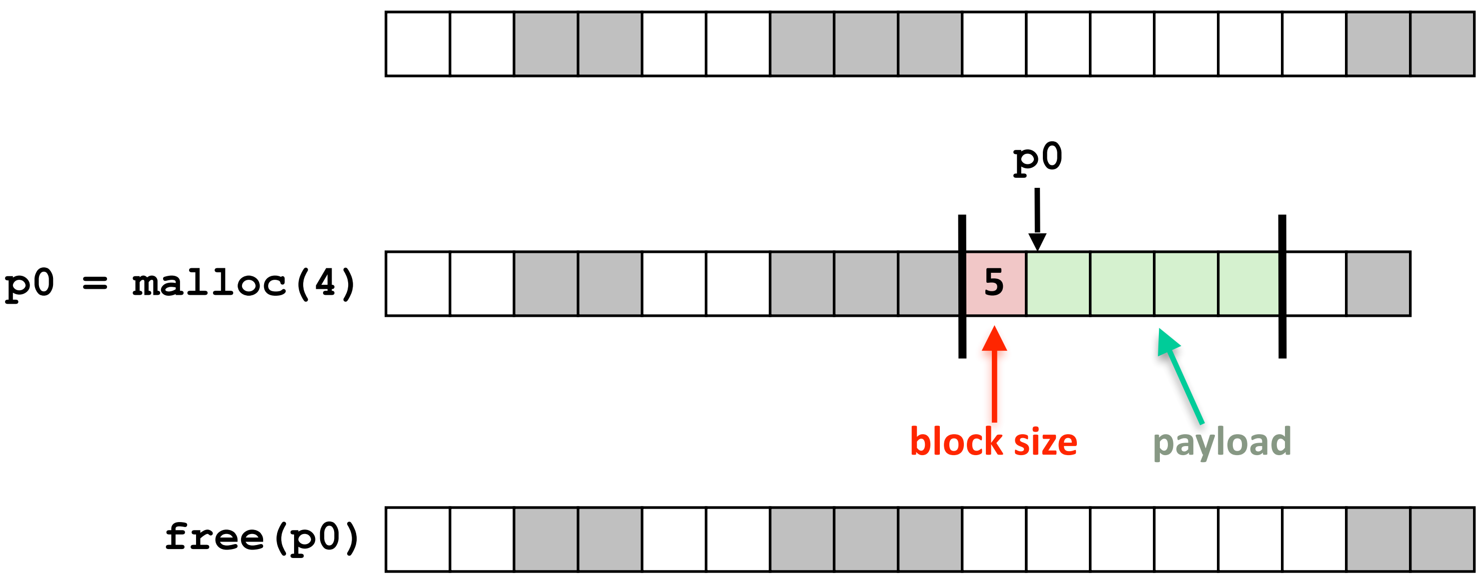
- **Depends on the pattern of future requests**
  - Thus, difficult to measure (if future request <=5, then no fragmentation, if >5, then fragmentation)

