Computer Architecture and Operating Systems
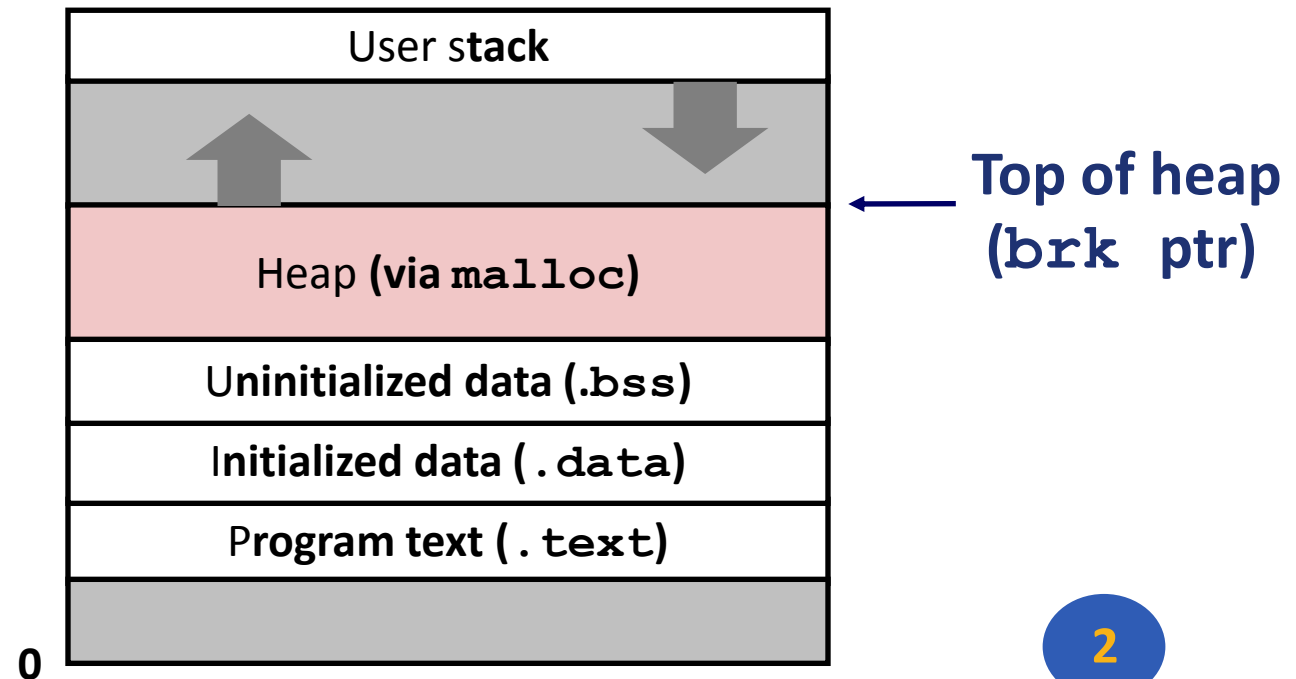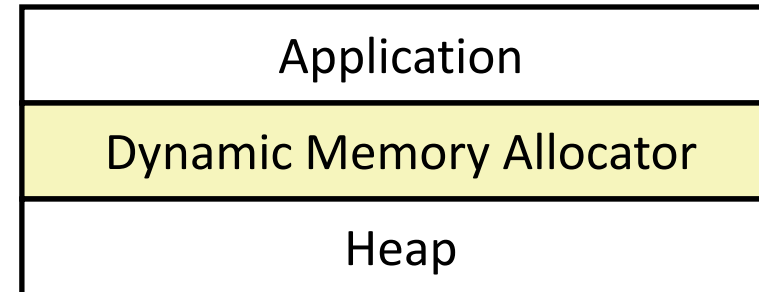# Lecture 5: Dynamic Memory Allocation

Andrei Tatarnikov

atatarnikov@hse.ru
@andrewt0301

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM 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*.

