



NATIONAL RESEARCH
UNIVERSITY



Computer Architecture and Operating Systems

Lecture 5: Dynamic Memory Allocation

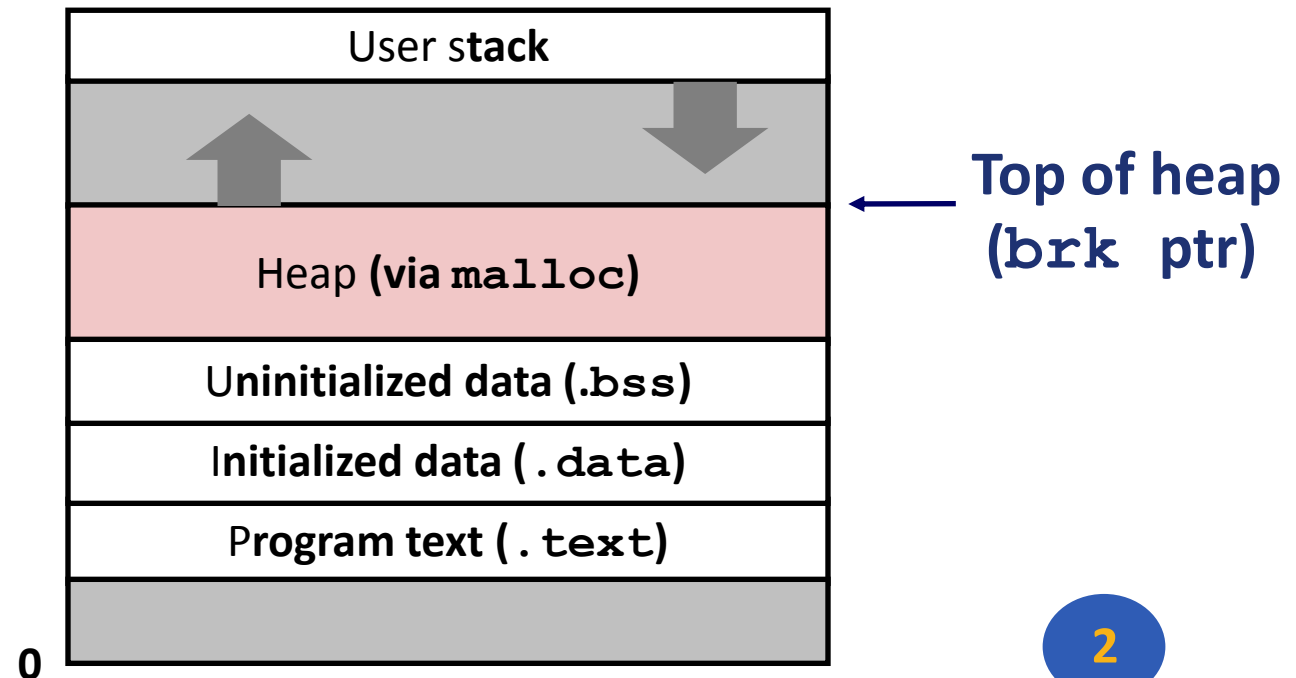
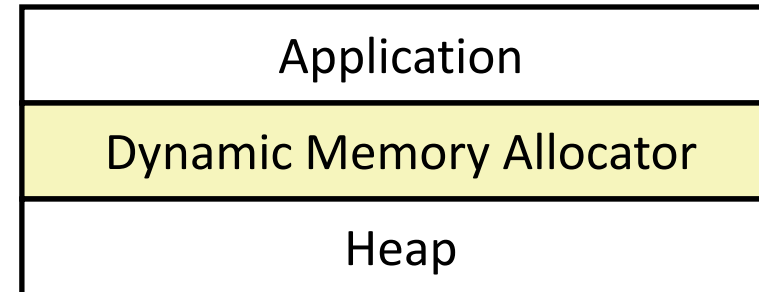
Andrei Tatarnikov

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*.



