Dynamic Memory Allocation: Basic Concepts

B&O Readings: 9.9

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

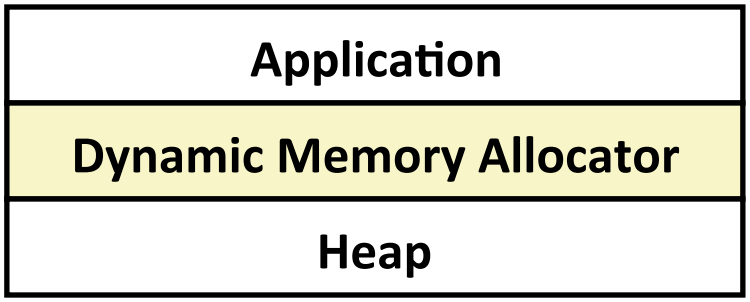
Note: these slides were originally created by Markus Püschel at Carnegie Mellon University

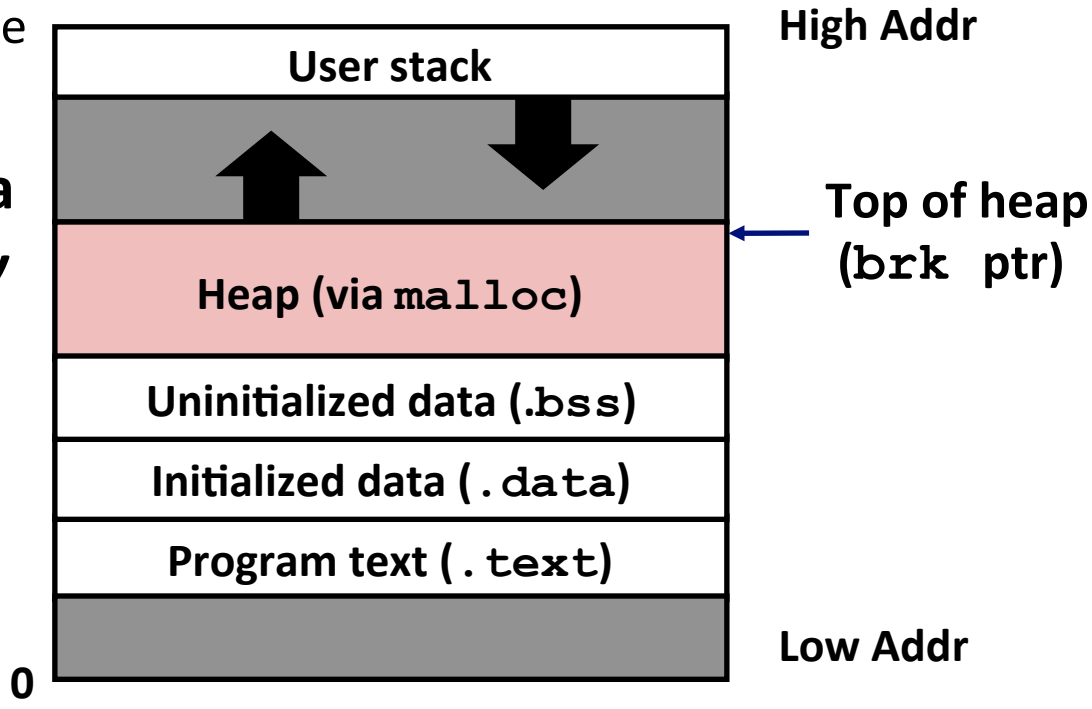
Today

- **Basic concepts**
- **Implicit free lists**

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 *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
    - 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`**

Other functions

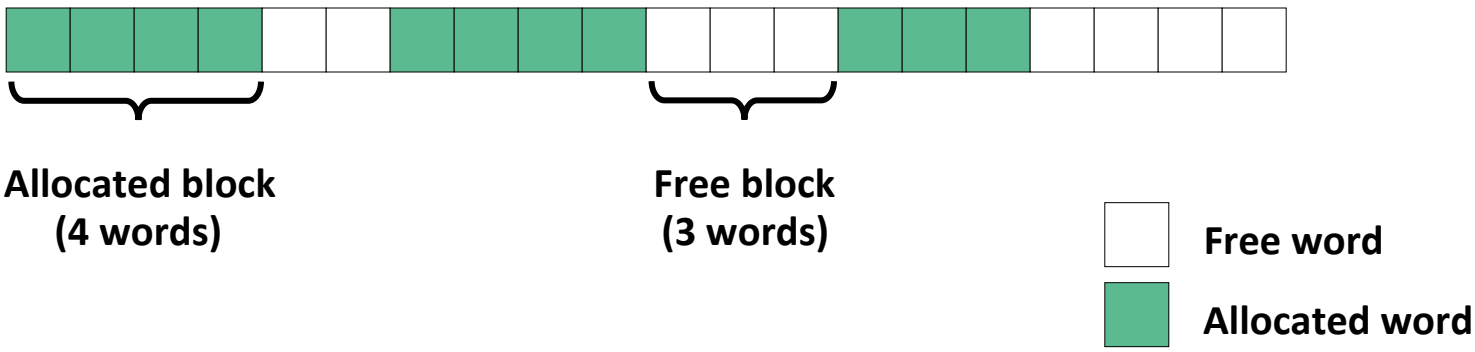
- **`calloc`:** initializes allocated block to zero
- **`realloc`:** changes size of a previously allocated block
- **`sbrk`:** used internally by allocators to grow or shrink heap

`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;


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

Assumptions Made in This Lecture

- Memory is word addressed (each word can hold a pointer)



