



# Dynamic Memory Allocation: Basic Concepts

15-213/18-213/15-513/18-613:  
Introduction to Computer Systems  
15<sup>th</sup> Lecture, March 23<sup>rd</sup>, 2021

# Announcements

- Cache Lab extended 2 days
- Malloc Lab *checkpoint* extended one day

# Understanding this Error

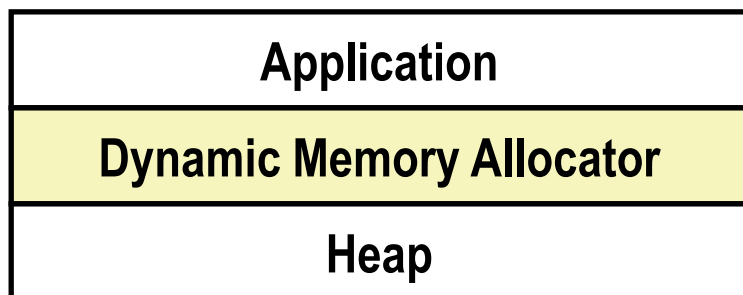
## ■ What causes this error? Why does it matter?

```
$ ./mm-corrupt
*** Error in `./mm-corrupt': free(): invalid next size (fast):
0x0000000000ffe010 ***
===== Backtrace: =====
/lib/x86_64-linux-gnu/libc.so.6(+0x777f5) [0x7f043efe67f5]
/lib/x86_64-linux-gnu/libc.so.6(+0x8038a) [0x7f043efef38a]
/lib/x86_64-linux-gnu/libc.so.6(cfree+0x4c) [0x7f043eff358c]
./mm-corrupt[0x400795]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0) [0x7f043ef8f840]
./mm-corrupt[0x400629]
===== Memory map: =====
...
```

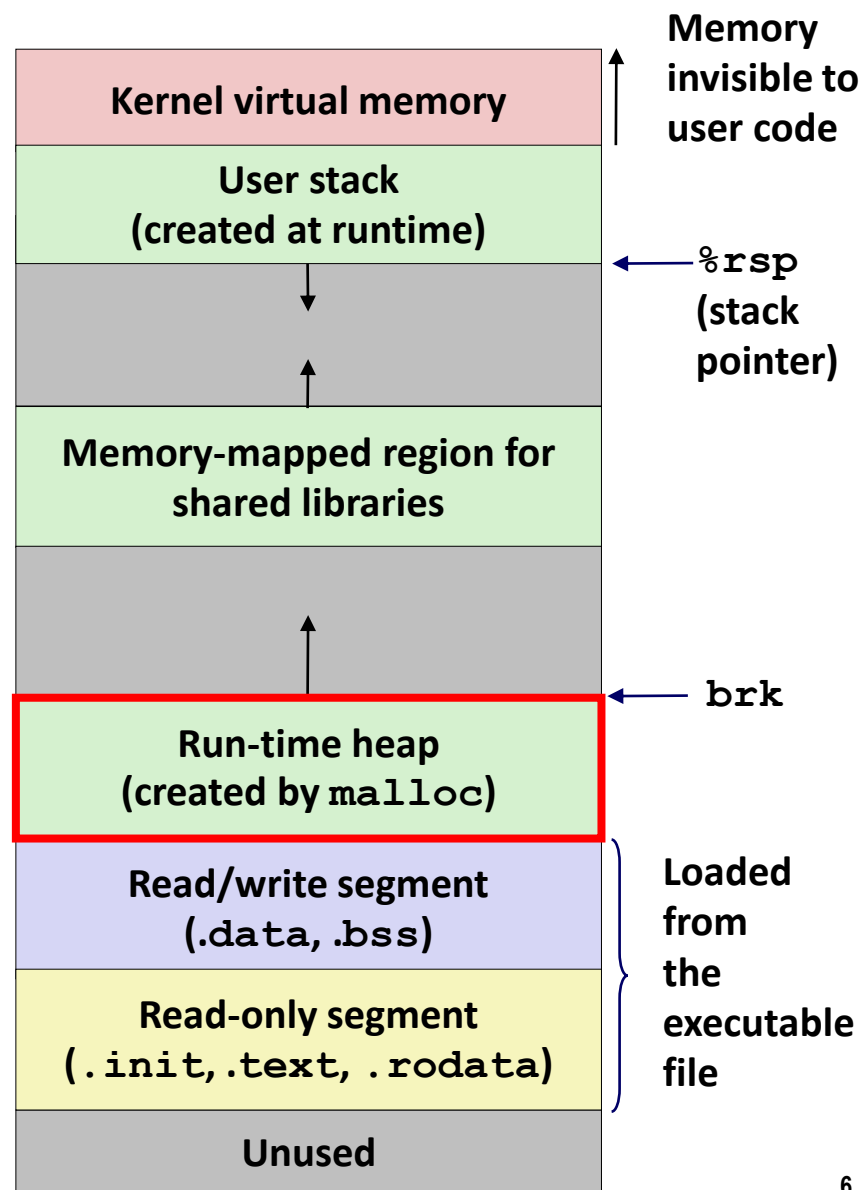
# Today

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

# Dynamic Memory Allocation



- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire virtual memory (VM) at run time.
  - for data structures whose size is only known at runtime
- Dynamic memory allocators manage an area of process VM 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., `new` and garbage collection in Java
- 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 aligned to a 16-byte boundary (on x86-64)
  - If **size == 0**, returns NULL
- Unsuccessful: returns NULL (0) and sets **errno** to ENOMEM

```
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**, **calloc**, 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(long n) {
    long i, *p;

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

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;
    /* Do something with p */
    . . .
    /* Return allocated block to the heap */
    free(p);
}
```

# Sample Implementation

## ■ Code

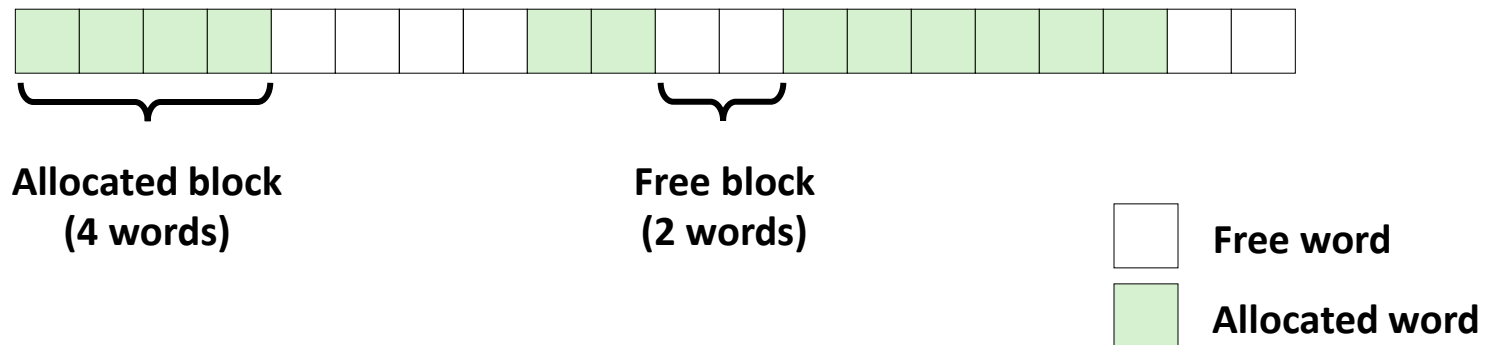
- File `mm-reference.c`
- Manages fixed size heap
- Functions `mm_malloc`, `mm_free`

## ■ Features

- Based on *words* of 8-bytes each
- Pointers returned by `malloc` are double-word aligned
  - Double word = 2 words
- Compile and run tests with command interpreter

# Visualization Conventions

- Show 8-byte words as squares
- Allocations are double-word aligned.



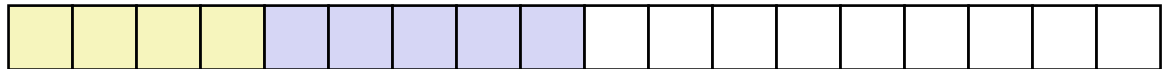
# Allocation Example (Conceptual)

```
#define SIZ sizeof(size_t)
```

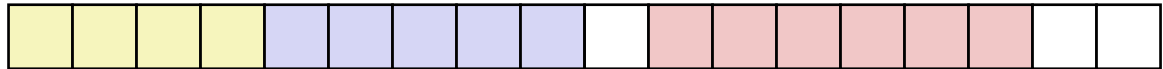
```
p1 = malloc(4*SIZ)
```



```
p2 = malloc(5*SIZ)
```



```
p3 = malloc(6*SIZ)
```



```
free(p2)
```



```
p4 = malloc(2*SIZ)
```



# Constraints

## ■ Applications

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

## ■ Explicit 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
  - 16-byte (x86-64) alignment on 64-bit systems
- 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?*

# Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- Goals: maximize throughput and peak memory utilization
  - 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 operations/second

# Performance Goal: Minimize Overhead

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- **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: Overhead after  $k+1$  requests**
  - Fraction of heap space *NOT* used for program data
  - $O_k = H_k / (\max_{i \leq k} P_i) - 1.0$