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

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

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

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

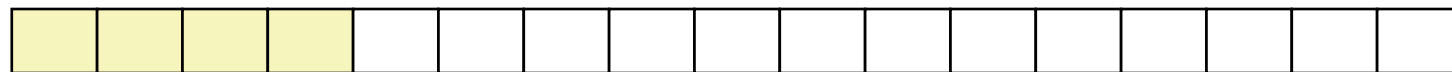
    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

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

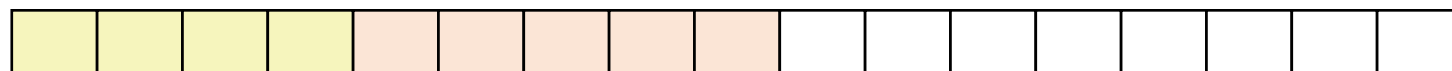
    /* Return allocated block to the heap */
    free(p);
}
```

Allocation Example

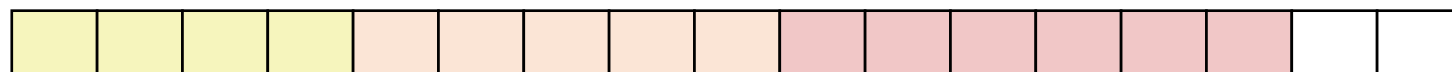
`p1 = malloc(4)`



`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:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- 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: Peak Memory Utilization

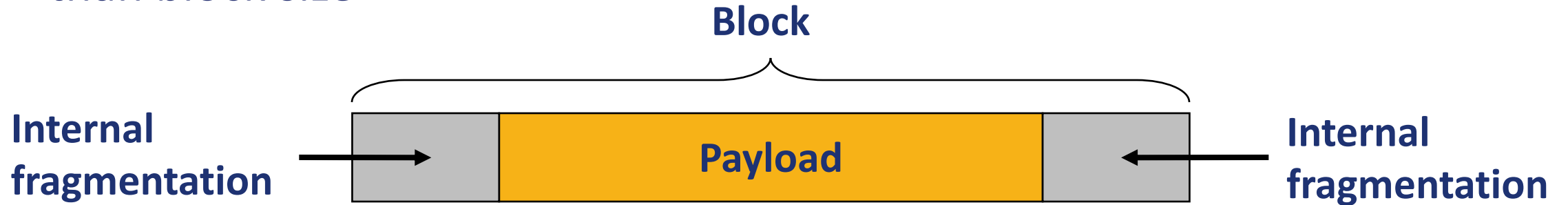