Application
Dynamic Memory Allocator
Heap

User s**tack**
Heap **(via `malloc`)**
U**ninitialized data (.bss)**
I**nitialized data (`.data`)**
P**rogram text (`.text`)**

← **Top of heap (`brk ptr`)**

0

2

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, ML, and Lisp

- Will discuss simple explicit memory allocation today

3

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 an 8-byte (x86) or 16-byte (x86-64) 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:` 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

```c
#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);
}
```

Allocation Example
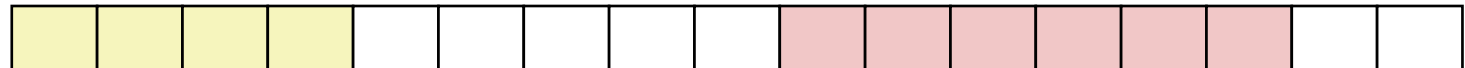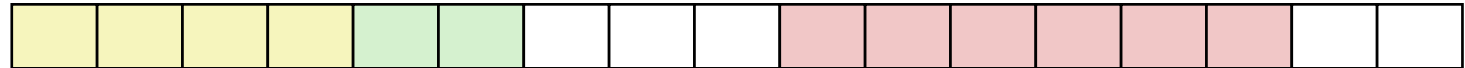

p1 = malloc(4)

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**
  - 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 (x86) or 16-byte (x86-64) alignment 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: Throughput

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

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

- *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*
  - $U_k = (max_{i<=k} P_i) / H_k$

9

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

Block

Internal fragmentation → | **Payload** | ← **Internal fragmentation**
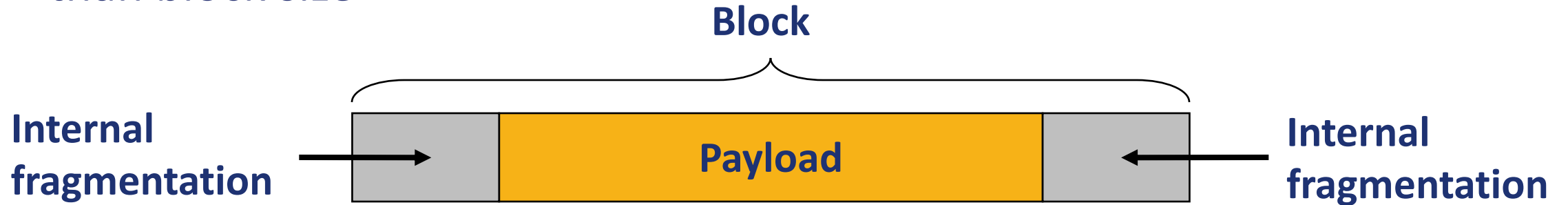
- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions
    (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of *previous* requests
  - Thus, easy to measure

External Fragmentation

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

`p1 = malloc(4)`
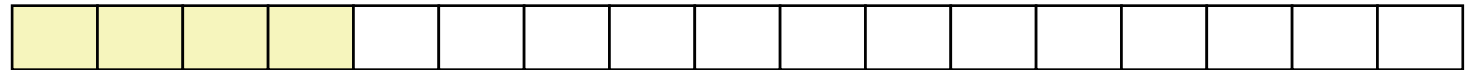
`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

`p4 = malloc(6)`   *Oops! (what would happen now?)*

- Depends on the pattern of future requests
 - Thus, difficult to measure

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?

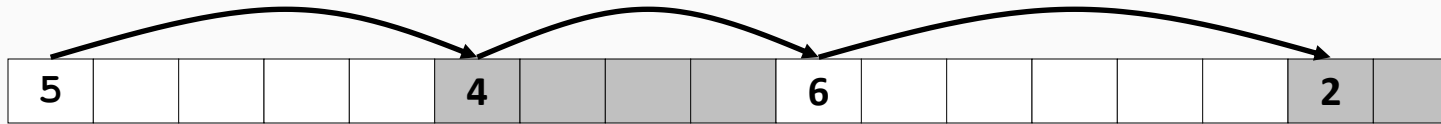- How do we reinsert freed block?