# Benchmark Example

## ■ Benchmark

### syn-array-short

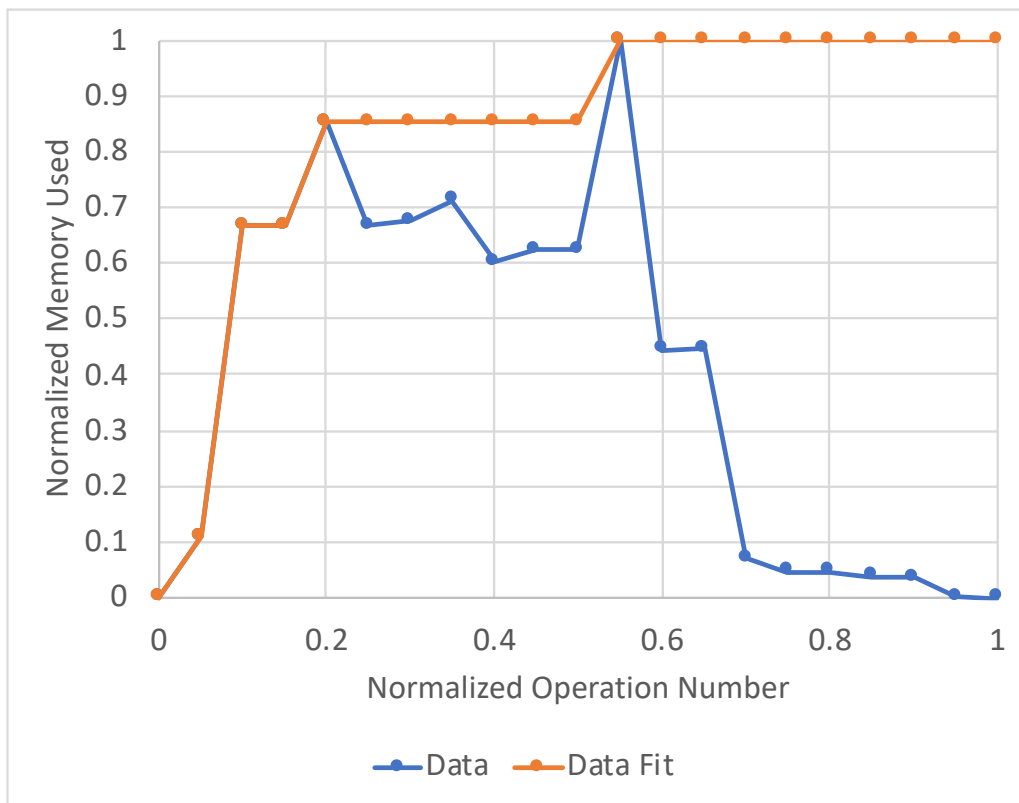
- Trace provided with malloc lab
- Allocate & free 10 blocks
- a = allocate
- f = free
- Bias toward allocate at beginning & free at end
- Blocks numbered 0–9
- Allocated: Sum of all allocated amounts
- Peak: Max so far of Allocated

Step	Command	Delta	Allocated	Peak
1	a 0 9904	9904	9904	9904
2	a 1 50084	50084	59988	59988
3	a 2 20	20	60008	60008
4	a 3 16784	16784	76792	76792
5	f 3	-16784	60008	76792
6	a 4 840	840	60848	76792
7	a 5 3244	3244	64092	76792
8	f 0	-9904	54188	76792
9	a 6 2012	2012	56200	76792
10	f 2	-20	56180	76792
11	a 7 33856	33856	90036	90036
12	f 1	-50084	39952	90036
13	a 8 136	136	40088	90036
14	f 7	-33856	6232	90036
15	f 6	-2012	4220	90036
16	a 9 20	20	4240	90036
17	f 4	-840	3400	90036
18	f 8	-136	3264	90036
19	f 5	-3244	20	90036
20	f 9	-20	0	90036



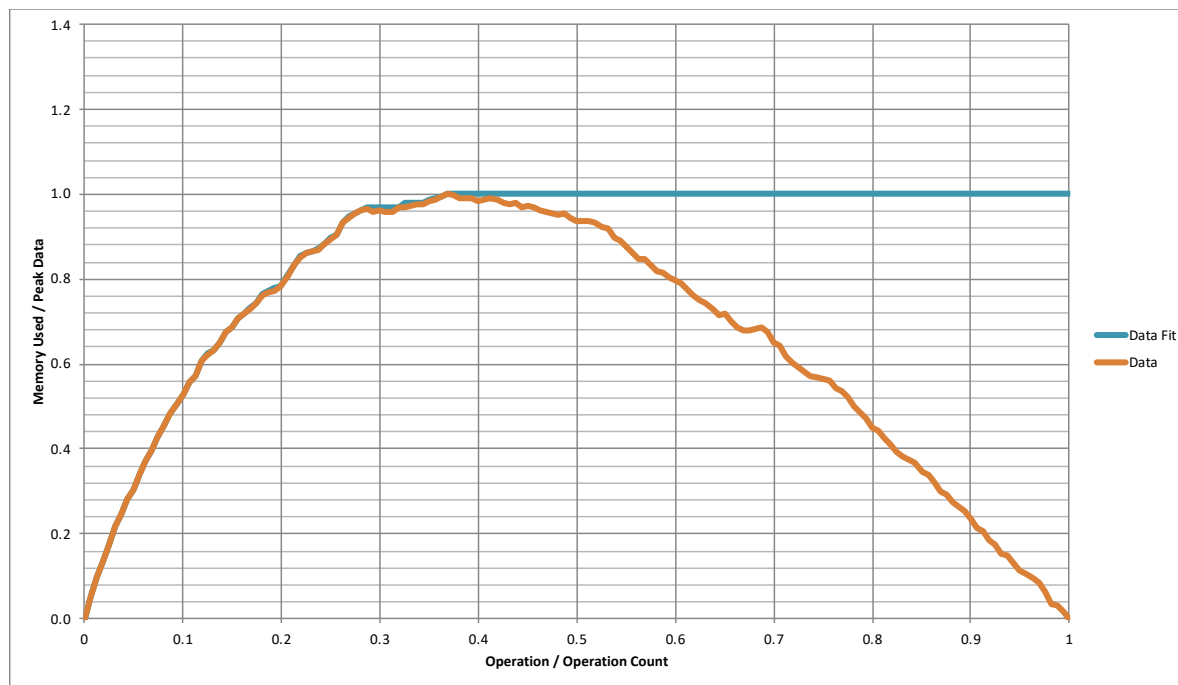
# Benchmark Visualization

Step	Command	Delta	Allocated	Peak
1	a 0 9904	9904	9904	9904
2	a 1 50084	50084	59988	59988
3	a 2 20	20	60008	60008
4	a 3 16784	16784	76792	76792
5	f 3	-16784	60008	76792
6	a 4 840	840	60848	76792
7	a 5 3244	3244	64092	76792
8	f 0	-9904	54188	76792
9	a 6 2012	2012	56200	76792
10	f 2	-20	56180	76792
11	a 7 33856	33856	90036	90036
12	f 1	-50084	39952	90036
13	a 8 136	136	40088	90036
14	f 7	-33856	6232	90036
15	f 6	-2012	4220	90036
16	a 9 20	20	4240	90036
17	f 4	-840	3400	90036
18	f 8	-136	3264	90036
19	f 5	-3244	20	90036
20	f 9	-20	0	90036



- Data line shows total allocated data ( $P_i$ )
- Data Fit line shows peak of total ( $\max_{i \leq k} P_i$ )
- Normalized in X & Y

# Full Benchmark Behavior



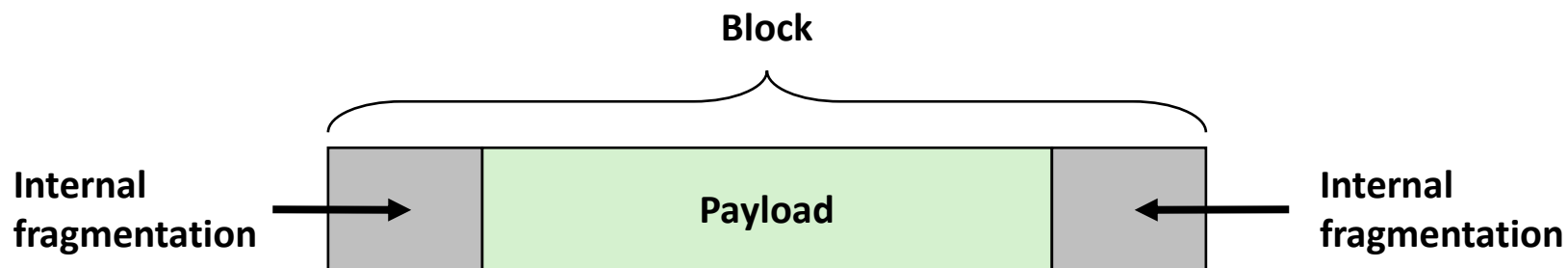
- **Given sequence of mallocs & frees (40,000 blocks)**
  - Starts with all mallocs, and shifts toward all frees
- **Manage space for all allocated blocks**
- **Metrics**
  - Data:  $P_i$
  - Data fit:  $\max_{i \leq k} P_i$