- Given some sequence of `malloc` and `free` requests:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- *Def: Aggregate payload P_k*
 - `malloc(p)` results in a block with a **payload** of `p` bytes
 - After request R_k has completed, the **aggregate payload** P_k is the sum of currently allocated payloads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

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

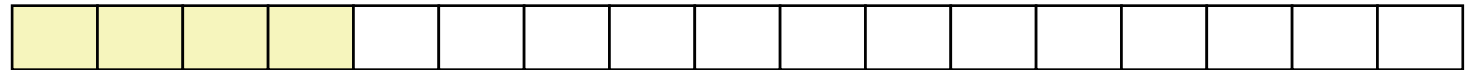


- Caused by
 - Overhead of maintaining heap data structures
 - Padding for alignment purposes
 - Explicit policy decisions (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of *previous* requests
 - Thus, easy to measure

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

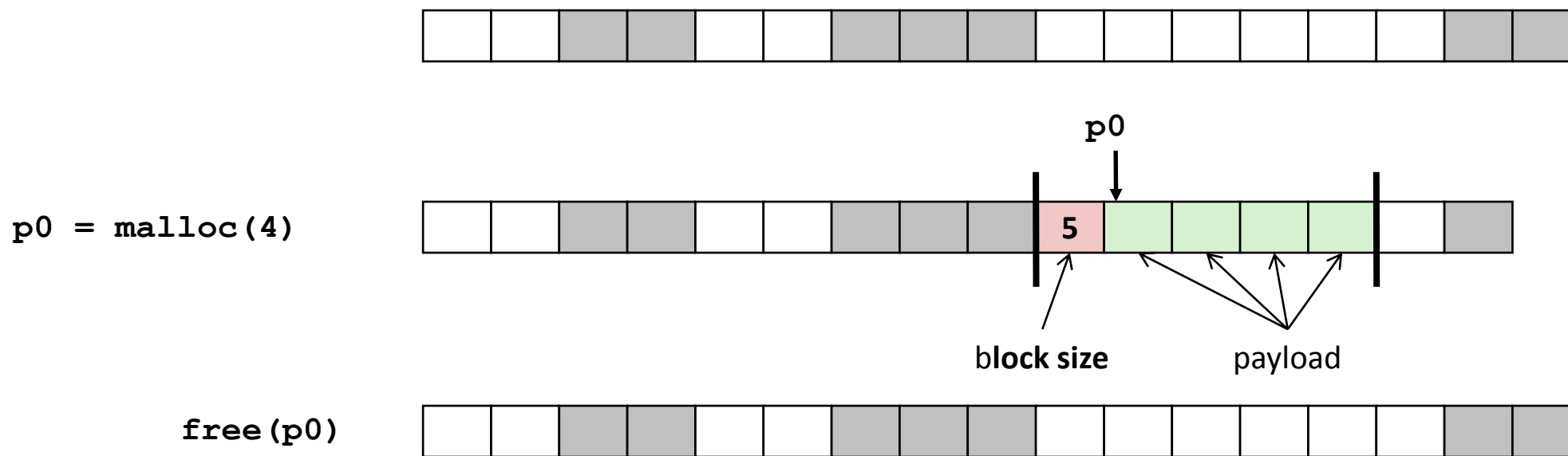
Implementation Issues

- 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

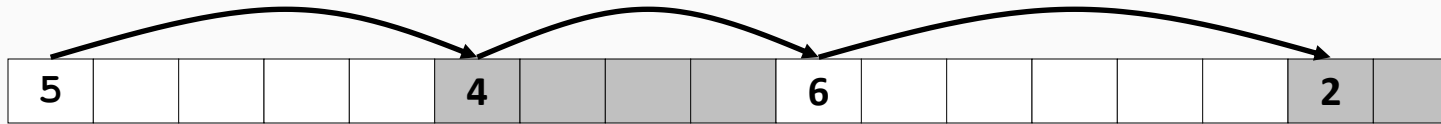