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

p0

p0 = malloc(4)

5

block size   payload

free(p0)
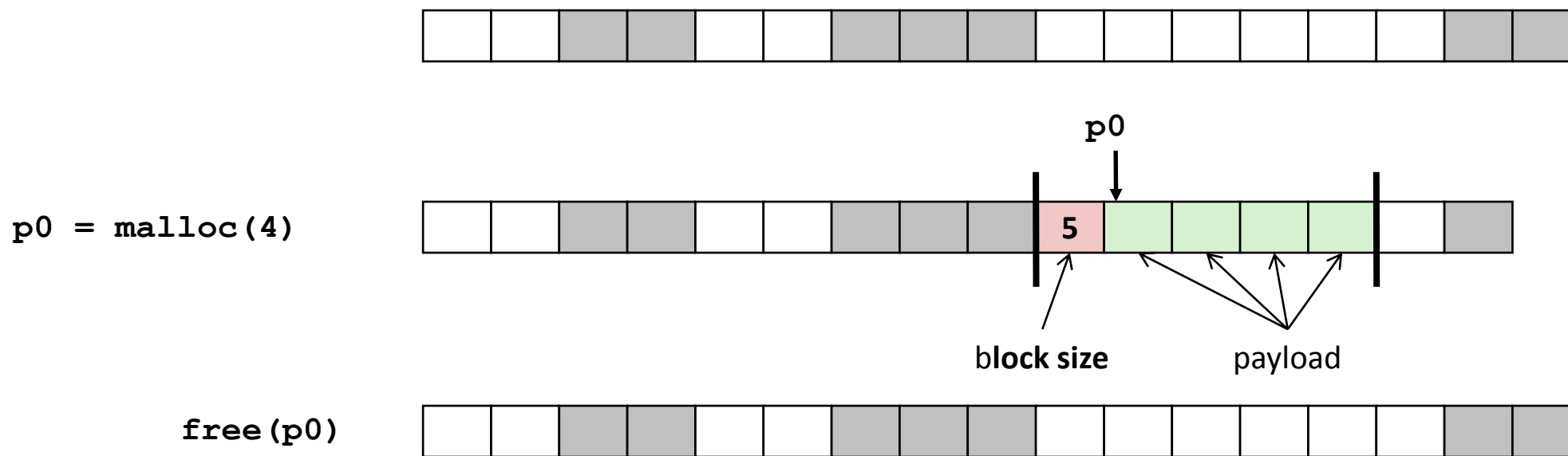
Keeping Track of Free Blocks

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

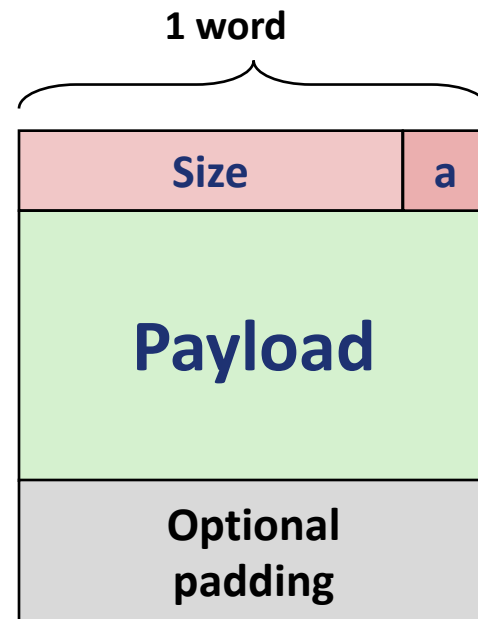- Method 2: *Explicit list* among the free blocks using pointers