# Fragmentation

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

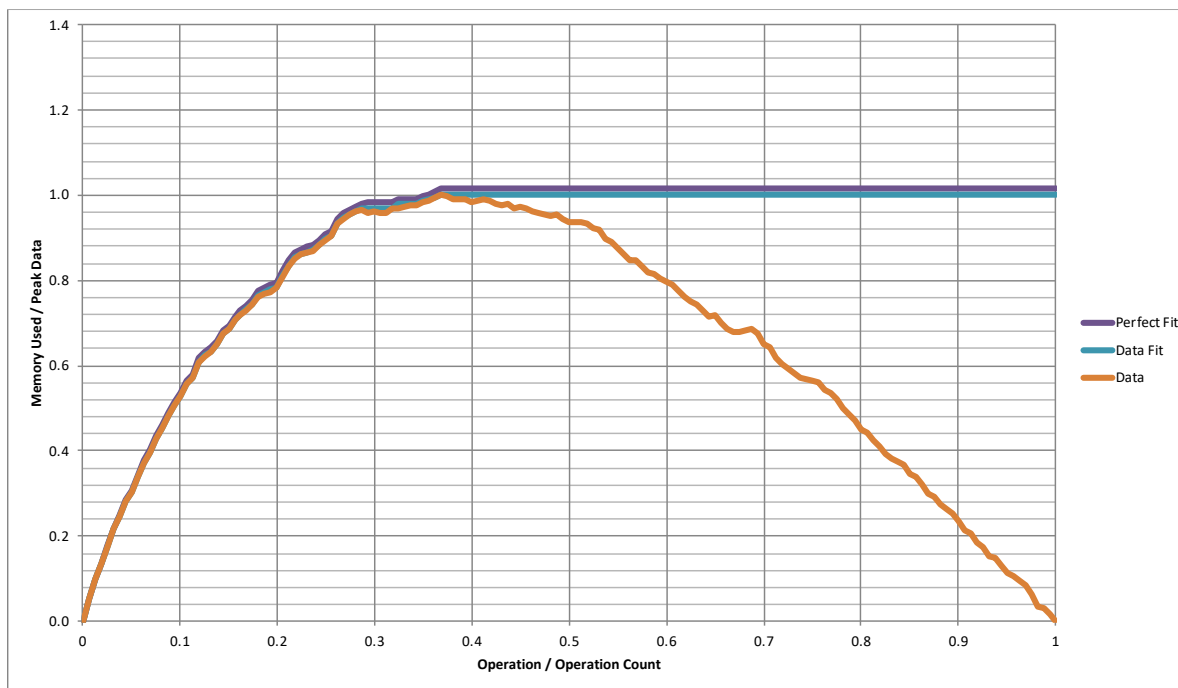
# Internal Fragmentation

- 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

# Internal Fragmentation Effect



- **Perfect Fit: Only requires space for allocated data, data structures, and unused space due to alignment constraints**
  - For this benchmark, 1.5% overhead
  - Cannot achieve in practice
    - Especially since cannot move allocated blocks

# External Fragmentation

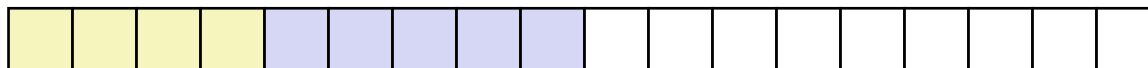
```
#define SIZ sizeof(size_t)
```

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

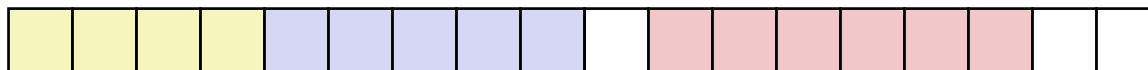
```
p1 = malloc(4*SIZ)
```



```
p2 = malloc(5*SIZ)
```



```
p3 = malloc(6*SIZ)
```



```
free(p2)
```

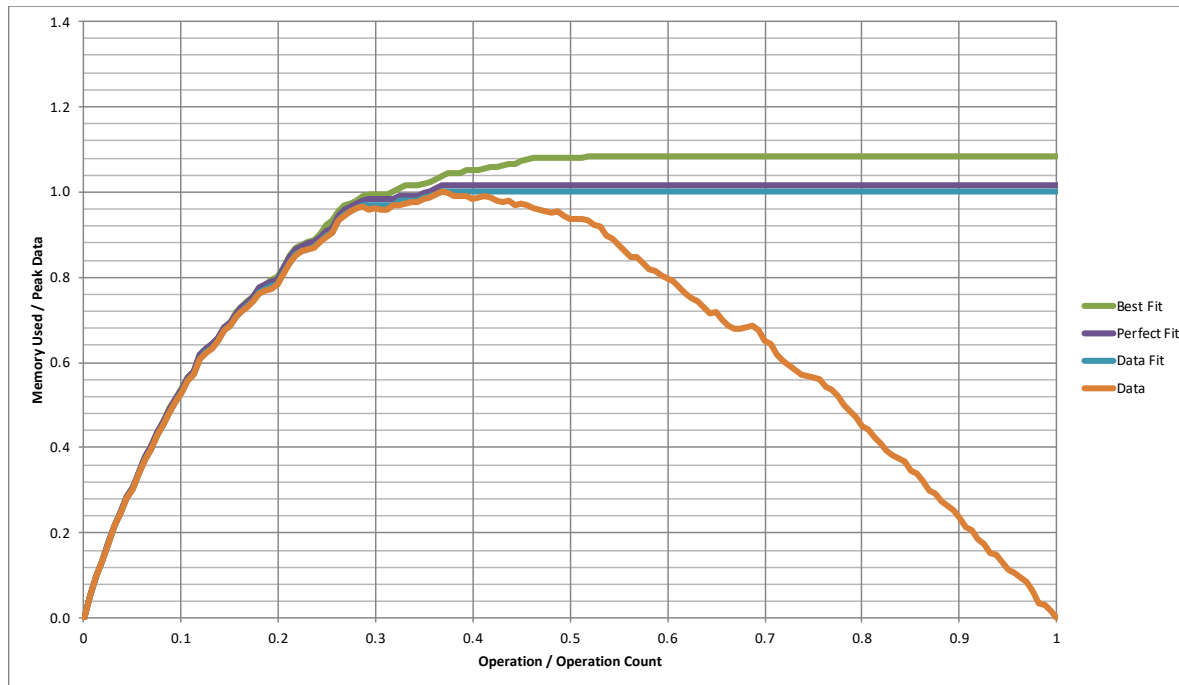


```
p4 = malloc(7*SIZ)
```

*Yikes! (what would happen now?)*

- Amount of external fragmentation depends on the pattern of future requests
  - Thus, difficult to measure

# External Fragmentation Effect



## ■ Best Fit: One allocation strategy

- (To be discussed later)
- Total overhead = 8.3% on this benchmark

# Implementation Issues

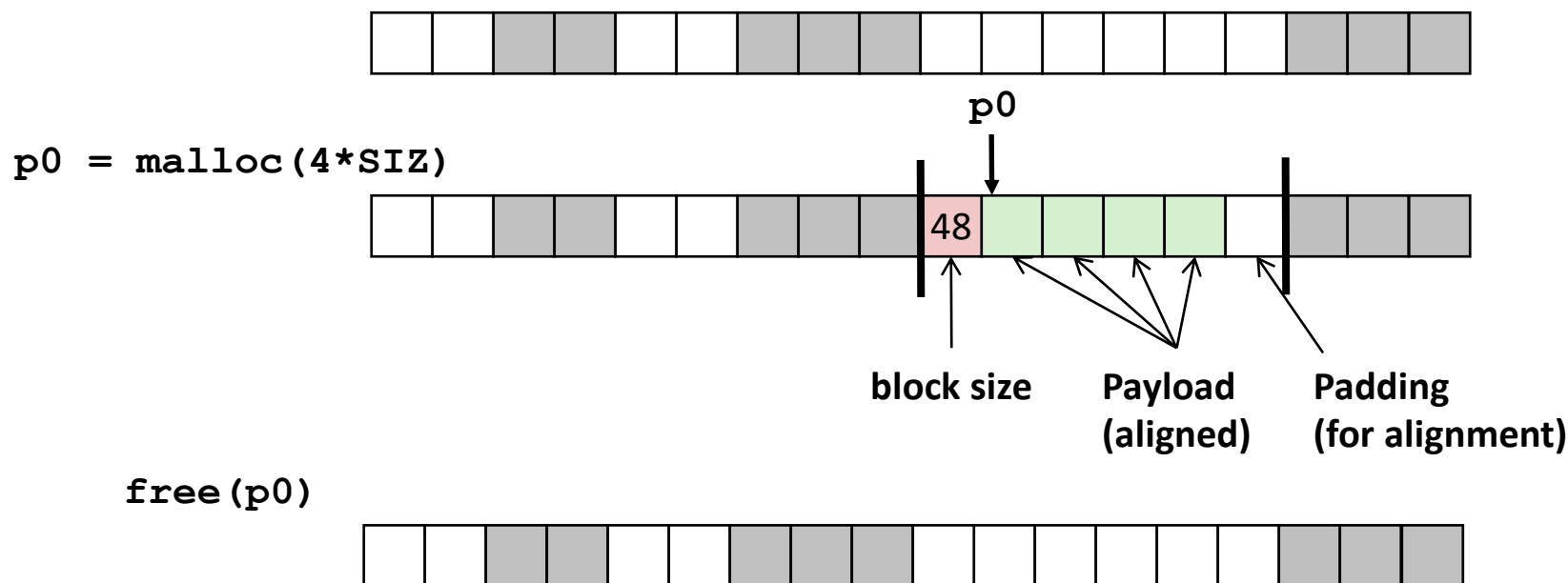
- **How do we know how much memory to free given just a pointer?**
- **How do we keep track of the free blocks?**
- **What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?**
- **How do we pick a block to use for allocation -- many might fit?**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