Allocation Example

```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

```
free(p2)
```

```
p4 = malloc(8)
```

Constraints

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

- **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
  - Must align blocks so they satisfy all alignment requirements
    - 8 byte alignment for GNU **`malloc`** (**`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

Performance Goal #1: Throughput

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

- **Maximize 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 Goal #2: Memory Utilization

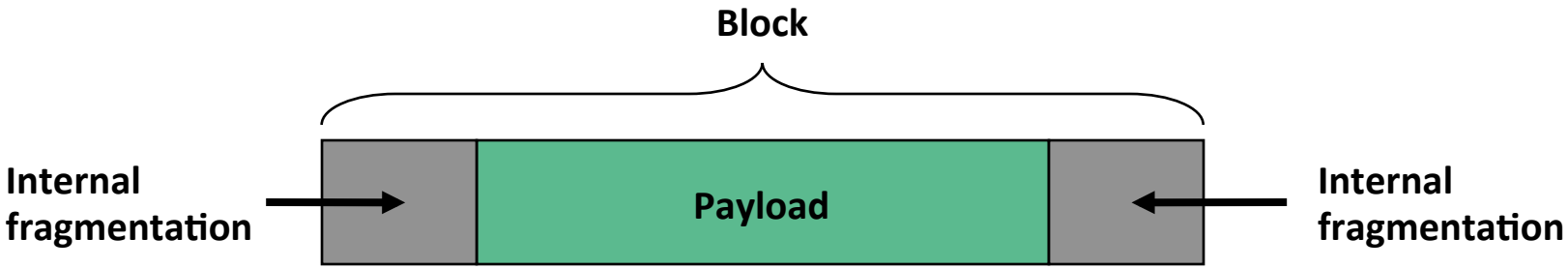
- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- Terminology:
  - **`malloc(p)`** results in a block with a **payload** of **`p`** bytes