- Standard method

- 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



Keeping Track of Free Blocks

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



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

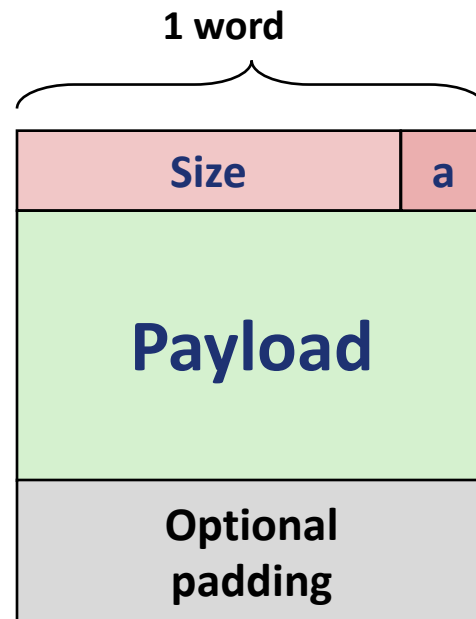


- Method 3: *Segregated free list*
 - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
 - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

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*



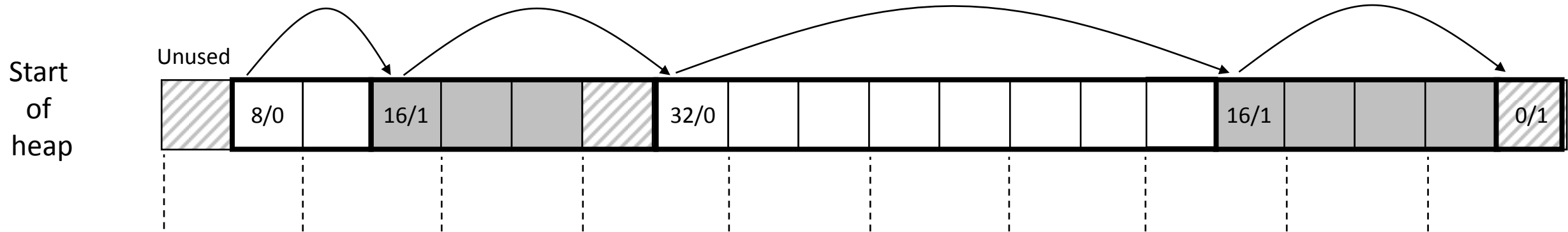
a = 1: Allocated block

a = 0: Free block

Size: block size

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

Detailed Implicit Free List Example



Double-word
aligned

Allocated blocks: shaded
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:*

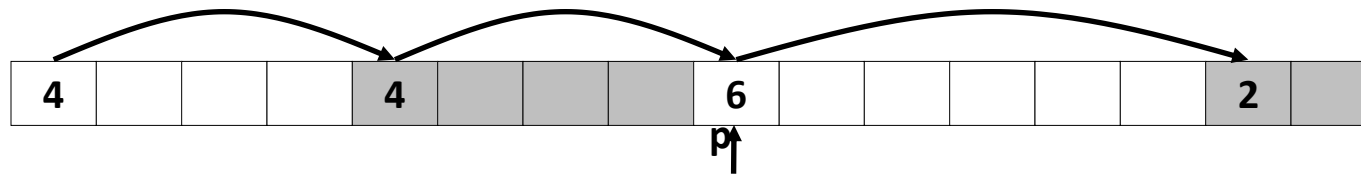