# Knowing How Much to Free

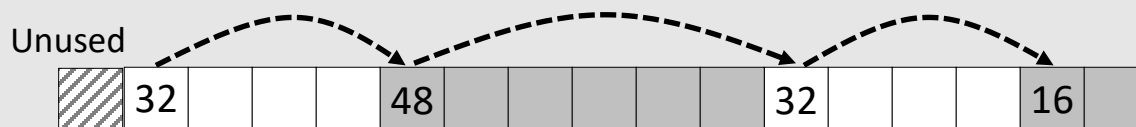
## ■ Standard method

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



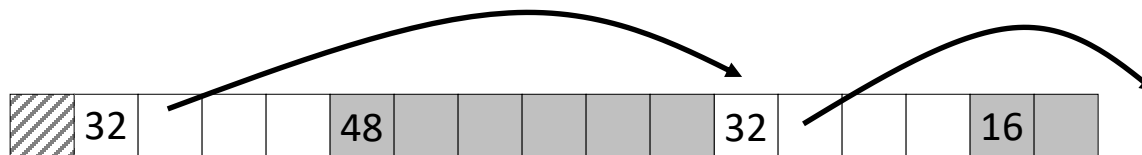
# Keeping Track of Free Blocks

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



Need to tag each block as allocated/free

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



Need space for 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

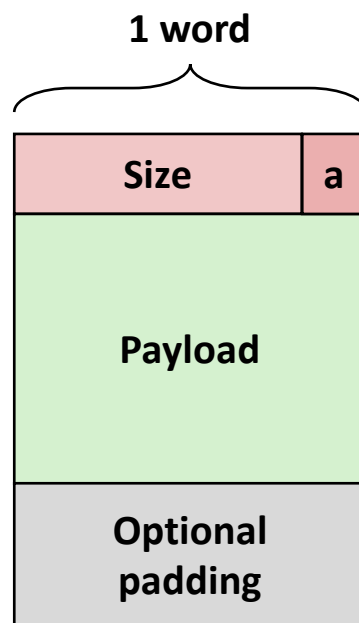
# Today

- Basic concepts
- **Implicit free lists**

# Method 1: Implicit Free List

- **For each block we need both size and allocation status**
  - Could store this information in two words: wasteful!
- **Standard trick**
  - When blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as an allocated/free flag
  - When reading the Size word, must mask out this bit

*Format of  
allocated and  
free blocks*



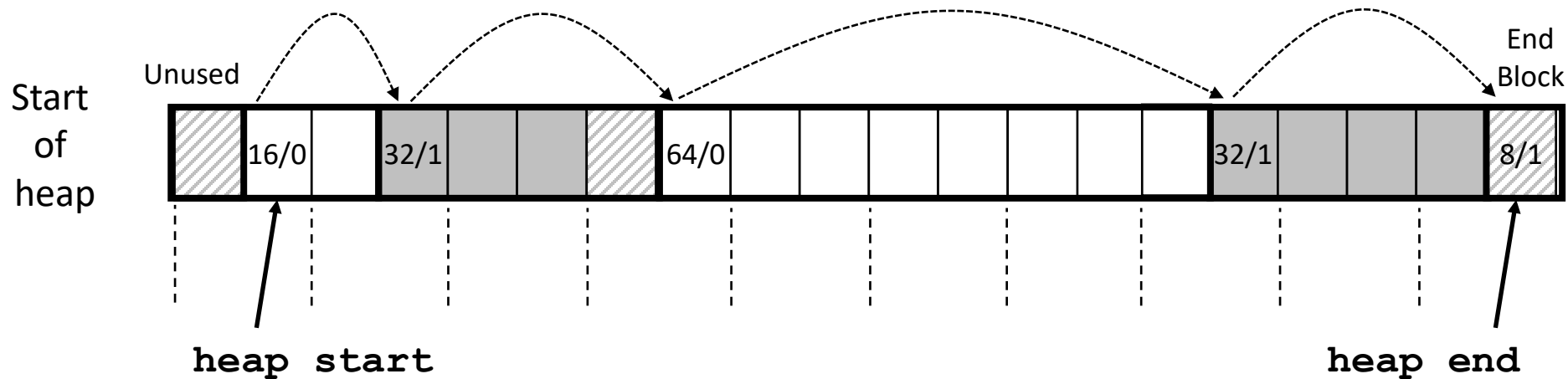
**a = 1: Allocated block**

**a = 0: Free block**

**Size: total block size**

**Payload: application data  
(allocated blocks only)**

# Detailed Implicit Free List Example



┆ Double-word  
┆ aligned

**Allocated blocks:** shaded

**Free blocks:** unshaded

**Headers:** labeled with “size in words/allocated bit”

Headers are at non-aligned positions

➔ Payloads are aligned

# Implicit List: Data Structures



## ■ Block declaration

```
typedef uint64_t word_t;
```

```
typedef struct block
{
    word_t header;
    unsigned char payload[0];           // Zero length array
} block_t;
```

## ■ Getting payload from block pointer // block\_t \*block

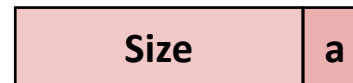
```
return (void *) (block->payload);
```

## ■ Getting header from payload // bp points to a payload

```
return (block_t *) ((unsigned char *) bp
    - offsetof(block_t, payload));
```

**C function `offsetof(struct, member)` returns offset of member within struct**

# Implicit List: Header access



- Getting allocated bit from header

```
return header & 0x1;
```

- Getting size from header

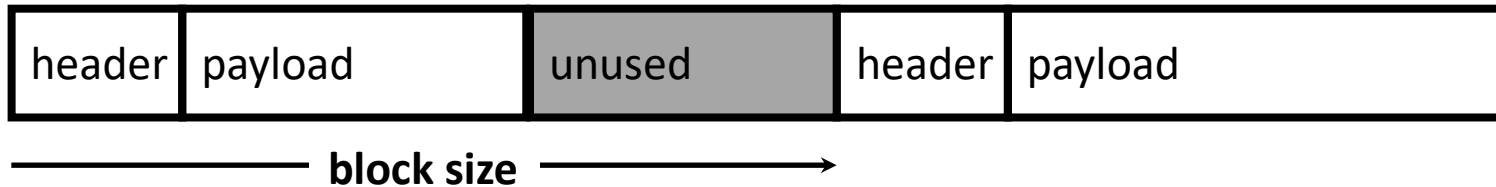
```
return header & ~0xfL;
```

- Initializing header

```
//block_t *block
```

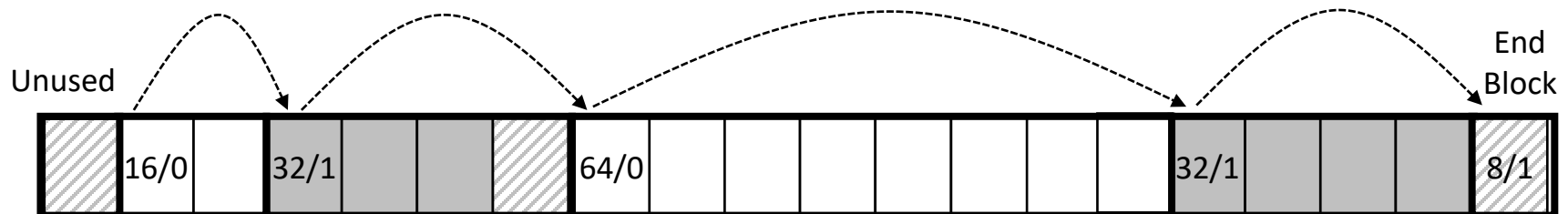
```
block->header = size | alloc;
```

# Implicit List: Traversing list



## ■ Find next block

```
static block_t *find_next(block_t *block)
{
    return (block_t *) ((unsigned char *) block
        + get_size(block));
}
```



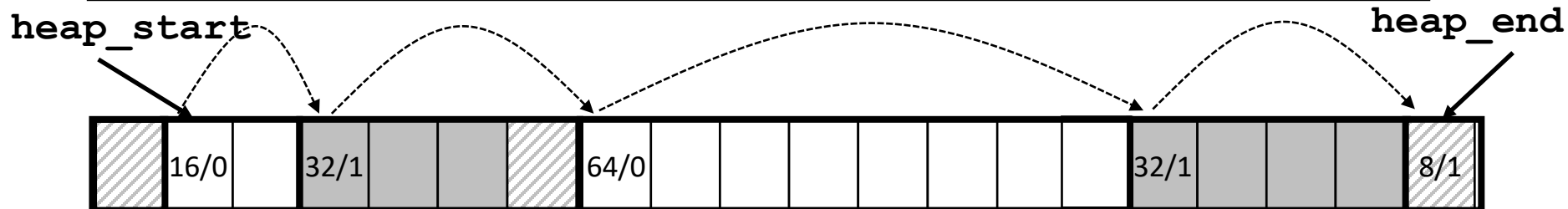


# Implicit List: Finding a Free Block

## ■ *First fit:*

- Search list from beginning, choose *first* free block that fits:
- Finding space for **asize** bytes (including header):

```
static block_t *find_fit(size_t asize)
{
    block_t *block;
    for (block = heap_start; block != heap_end;
         block = find_next(block)) {
        {
            if (!(get_alloc(block))
                && (asize <= get_size(block)))
                return block;
        }
    }
    return NULL; // No fit found
}
```



# 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)
- In practice it can cause “splinters” at beginning of list

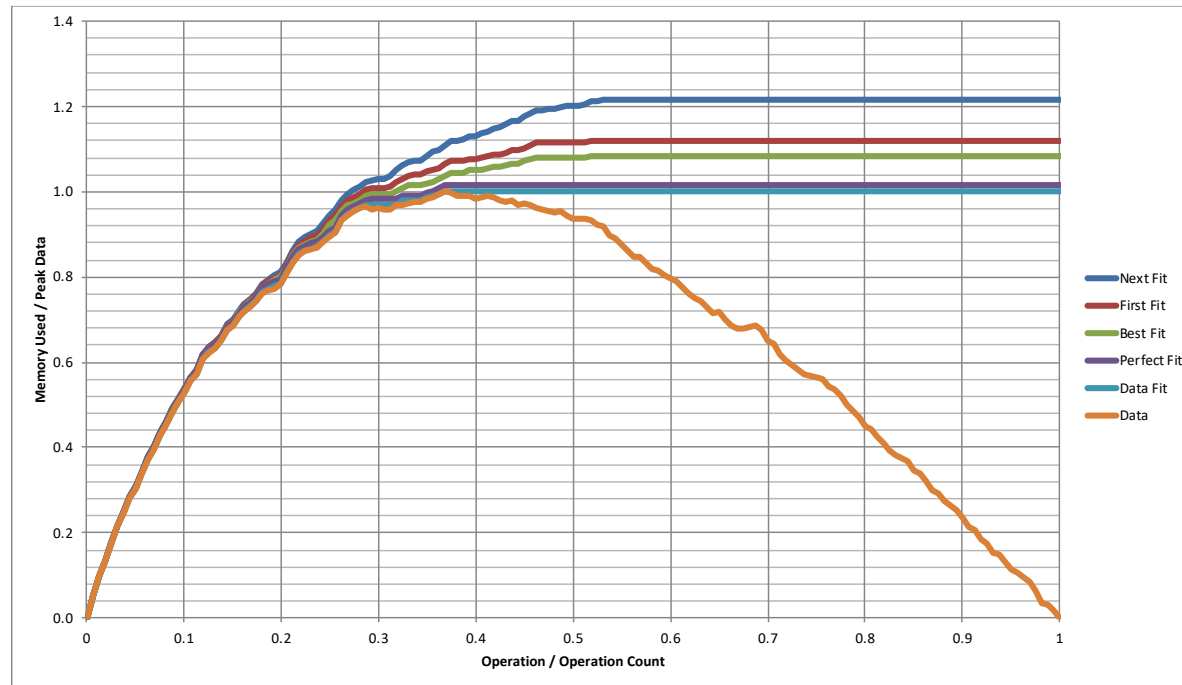
## ■ *Next fit:*

- 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
- Still a greedy algorithm. No guarantee of optimality