  - after R_k, **aggregate payload** P_k = sum of all current (non-freed) payloads
  - **H_k** : current heap size (can only increase)
  - **Peak memory utilization**: highest ratio between the aggregate payload and the size of the heap (best possible ratio = 1)
  - *Use what you have. Don't be wasteful.*
- **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

Internal Fragmentation

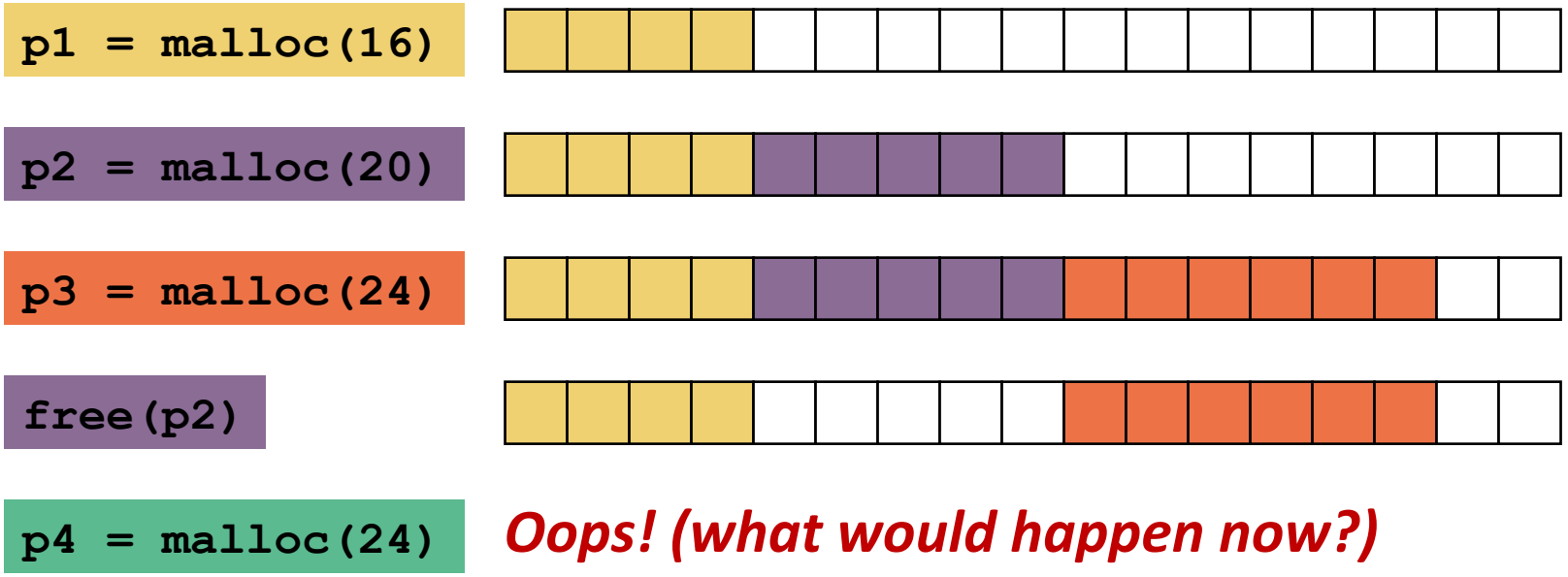
- **For a given block, *internal fragmentation* occurs if payload is smaller than block size**



- **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

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 measure

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?

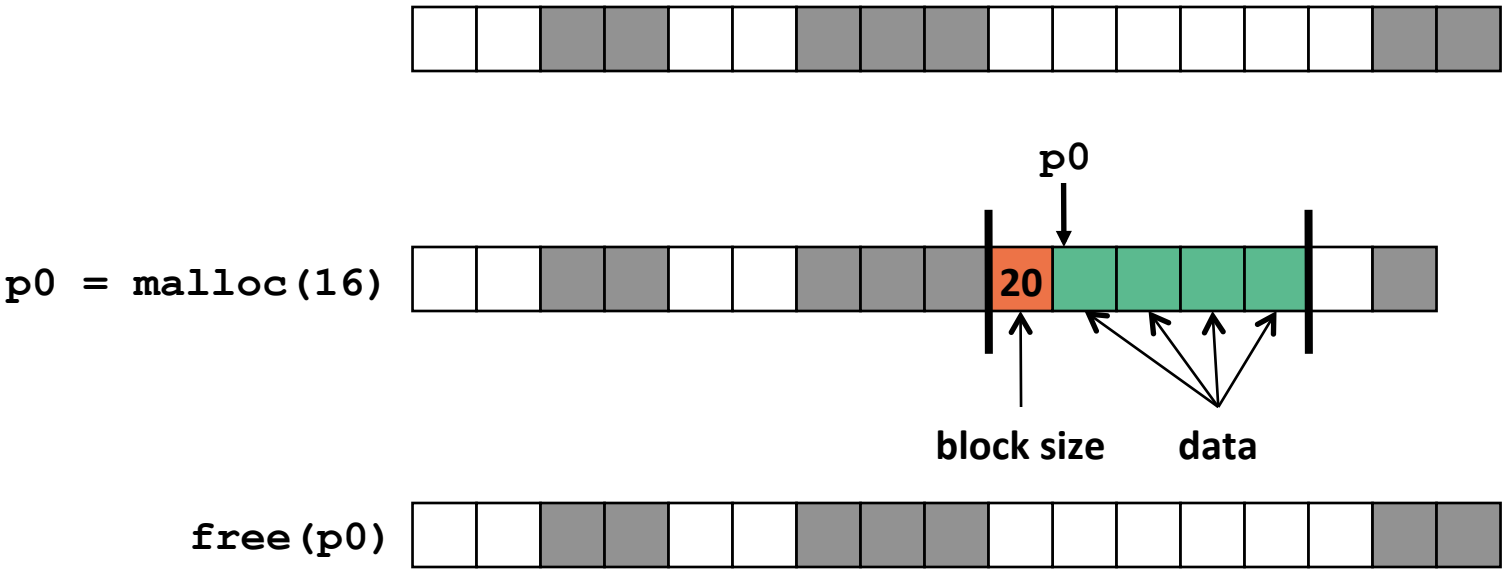