Implementation Issues

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

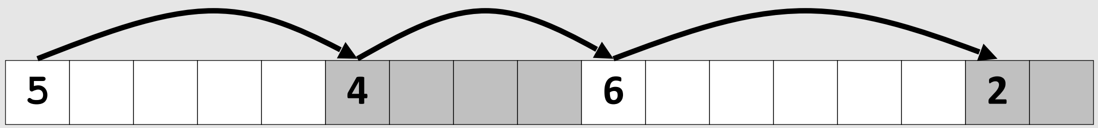
Knowing How Much to Free

- **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

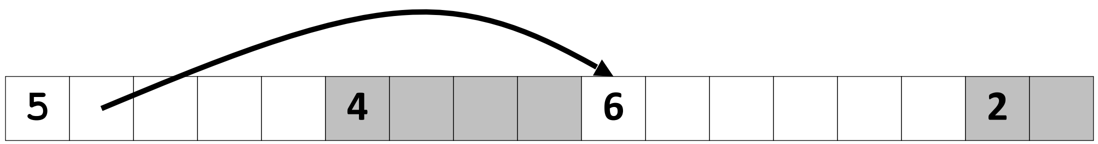


Keeping Track of Free Blocks

- 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***

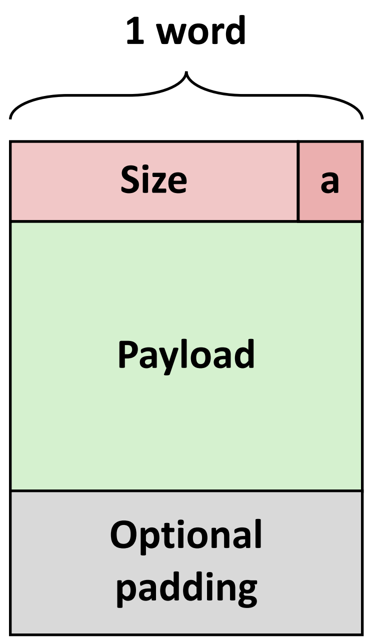
Today

- Basic concepts
- **Implicit free lists**

Method 1: Implicit List

- For each block we need both size and allocation status

Format of allocated and free blocks



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

Size: block size

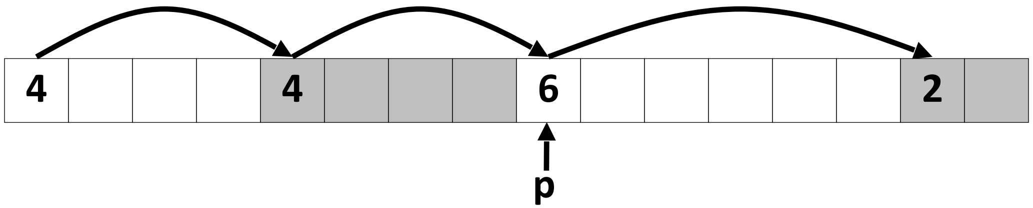
Payload: application data (allocated blocks only)

Implicit List: Finding a Free Block

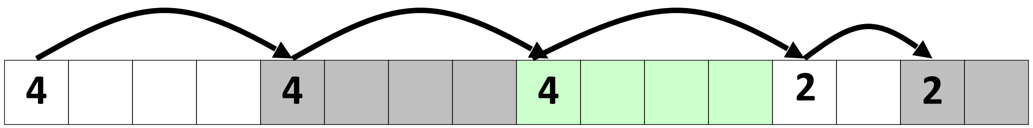
- ***First fit:***
  - Search list from beginning, choose ***first*** free block that fits:
  - Can take linear time in total number of blocks (allocated and free)
- ***Next fit:***
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- ***Best fit:***
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps fragments small—usually improves memory utilization
  - Will typically run slower than first fit

Implicit List: Allocating in Free Block

- **Allocating in a free block: *splitting***
  - Since allocated space might be smaller than free space, we might want to split the block