# Comparing Strategies



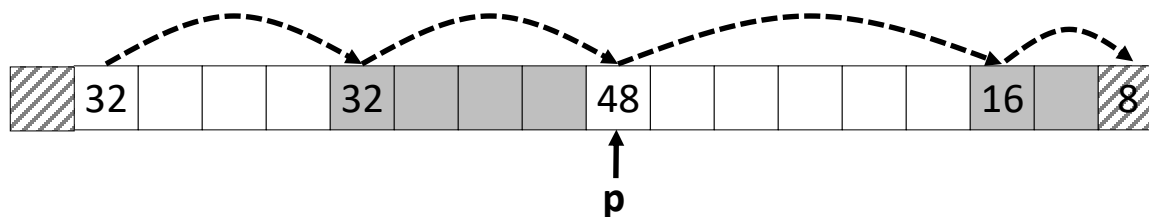
## ■ Total Overheads (for this benchmark)

- Perfect Fit: 1.6%
- Best Fit: 8.3%
- First Fit: 11.9%
- Next Fit: 21.6%

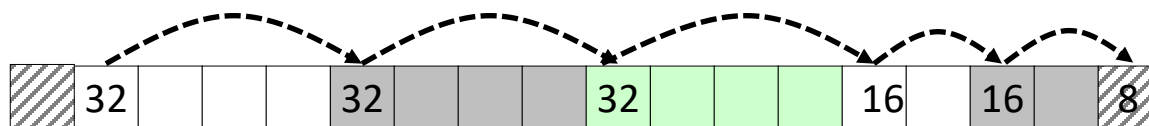
# Implicit List: Allocating in Free Block

## ■ Allocating in a free block: *splitting*

- Since allocated space might be smaller than free space, we might want to split the block

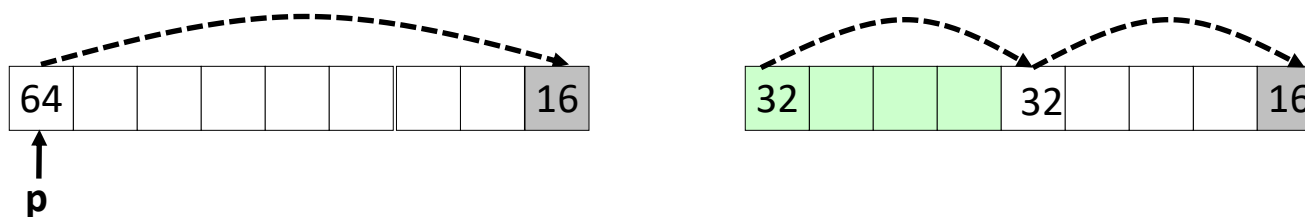


`split_block(p, 32)`



# Implicit List: Splitting Free Block

```
split_block(p, 32)
```



```
// Warning: This code is incomplete
```

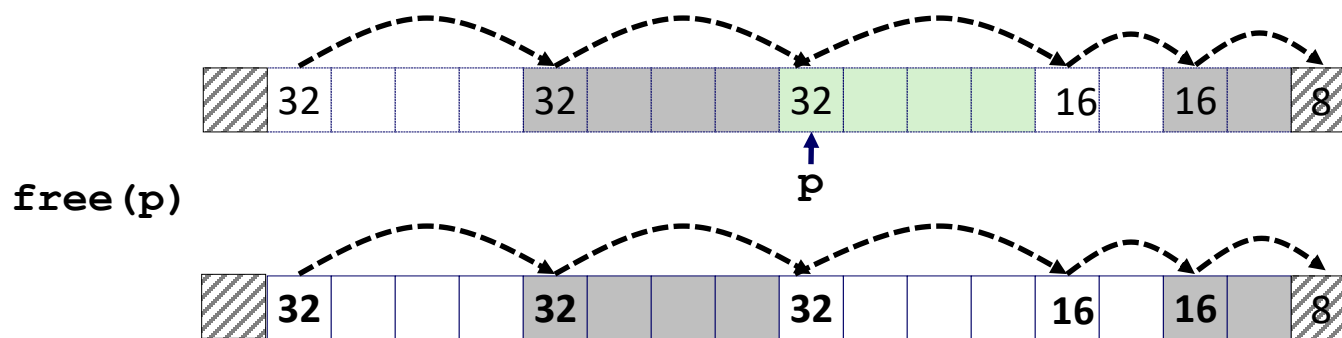
```
static void split_block(block_t *block, size_t asize) {
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
    }
}
```

# Implicit List: Freeing a Block

## ■ Simplest implementation:

- Need only clear the “allocated” flag
- But can lead to “false fragmentation”



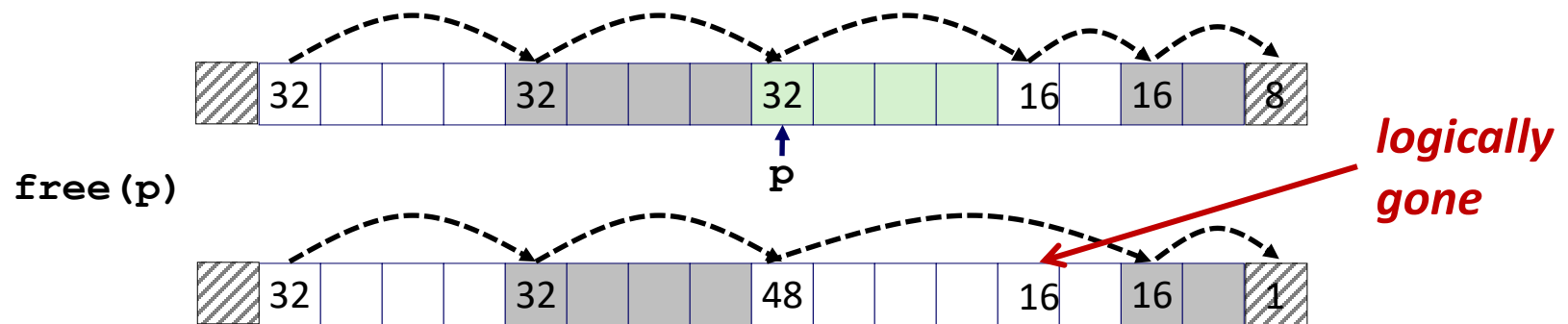
`malloc(5*SIZ)`

***Yikes!***

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

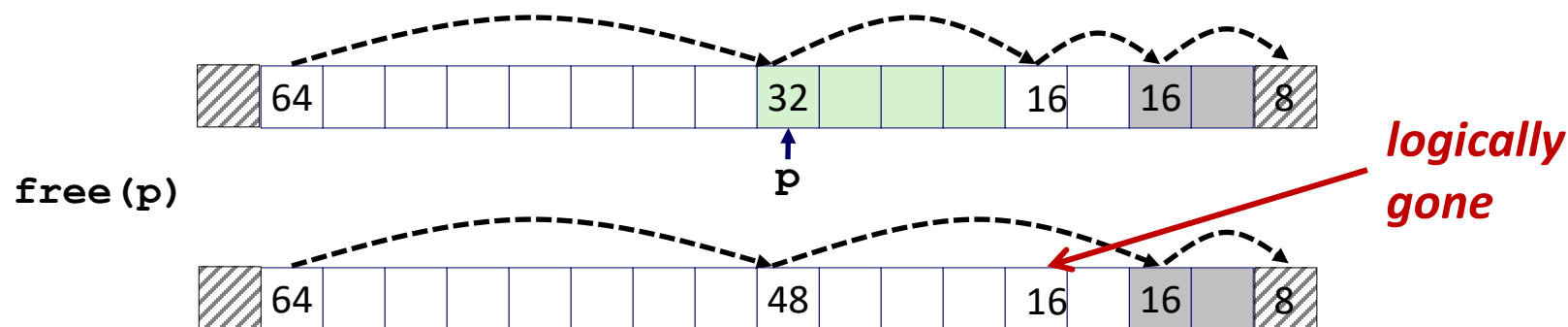
# Implicit List: Coalescing

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



# Implicit List: Coalescing

- Join (*coalesce*) with next block, if it is free
  - Coalescing with next block



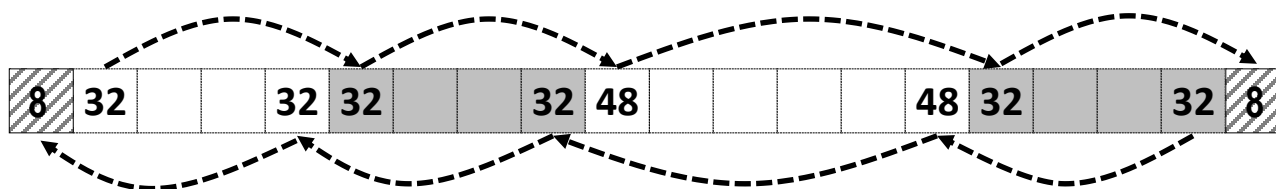
- How do we coalesce with *previous* block?
  - How do we know where it starts?
  - How can we determine whether its allocated?