3. When allocating a structure that is smaller than the free block it is placed in, what do we do with the extra space?

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

5. How do we reinsert freed block?

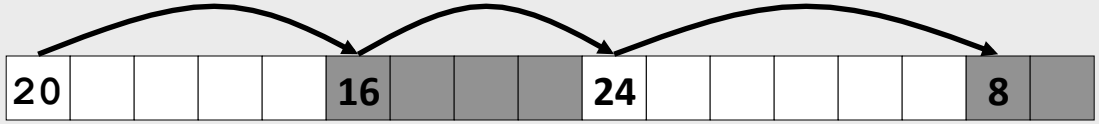
Q1: Knowing How Much to Free

- **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

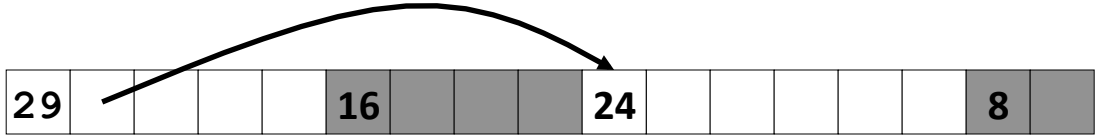


Q2: Keeping Track of Free Blocks

- **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

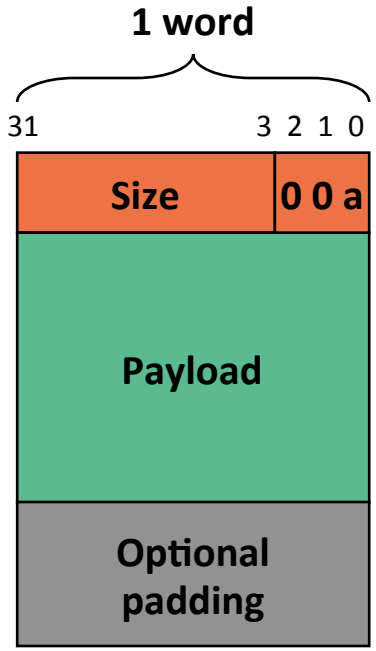
Today

- Basic concepts
- **Implicit free lists**

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

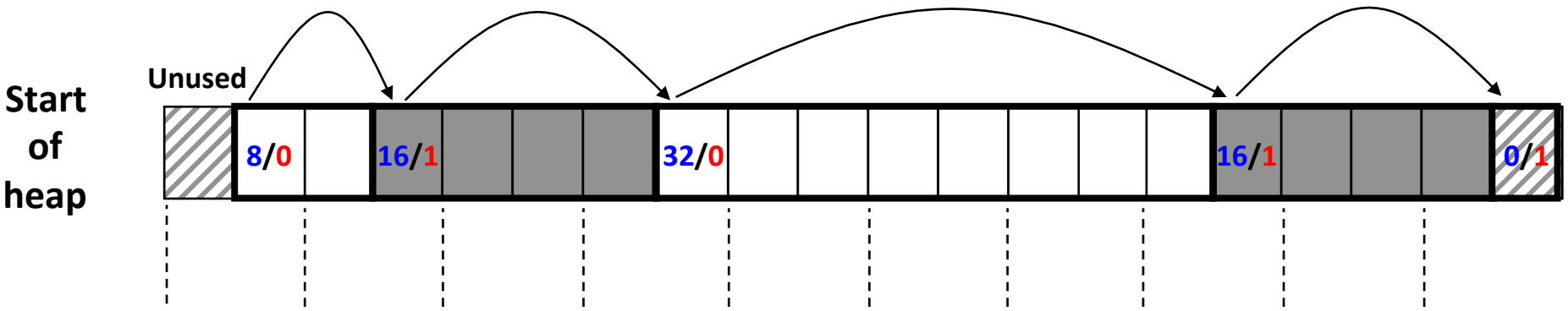


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



Double-word aligned

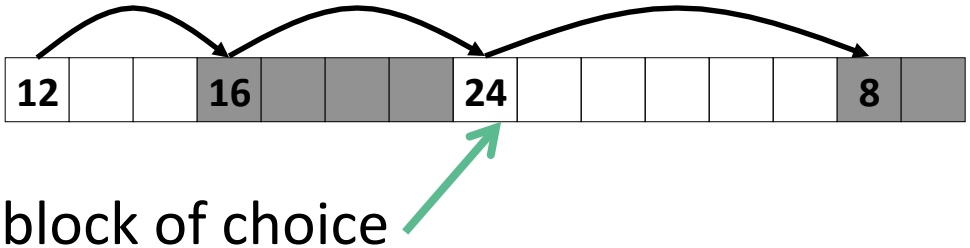