- Like first fit, but search list starting where previous search finished
- Should often be faster than first fit: avoids re-scanning unhelpful blocks
- Some research suggests that fragmentation is worse

■ *Best fit:*

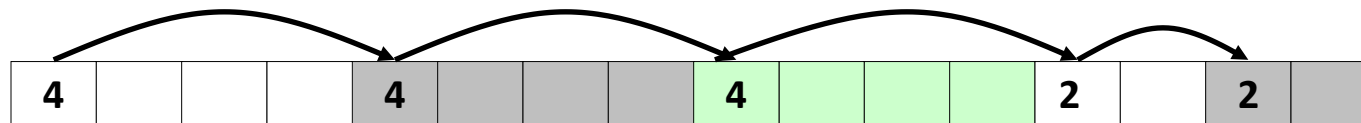
- Search the list, choose the *best* free block: fits, with fewest bytes left over
- Keeps fragments small—usually improves memory utilization
- Will typically run slower than first fit

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

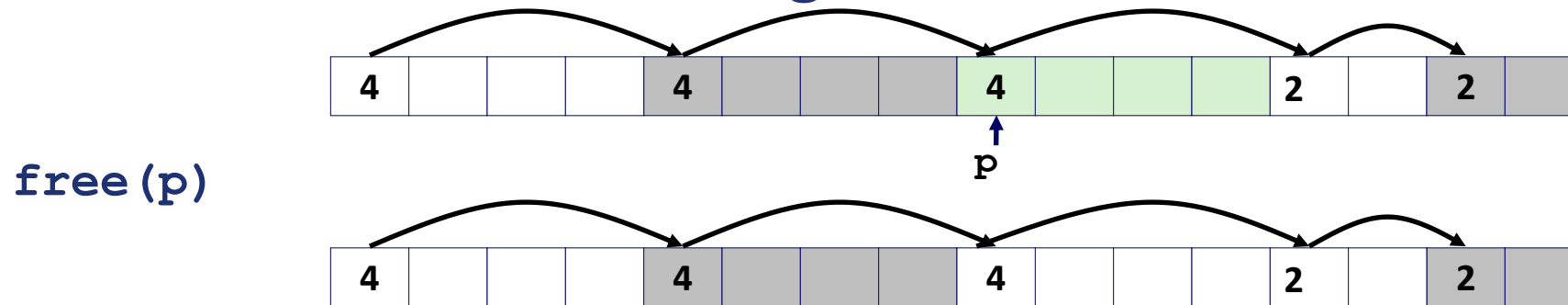
Implicit List: Freeing a Block

- Simplest implementation:

- Need only clear the “allocated” flag

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

- But can lead to “false fragmentation”

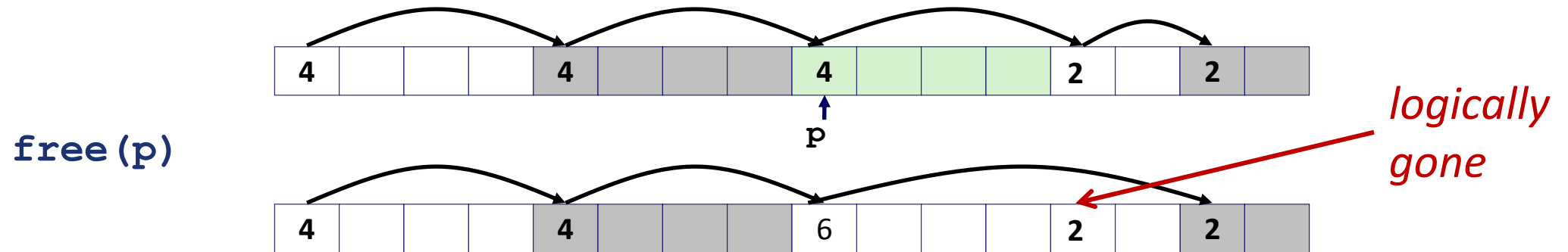


`malloc(5)` **Oops!**

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