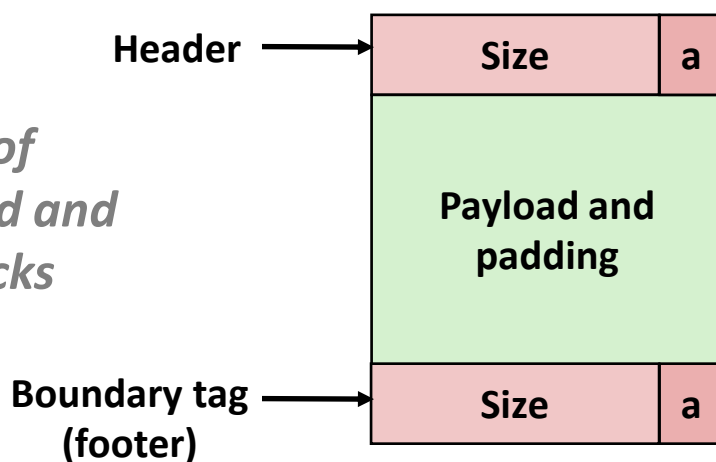
# Implicit List: Bidirectional Coalescing

## ■ *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**

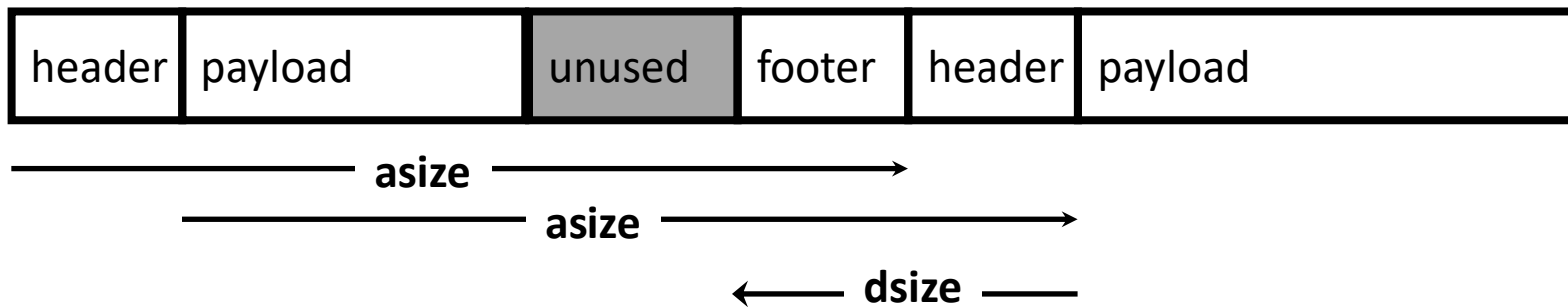
**Payload: Application data  
(allocated blocks only)**

# Quiz Time!

Check out:

<https://canvas.cmu.edu/courses/17808>

# Implementation with Footers



## ■ Locating footer of current block

```
const size_t dsize = 2*sizeof(word_t);

static word_t *header_to_footer(block_t *block)
{
    size_t asize = get_size(block);
    return (word_t *) (block->payload + asize - dsize);
}
```

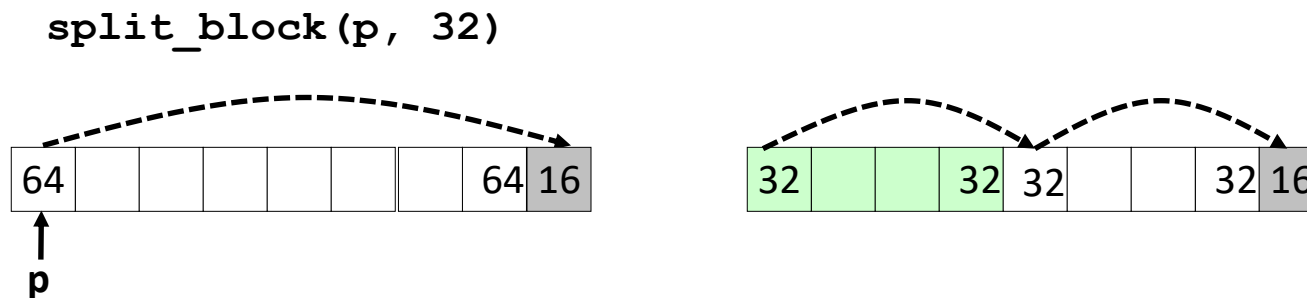
# Implementation with Footers



## ■ Locating footer of previous block

```
static word_t *find_prev_footer(block_t *block)
{
    return &(block->header) - 1;
}
```

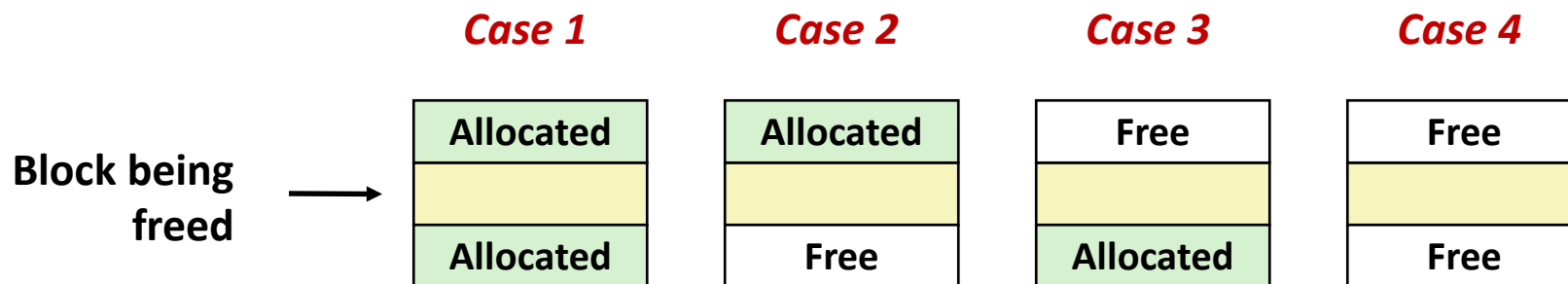
# Splitting Free Block: Full Version



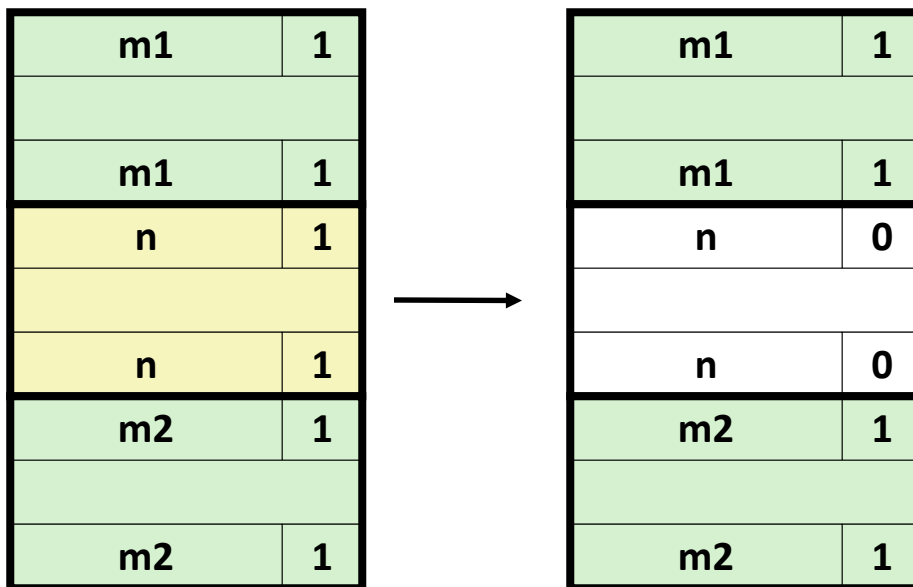
```
static void split_block(block_t *block, size_t asize) {
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        write_footer(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
        write_footer(block_next, block_size - asize, false);
    }
}
```