- 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

15

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
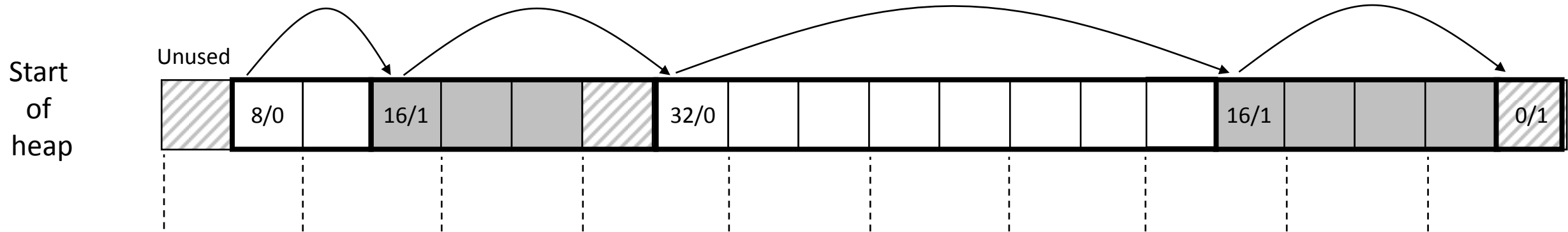
Format of allocated and free blocks

```
           1 word
        ┌──────────┐
  ┌─────────────┬───┐
  │    Size     │ a │
  ├─────────────┴───┤
  │                 │
  │    Payload      │
  │                 │
  ├─────────────────┤
  │    Optional     │
  │    padding      │
  └─────────────────┘
```

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

Size: block size

Payload: application data
(allocated blocks only)

Detailed Implicit Free List Example

Start of heap

Unused

| 8/0 | 16/1 | 32/0 | 16/1 | 0/1 |

Double-word aligned

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

Implicit List: Finding a Free Block

- **First fit:**
    - **Search list from beginning, choose *first* free block that fits:**

```
p = start;
while ((p < end) &&          \\ not passed end
       ((*p & 1) ||          \\ already allocated
        (*p  <= len)))        \\ too small
   p = p + (*p & -2);        \\ goto next block (word addressed)
```

    - **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**
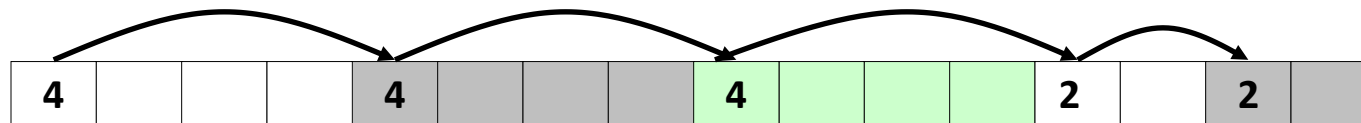
- **Best fit:**
    - **Search the list, choose the *best* free block: fits, with fewest bytes left over**
    - **Keeps fragments small—usually improves memory utilization**
    - **Will typically run slower than first fit**

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
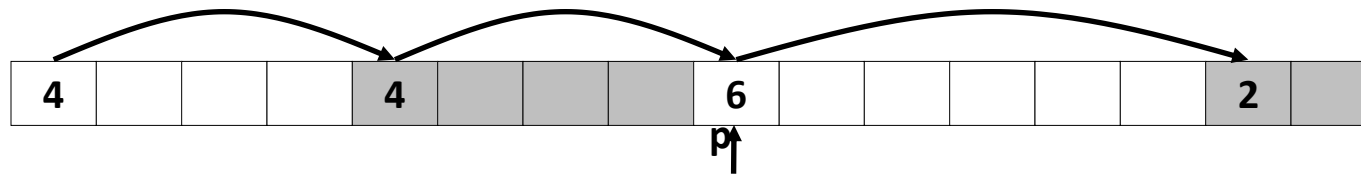
`addblock(p, 4)`


```
void addblock(ptr p, int len) {
   int newsize = ((len + 1) >> 1) << 1;   // round up to even
   int oldsize = *p & -2;                  // mask out low bit
   *p = newsize | 1;                       // set new length
   if (newsize < oldsize)
     *(p+newsize) = oldsize - newsize;     // set length in remaining
}                                          //    part of block
```
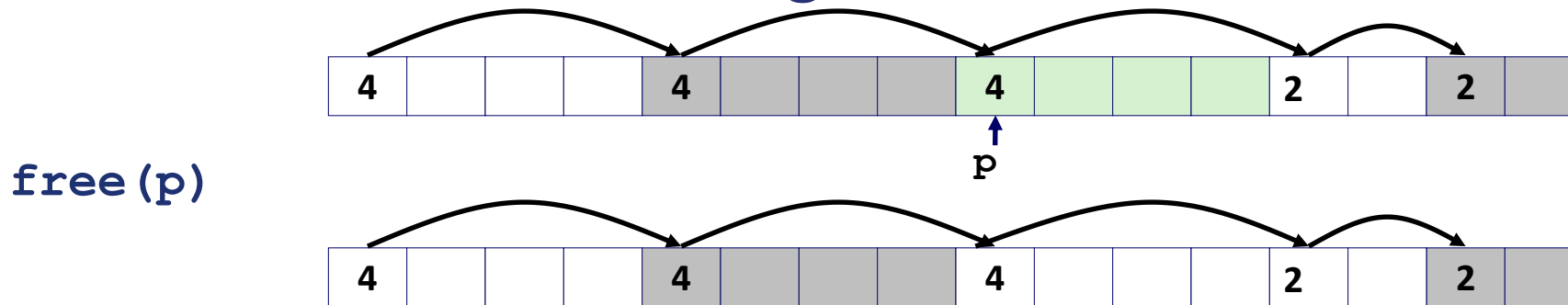
Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag