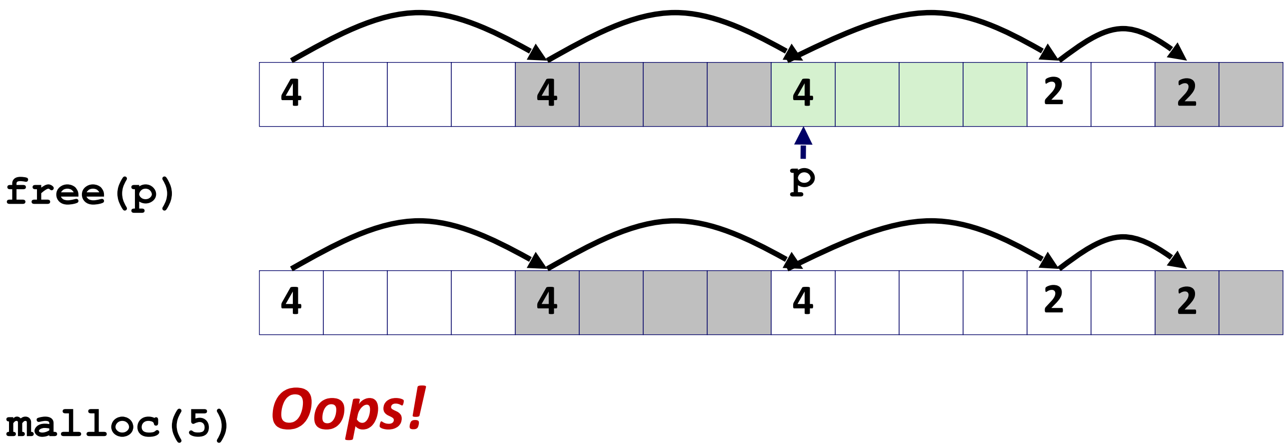


`malloc(4)`



Implicit List: Freeing a Block

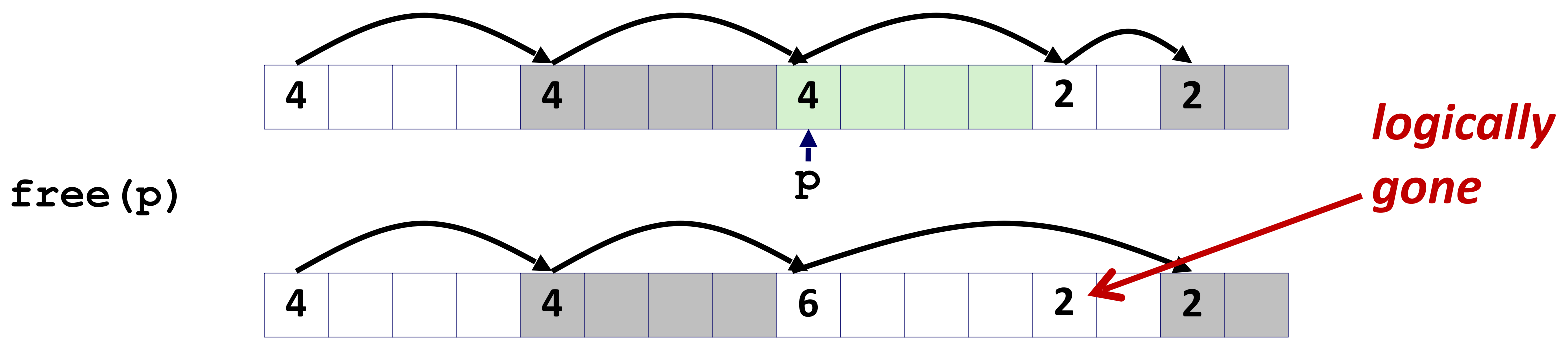
- **Simplest implementation:**
  - Need only clearing the "allocated" flag
  - But can lead to "false fragmentation"



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

Implicit List: Coalescing

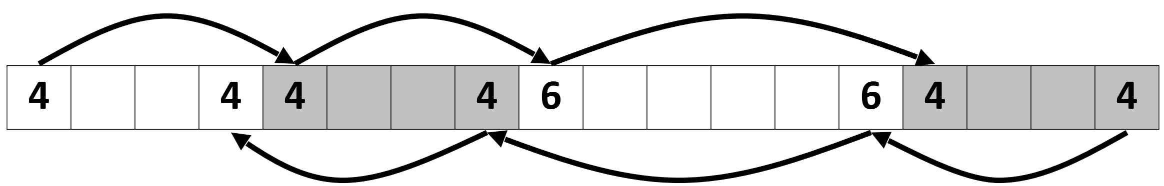
- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block



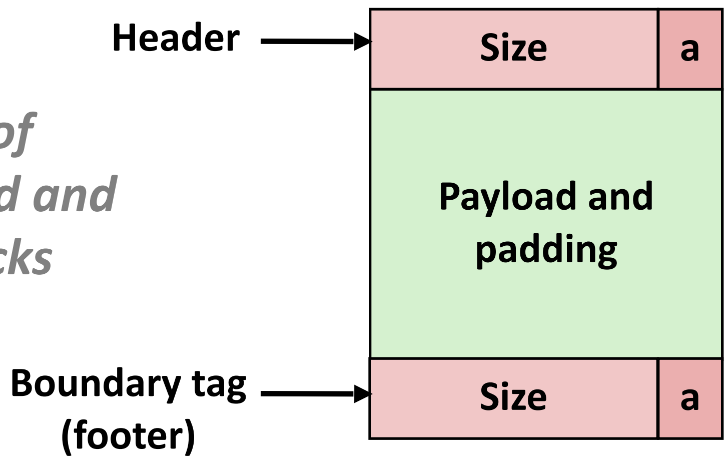
  - But how do we coalesce with *previous* block?