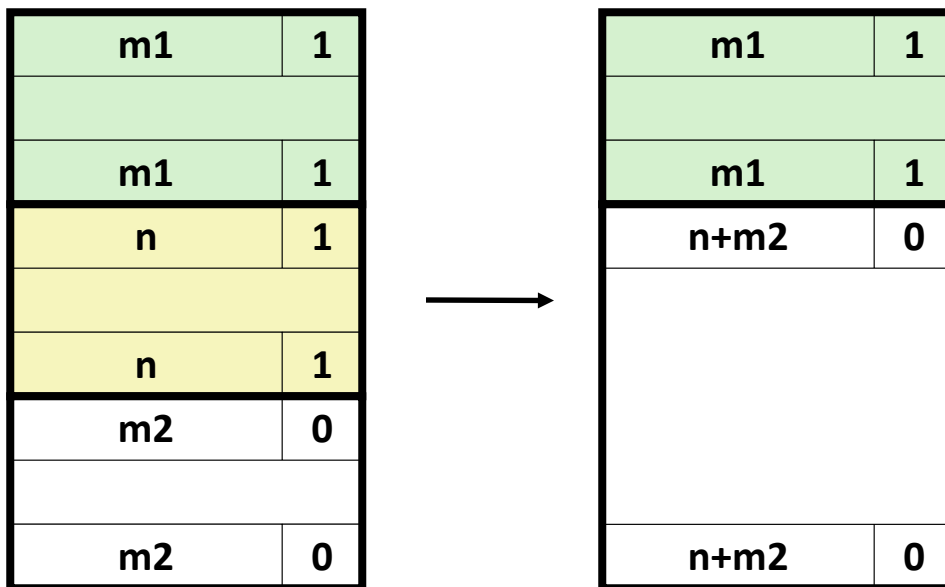
# Constant Time Coalescing



# Constant Time Coalescing (Case 1)

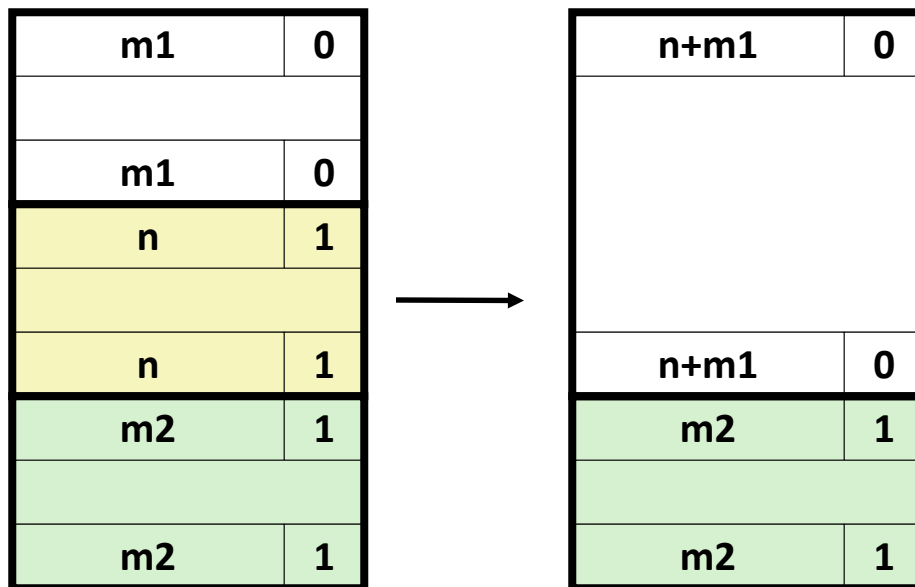


# Constant Time Coalescing (Case 2)

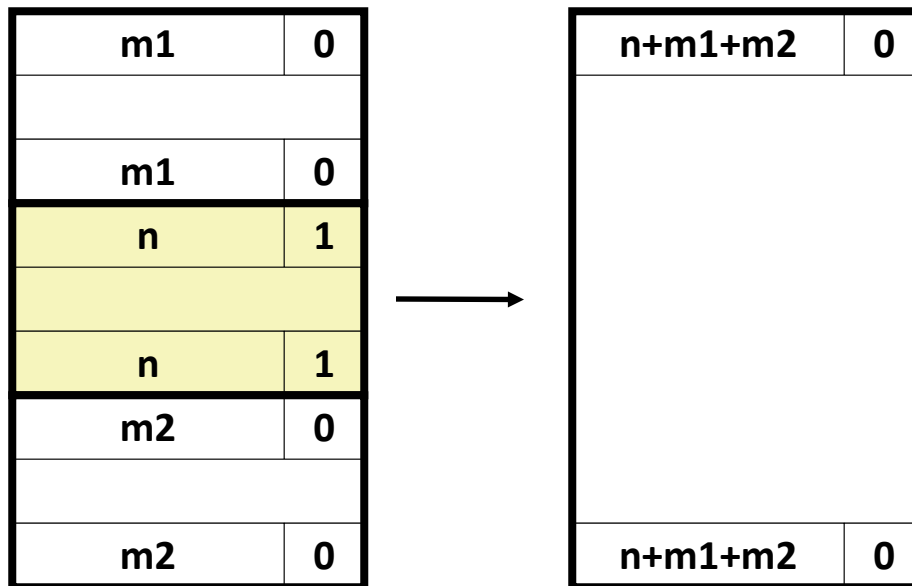




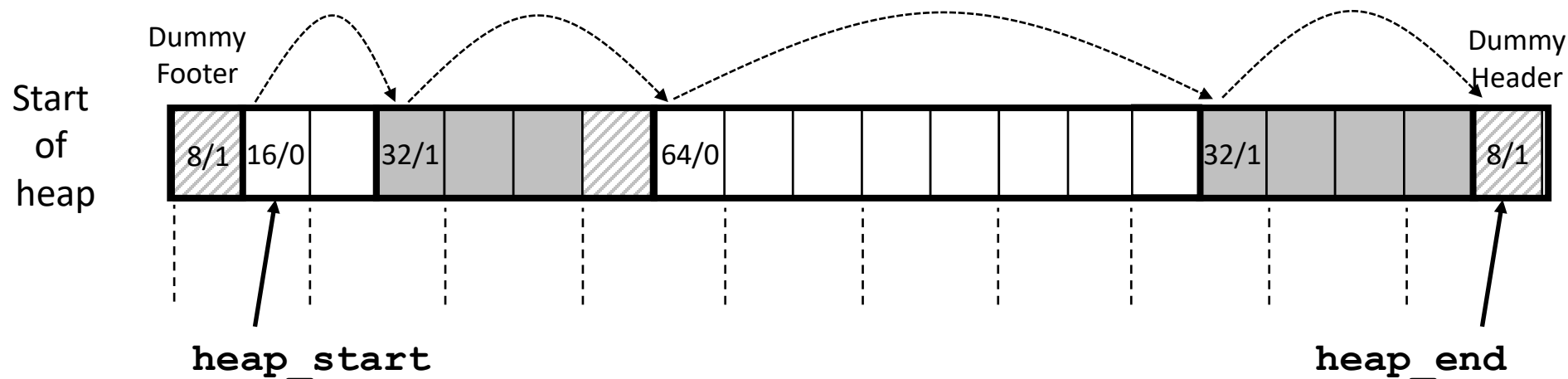
# Constant Time Coalescing (Case 3)



# Constant Time Coalescing (Case 4)



# Heap Structure



## ■ Dummy footer before first header

- Marked as allocated
- Prevents accidental coalescing when freeing first block

## ■ Dummy header after last footer

- Prevents accidental coalescing when freeing final block

# Top-Level Malloc Code

```
const size_t dsize = 2*sizeof(word_t);

void *mm_malloc(size_t size)
{
    size_t asize = round_up(size + dsize, dsize);

    block_t *block = find_fit(asize);

    if (block == NULL)
        return NULL;

    size_t block_size = get_size(block);
    write_header(block, block_size, true);
    write_footer(block, block_size, true);

    split_block(block, asize);

    return header_to_payload(block);
}
```

$$\begin{aligned} \text{round\_up}(n, m) \\ &= \\ m * ((n+m-1) / m) \end{aligned}$$

# Top-Level Free Code

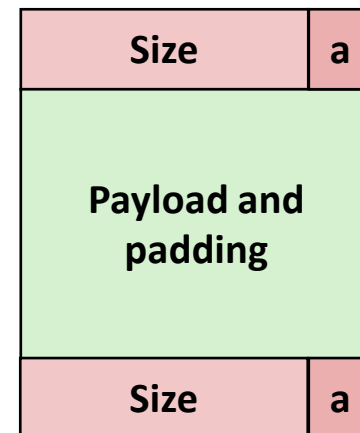
```
void mm_free(void *bp)
{
    block_t *block = payload_to_header(bp);
    size_t size = get_size(block);

    write_header(block, size, false);
    write_footer(block, size, false);

    coalesce_block(block);
}
```

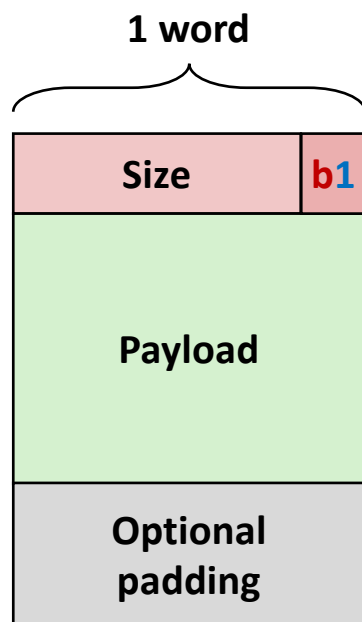
# Disadvantages of Boundary Tags

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



# No Boundary Tag for Allocated Blocks

- Boundary tag needed only for free blocks
- When sizes are multiples of 16, have 4 spare bits

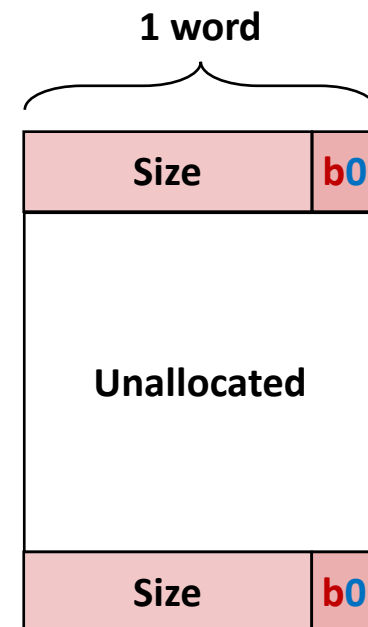


Allocated  
Block

**a = 1: Allocated block**  
**a = 0: Free block**  
**b = 1: Previous block is allocated**  
**b = 0: Previous block is free**