Allocated blocks: shaded grey
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

Q4: 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

Q3: Implicit List: Allocating in Free Block

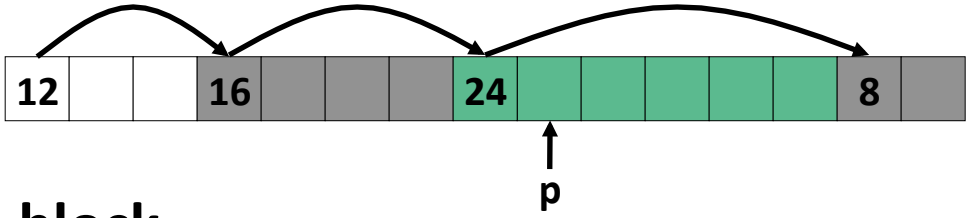
Suppose we need to allocate 3 words



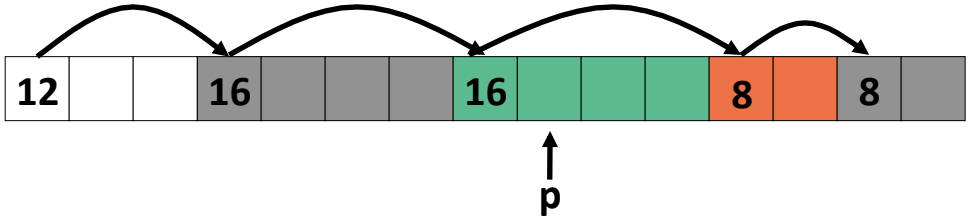
This is our free block of choice

Two options:

1. **Allocate the whole block (internal fragmentation!)**

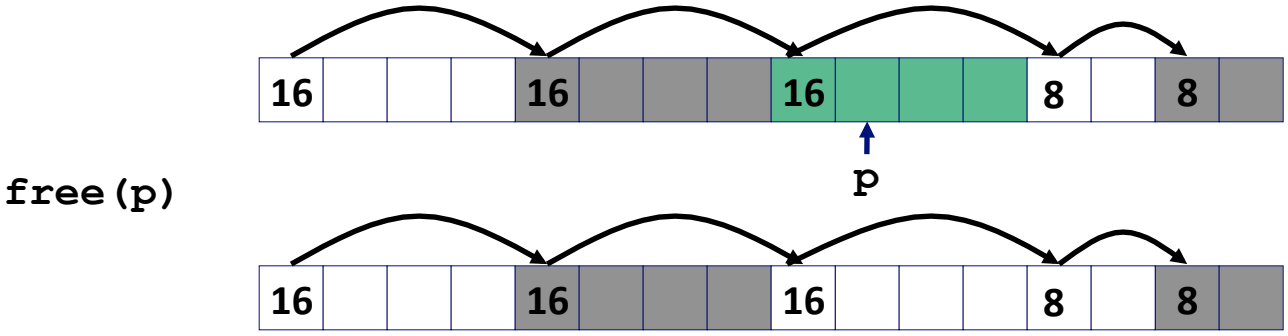


2. **Split the free block**



Q5: Implicit List: Freeing a Block

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

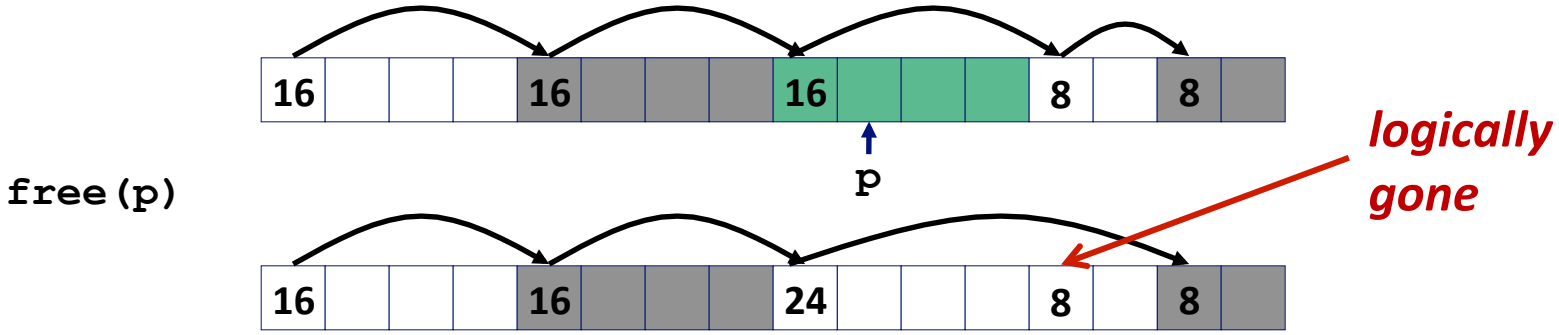


`malloc(20)` ***Oops!***

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

Implicit List: Coalescing

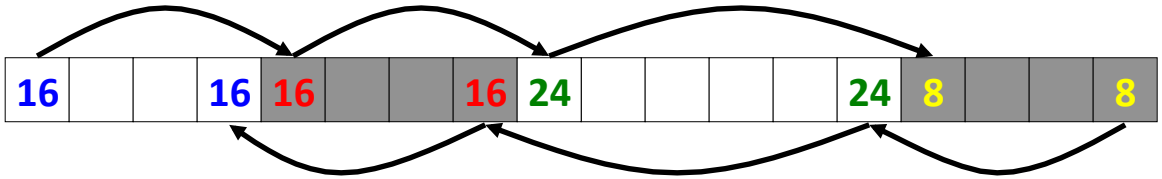
- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block

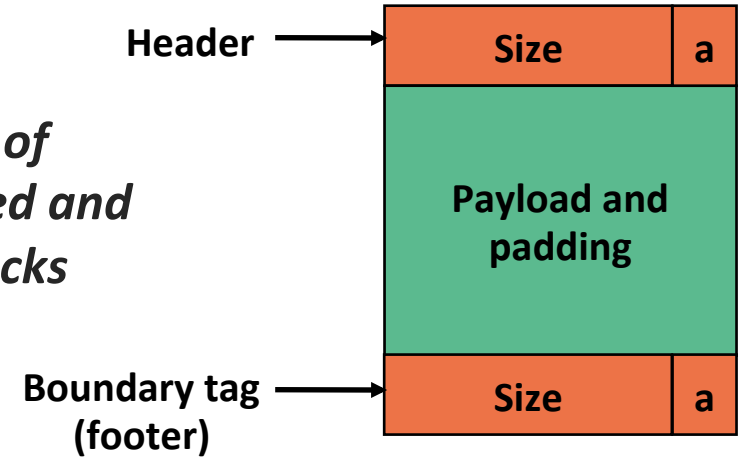


How do we coalesce with *previous* block?

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!





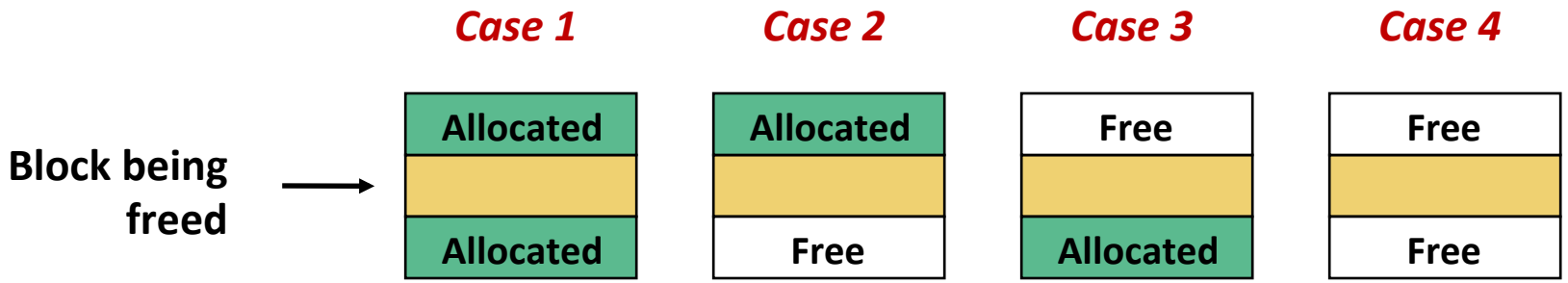
Format of allocated and free blocks

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

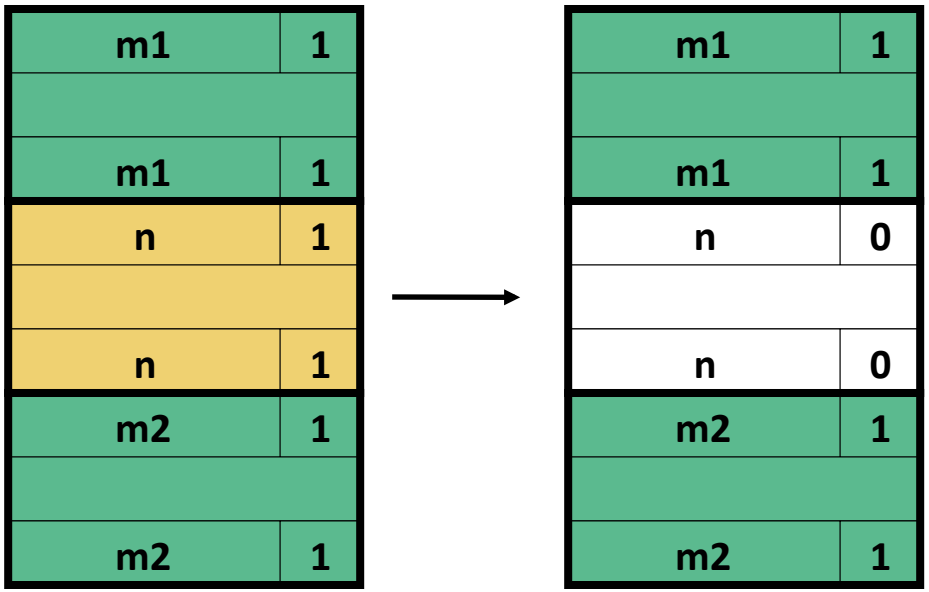
Size: Total block size

Payload: Application data (allocated blocks only)

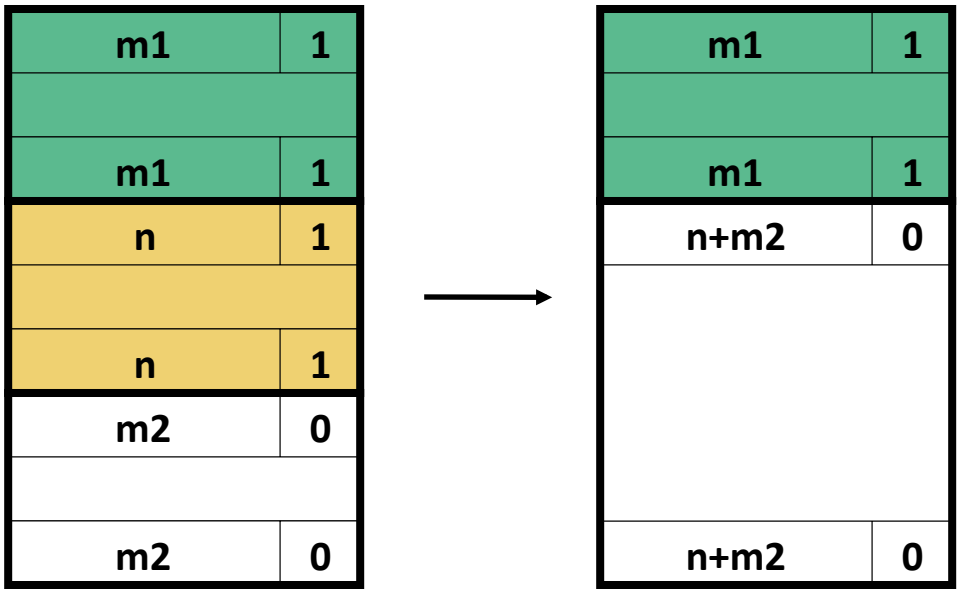
Constant Time Coalescing



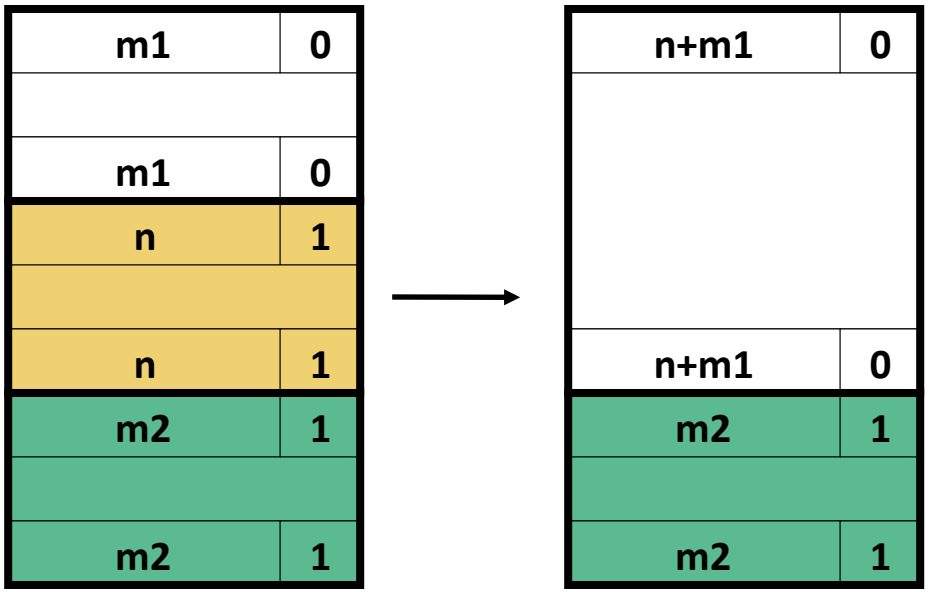
Constant Time Coalescing (Case 1)



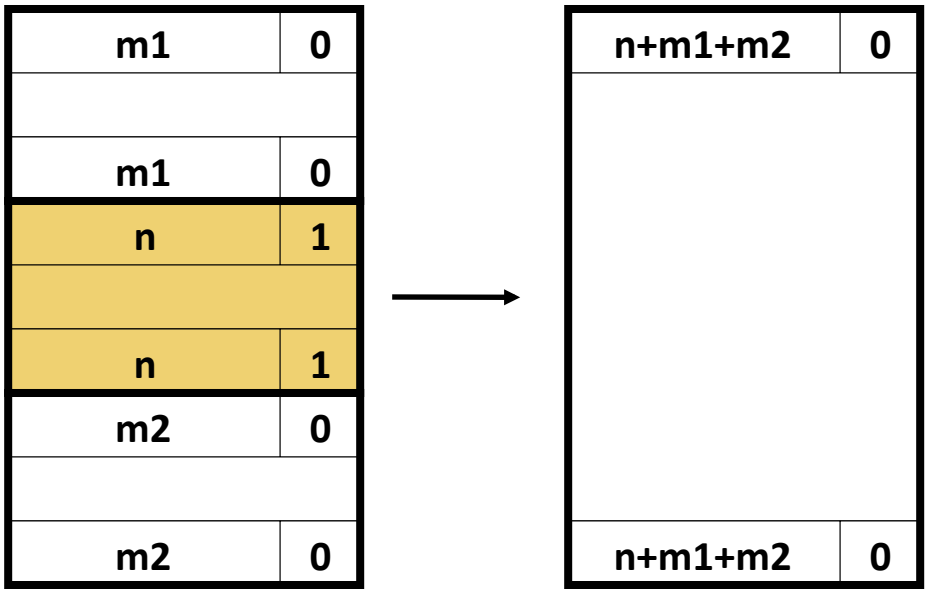
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)

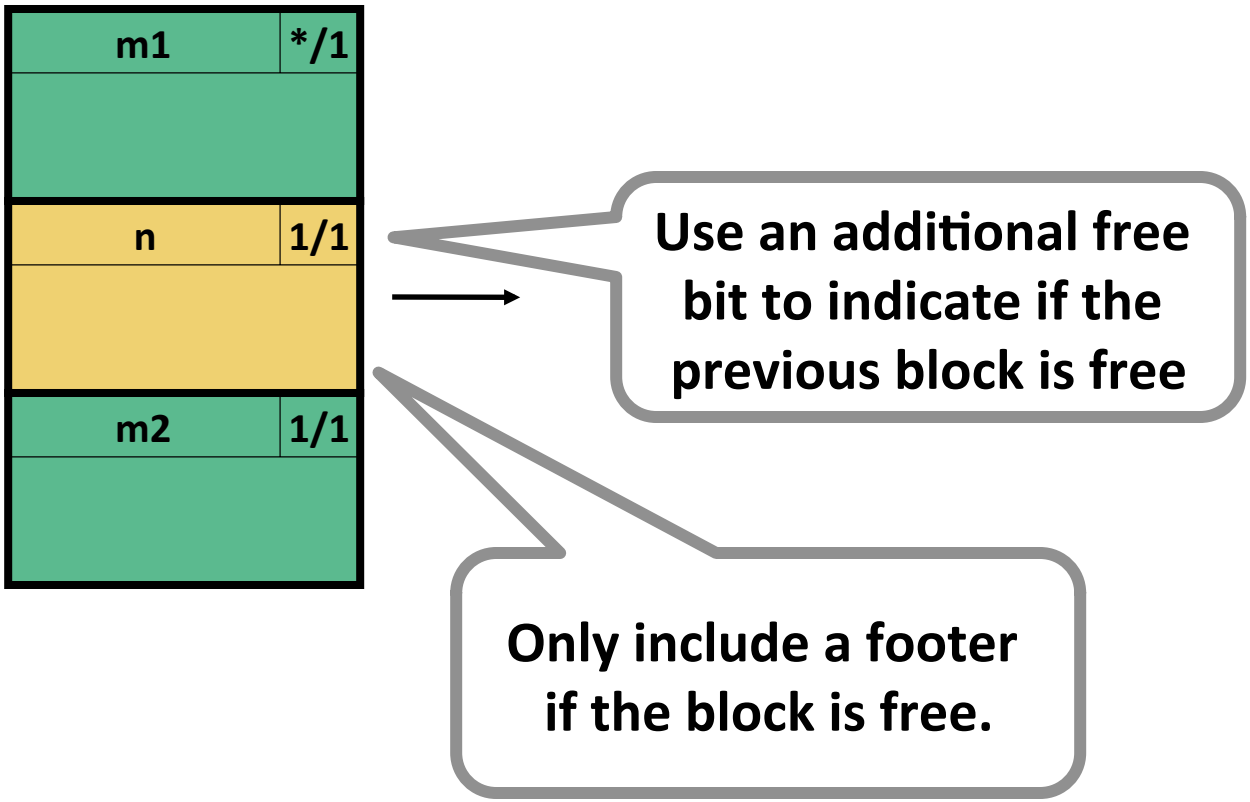


Disadvantages of Boundary Tags