    ```
    void free_block(ptr p) { *p = *p & -2 }
    ```

  - But can lead to "false fragmentation"
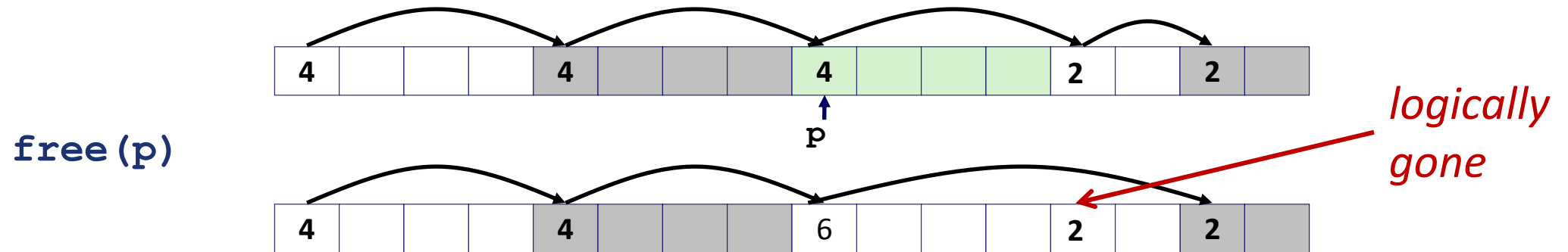
| 4 | | | | 4 | | | | 4 | | | | 2 | | 2 | |

free(p)