Implicit List: Bidirectional Coalescing

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



Format of allocated and free blocks

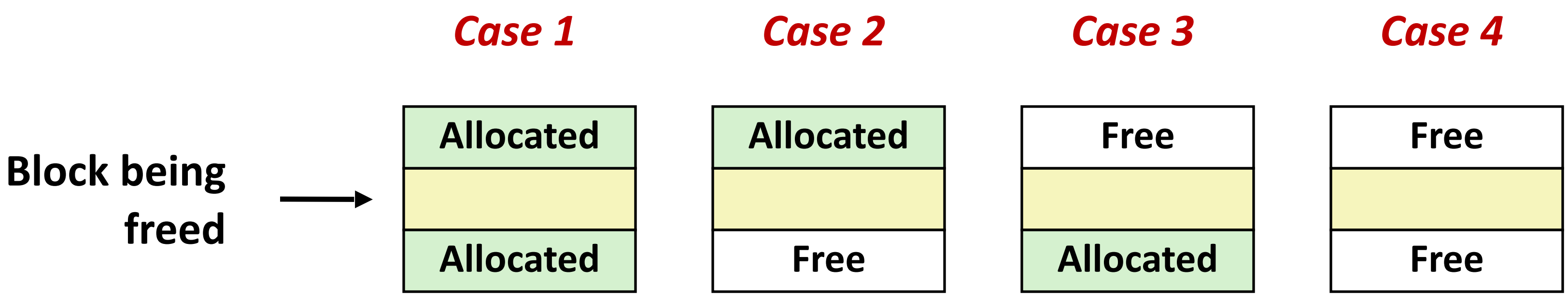


a = 1: Allocated block
a = 0: Free block

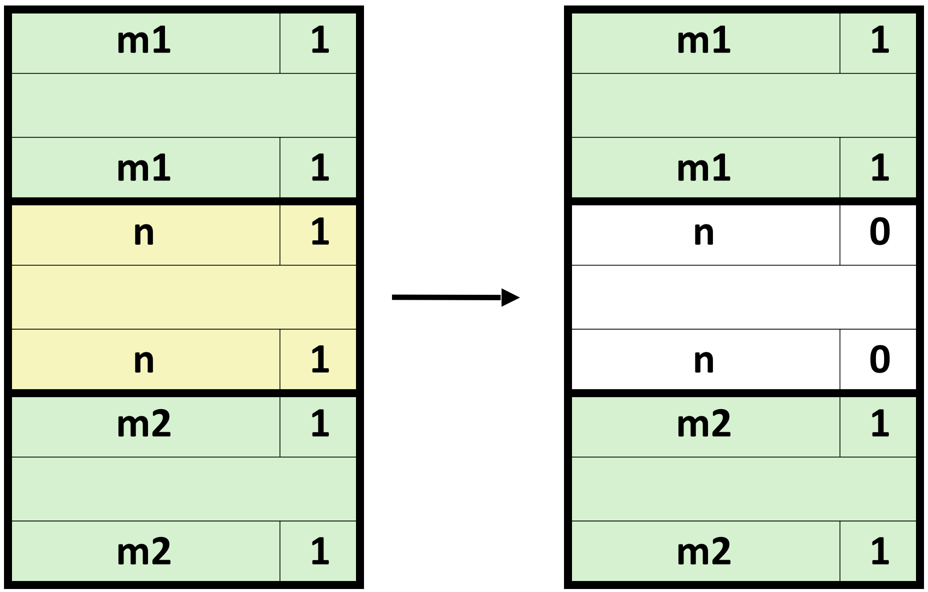
Size: Total block size

Payload: Application data (allocated blocks only)