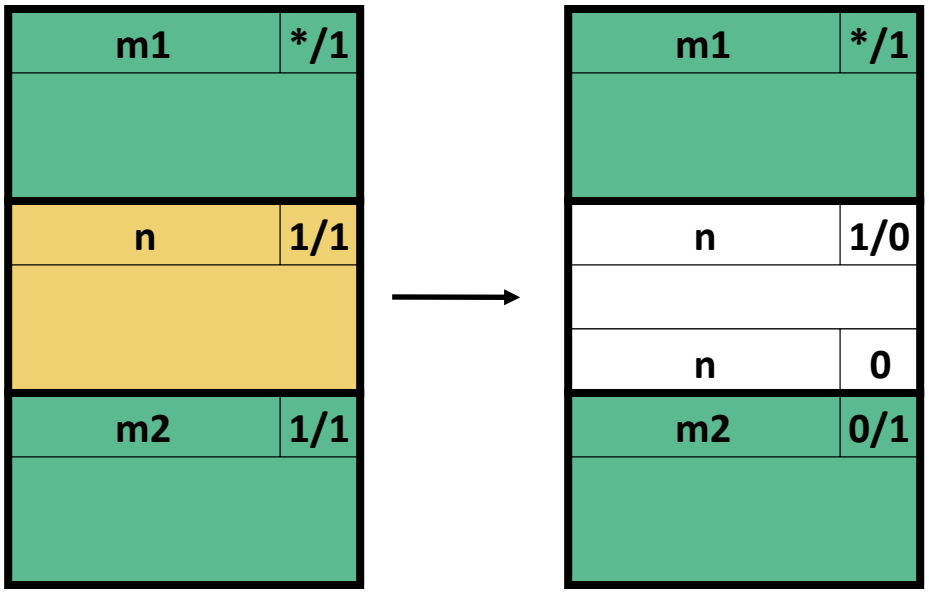
- **Internal fragmentation**

- **Can it be optimized?**
  - Which blocks need the footer tag?
  - What does that mean?

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

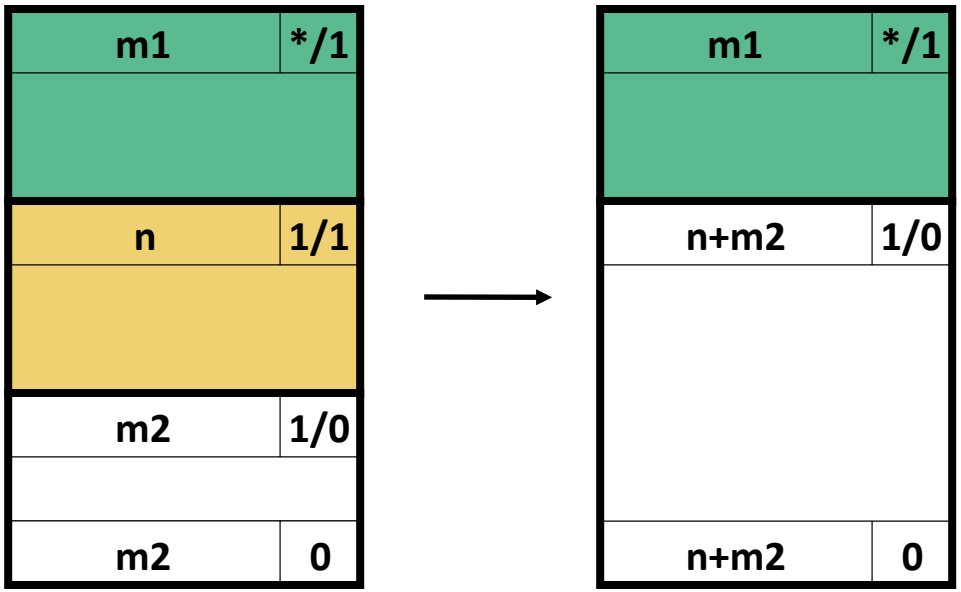


Constant Time Coalescing When Allocated Block Has No Footer (Case 1)

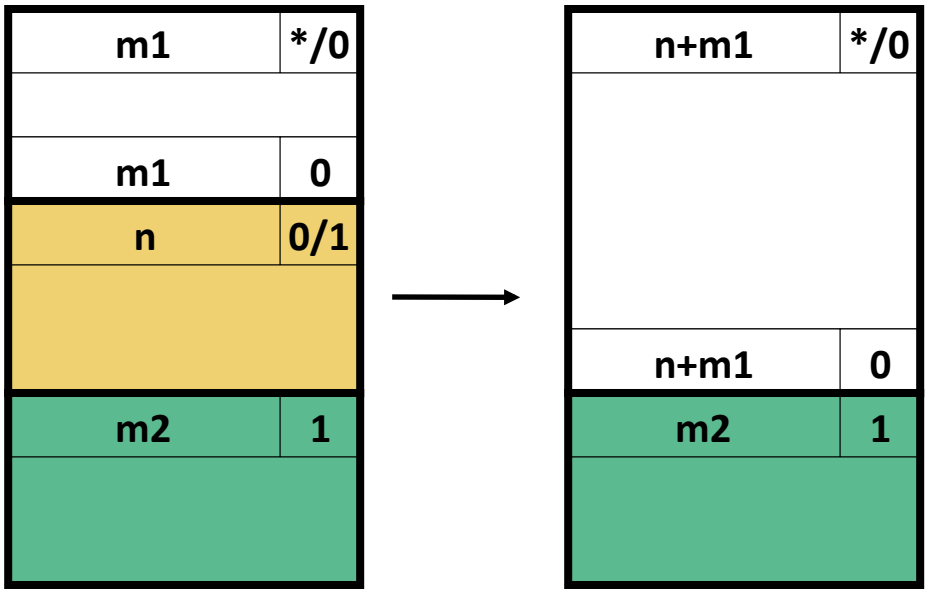


Need to modify next block's bit when allocate / free a new block.

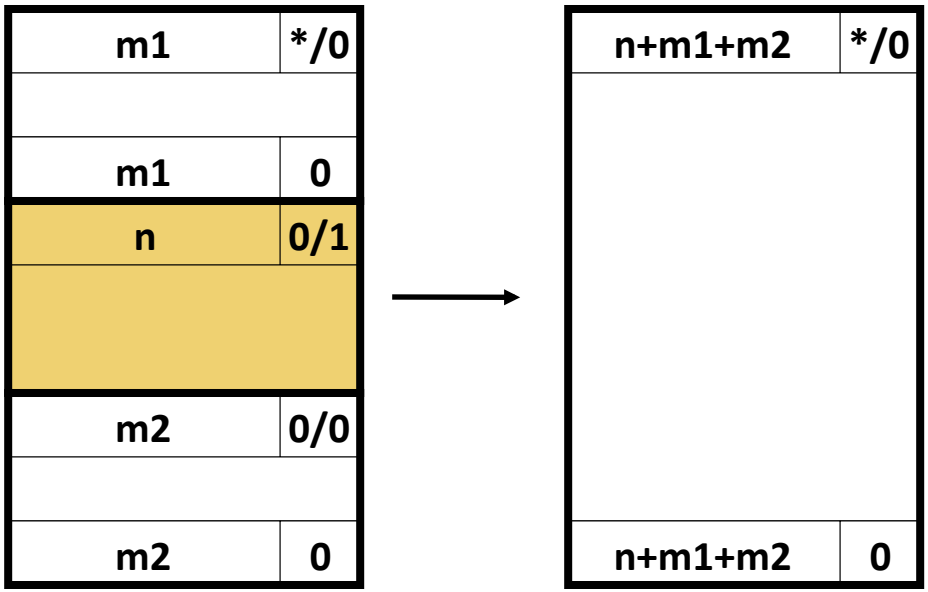
Constant Time Coalescing When Allocated Block Has No Footer (Case 2)



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



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

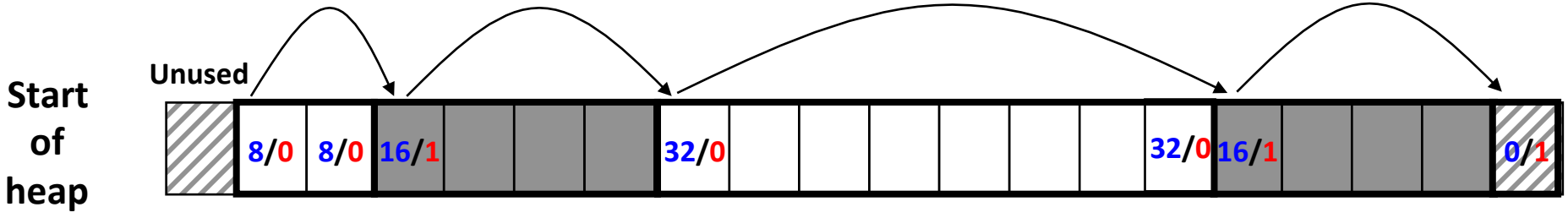


Implementing an Allocator with Implicit Free List