p

| 4 | | | | 4 | | | | 4 | | | | 2 | | 2 | |

malloc(5) *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
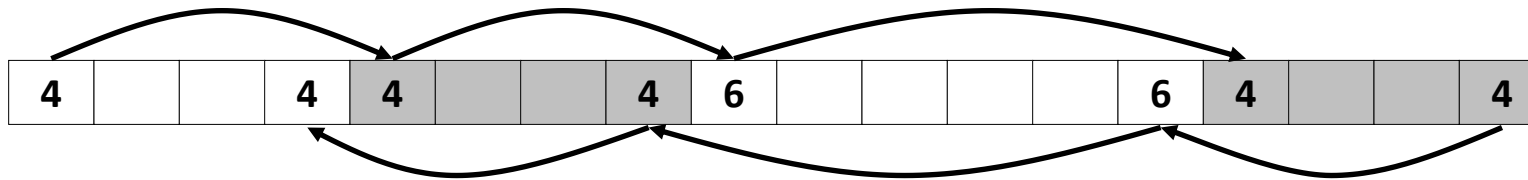


```
void free_block(ptr p) {
    *p = *p & -2;            // clear allocated flag
    next = p + *p;           // find next block
    if ((*next & 1) == 0)
        *p = *p + *next;     // add to this block if
}                            //    not allocated
```
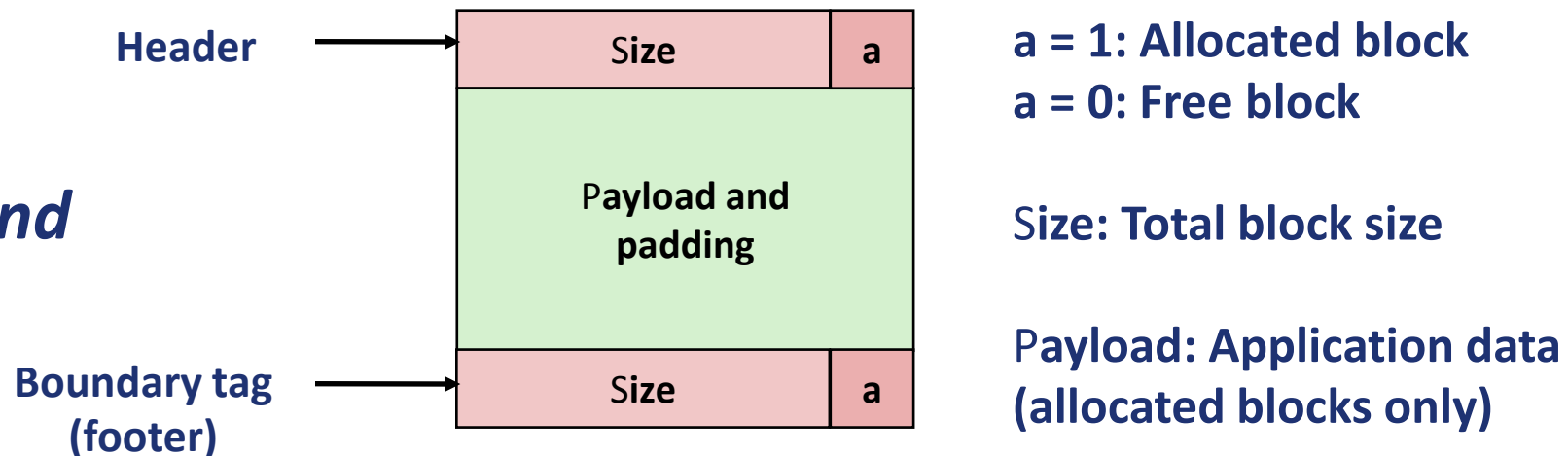
  - But how do we coalesce with *previous* block?

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

Header

Boundary tag (footer)

Size	a
Payload and padding	
Size	a

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

Size: Total block size

Payload: Application data (allocated blocks only)

Constant Time Coalescing

Block being freed →

Case 1	*Case 2*	*Case 3*	*Case 4*
Allocated	Allocated	Free	Free
Allocated	Free	Allocated	Free