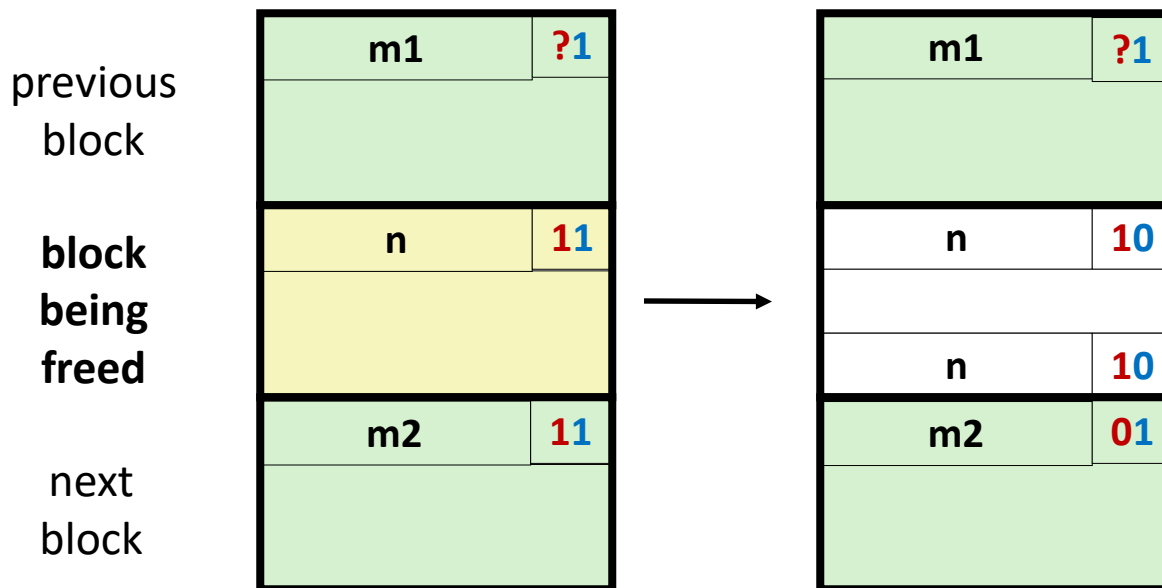
Size: block size

Payload: application data



Free  
Block

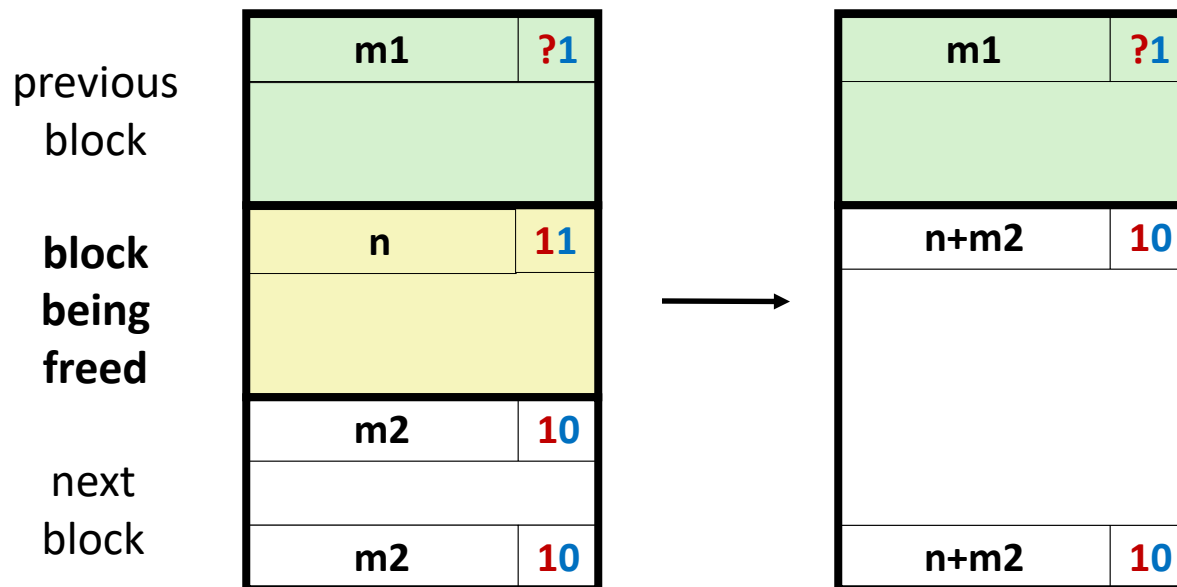
# No Boundary Tag for Allocated Blocks (Case 1)



**Header:** Use 2 bits (address bits always zero due to alignment):  
**(previous block allocated)** $\ll 1$  | **(current block allocated)**

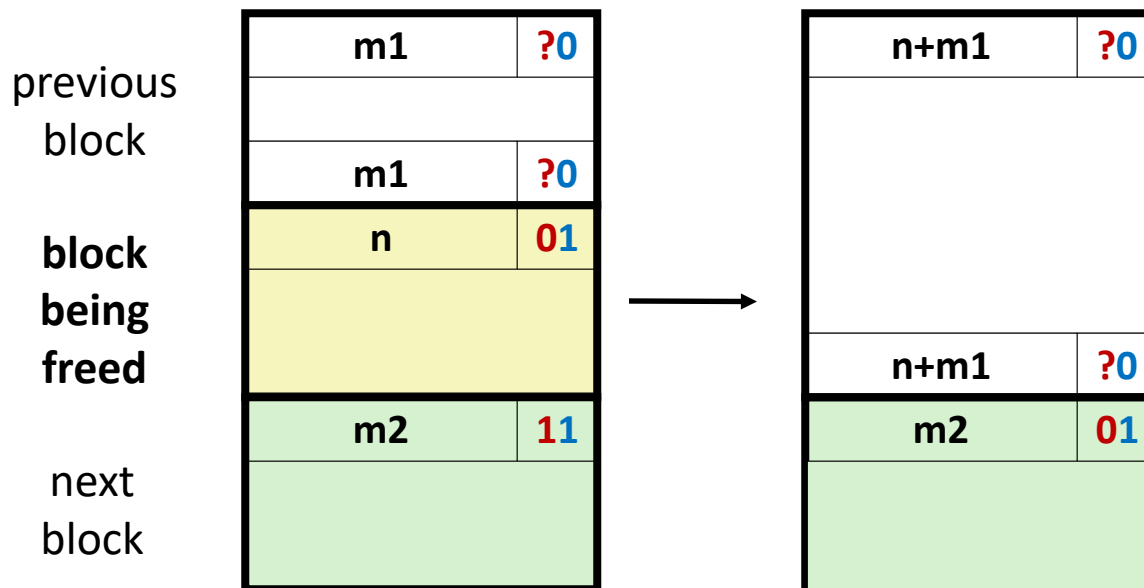


# No Boundary Tag for Allocated Blocks (Case 2)



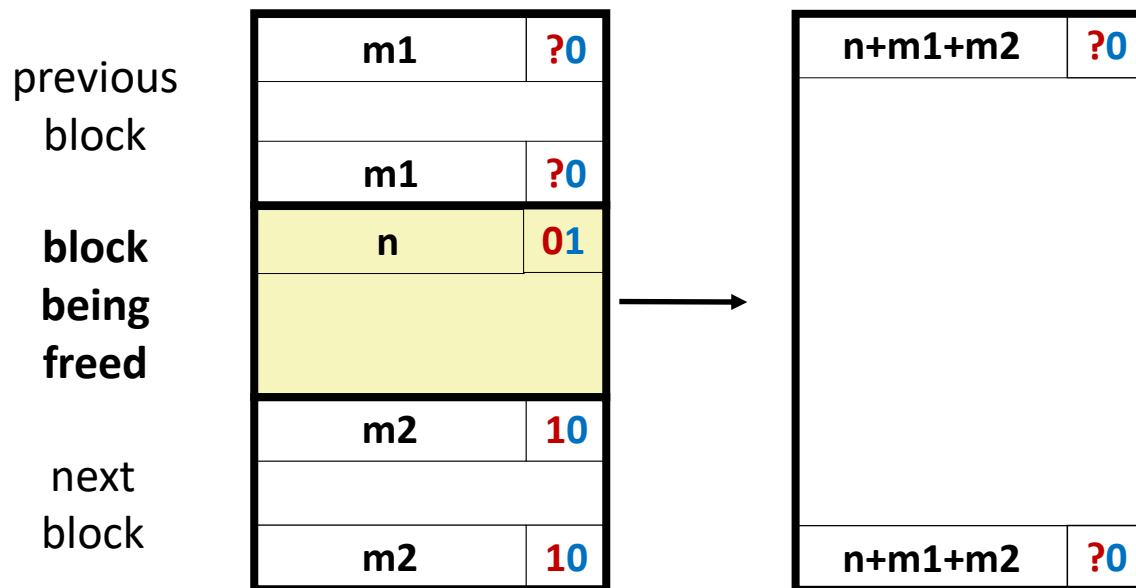
Header: Use 2 bits (address bits always zero due to alignment):  
 (previous block allocated) << 1 | (current block allocated)

# No Boundary Tag for Allocated Blocks (Case 3)



Header: Use 2 bits (address bits always zero due to alignment):  
**(previous block allocated)**<<1 | **(current block allocated)**

# No Boundary Tag for Allocated Blocks (Case 4)



Header: Use 2 bits (address bits always zero due to alignment):  
 (previous block allocated) << 1 | (current block allocated)

# Summary of Key Allocator Policies

## ■ Placement policy:

- First-fit, next-fit, best-fit, etc.
- Trades off lower throughput for less fragmentation
- *Interesting observation*: segregated free lists (next lecture) approximate a best fit placement policy without having to search 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*: try to improve performance of **free** by deferring coalescing until needed.

# Implicit Lists: Summary

- **Implementation: very simple**
- **Allocate cost:**
  - linear time worst case
- **Free cost:**
  - constant time worst case
  - even with coalescing
- **Memory Overhead**
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- **Not used in practice for `malloc/free` because of linear-time allocation**
  - used in many special purpose applications
- **However, the concepts of splitting and boundary tag coalescing are general to *all* allocators**