```
#define WSIZE 4 /* Word and header / footer size */
#define DSIZE 8 /* Double word size (8 bytes) */

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc)  ((size) | (alloc))

/* Read and write a word (4 byted) at address p */
#define GET(p)        (*(size_t *)(p))
#define PUT(p, val)   (*(size_t *)(p) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp)        ((char *)(bp) - WSIZE)
#define FTRP(bp)        ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* Given block ptr bp, compute address of next and prev blocks */
#define NEXT_BLKP(bp) \
((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp) \
((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

Implementing an Allocator with Implicit Free List

```
int mm_init(void)
{
    /* create the initial empty heap */
    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_listp, 0); /* alignment padding */
    PUT(heap_listp+WSIZE, PACK(DSIZE, 1)); /* prologue header */
    PUT(heap_listp+DSIZE, PACK(DSIZE, 1)); /* prologue footer */
    PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1));  /* epilogue header */
    heap_listp += DSIZE;

    /* Extend the heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}√
```



Summary of Key Allocator Policies

- **Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - Tradeoffs: throughput vs. fragmentation
  - ***Interesting observation*:** segregated free lists (more next lecture) approximate best fit placement policy without searching entire free list
- **Splitting policy:**
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- **Coalescing policy:**
  - ***Immediate coalescing:*** coalesce each time `free` is called
  - ***Deferred coalescing:*** improve performance by deferring until needed
    - Coalesce as you scan the free list for `malloc`
    - Coalesce when external fragmentation reaches some threshold

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**