m1	1
m1	1
n	1
n	1
m2	1
m2	1

→
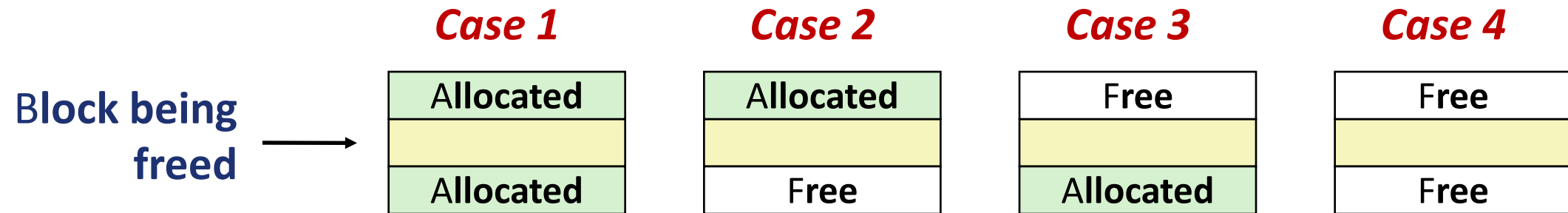
m1	1
m1	1
n	0
n	0
m2	1
m2	1
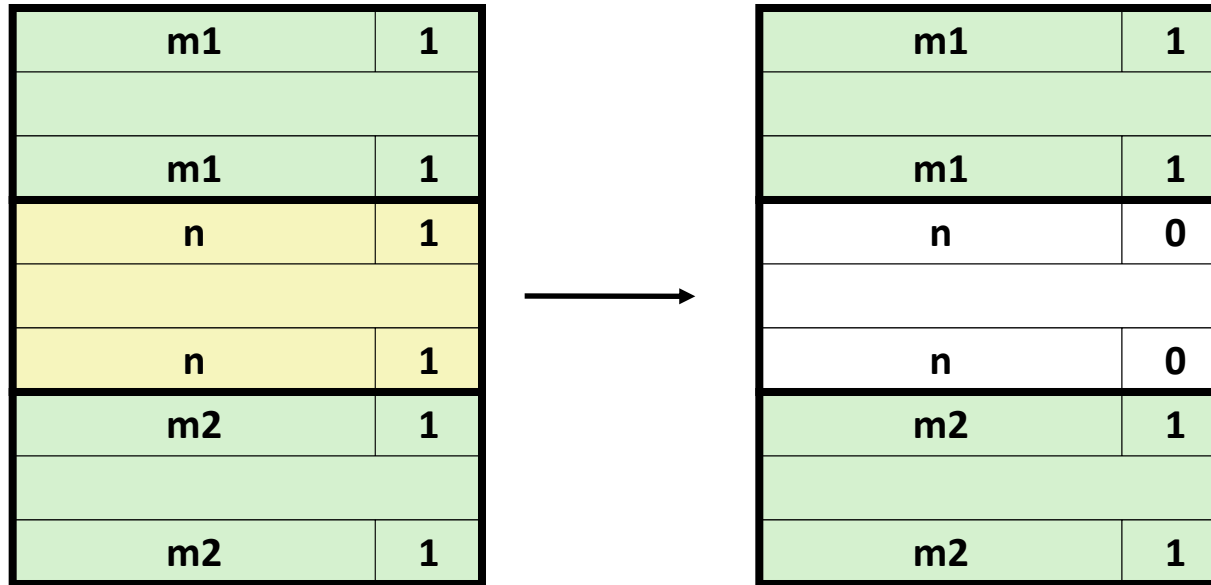
Constant Time Coalescing (Case 2)