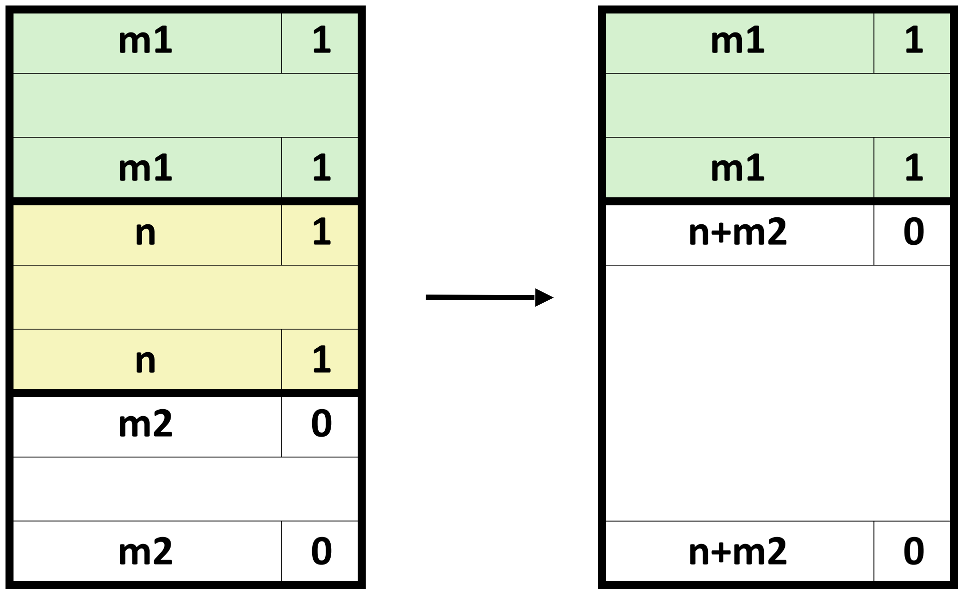
Constant Time Coalescing



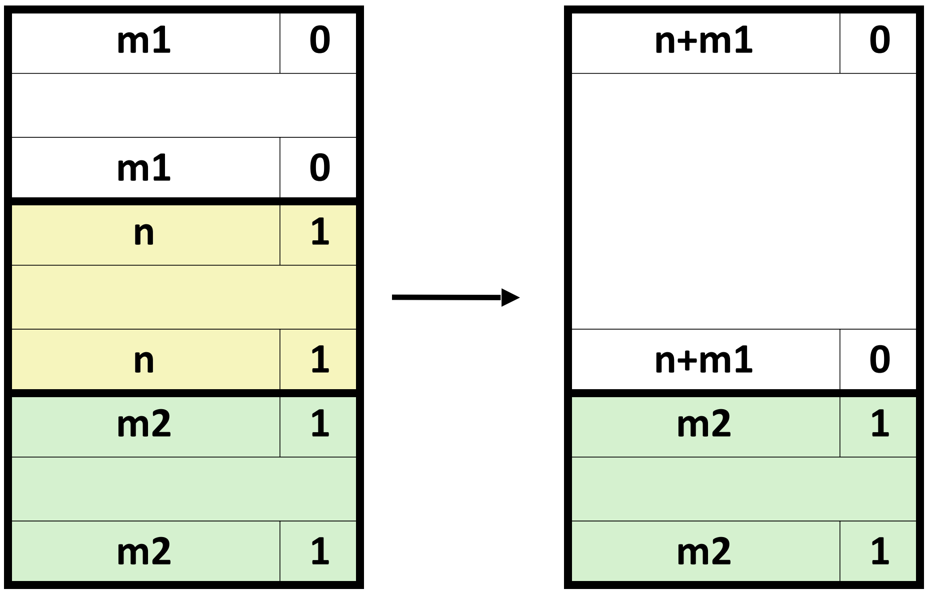
Constant Time Coalescing (Case 1)



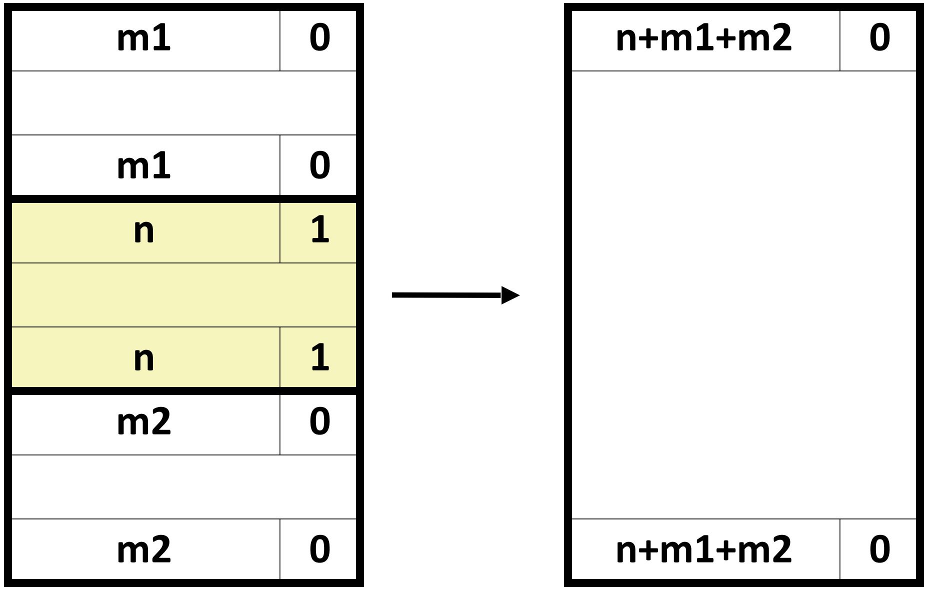
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Coalescing Policy

- **Coalescing policy:**
  - ***Immediate coalescing:*** coalesce each time **`free`** is called
  - ***Deferred coalescing:*** try to improve performance of **`free`** by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for **`malloc`**
    - Coalesce when the amount of external fragmentation reaches some threshold