m1	0
m1	0
n	1
n	1
m2	1
m2	1

→

n+m1	0
n+m1	0
m2	1
m2	1

26

m1	0		n+m1+m2	0
m1	0			
n	1	→		
n	1			
m2	0			
m2	0		n+m1+m2	0
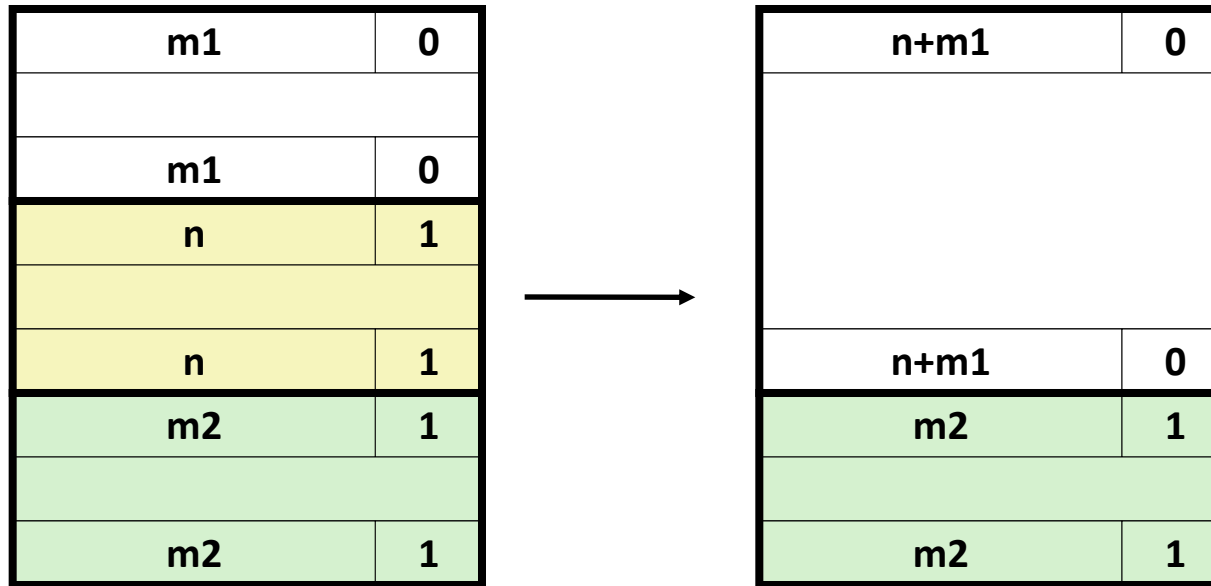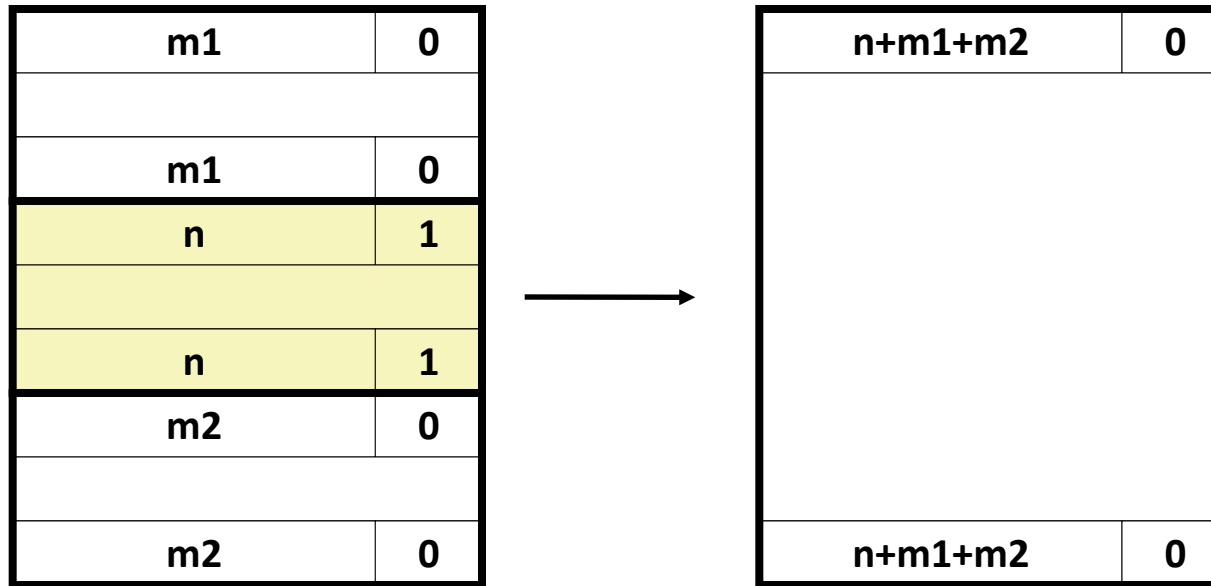
Disadvantages of Boundary Tags

- Internal fragmentation

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

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. Examples:
    - Coalesce as you scan the free list for `malloc`
    - Coalesce when the amount of 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` 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

Any Questions?

```
                   .text
       __start:    addi t1, zero, 0x18
                   addi t2, zero, 0x21
cycle:             beq t1, t2, done
                   slt t0, t1, t2
                   bne t0, zero, if_less
                   nop
                   sub t1, t1, t2
                   j cycle
                   nop
if_less:           sub t2, t2, t1
                   j cycle
done:              add t3, t1, zero
```