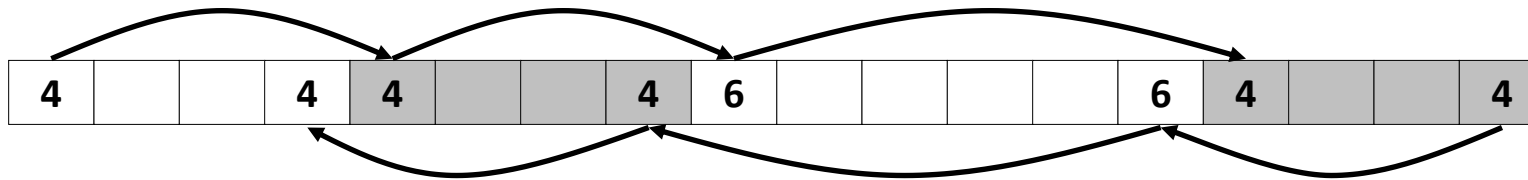
```
void free_block(ptr p) {  
    *p = *p & -2;           // clear allocated flag  
    next = p + *p;          // find next block  
    if ((*next & 1) == 0)  
        *p = *p + *next;    // add to this block if  
                             // not allocated  
}
```

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

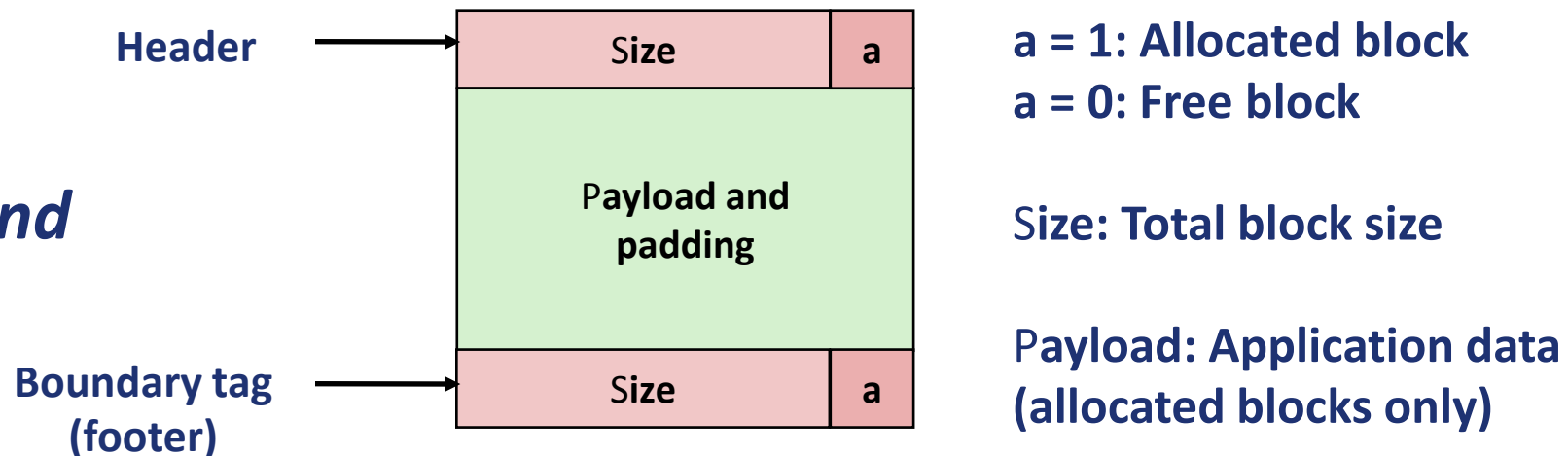
Implicit List: Bidirectional Coalescing

■ *Boundary tags* [Knuth73]

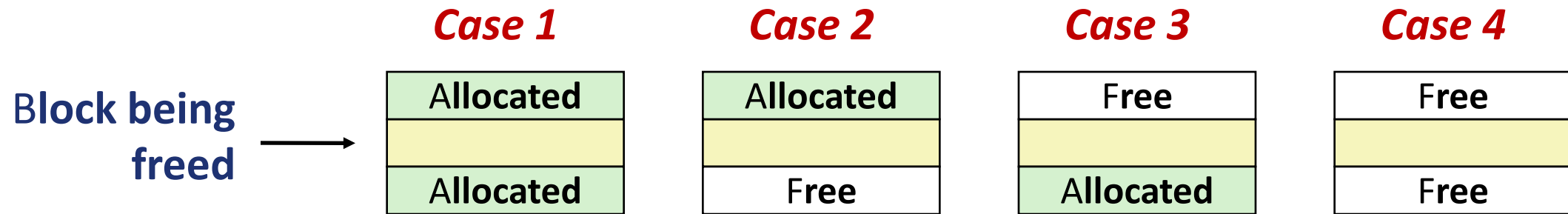
- Replicate size/allocated word at “bottom” (end) of free blocks
- Allows us to traverse the “list” backwards, but requires extra space
- Important and general technique!



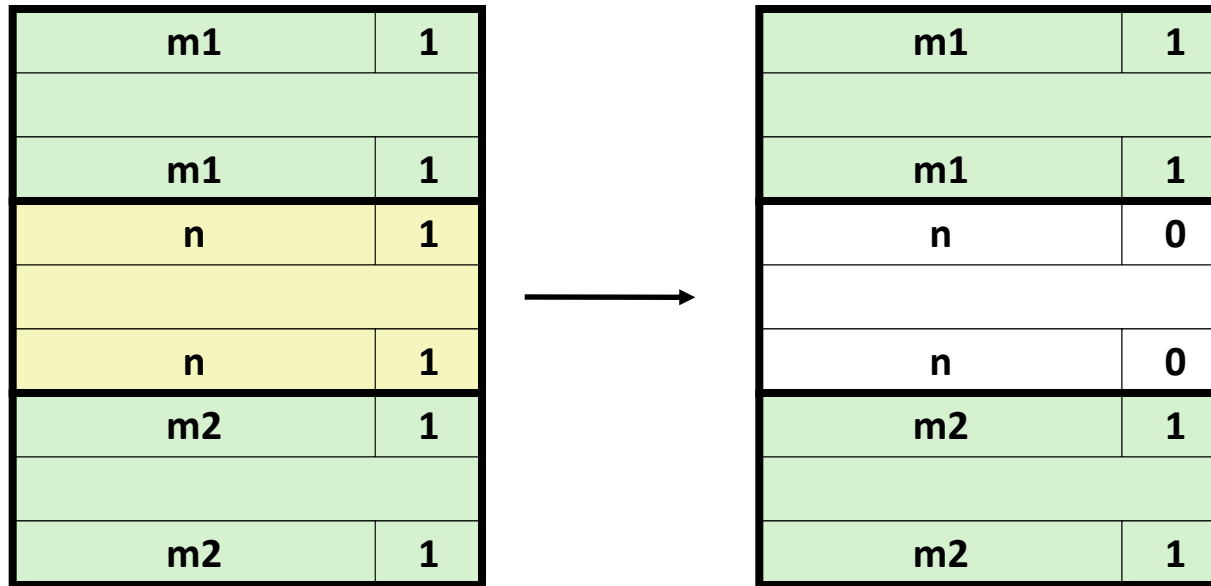
Format of allocated and free blocks



Constant Time Coalescing



Constant Time Coalescing (Case 1)



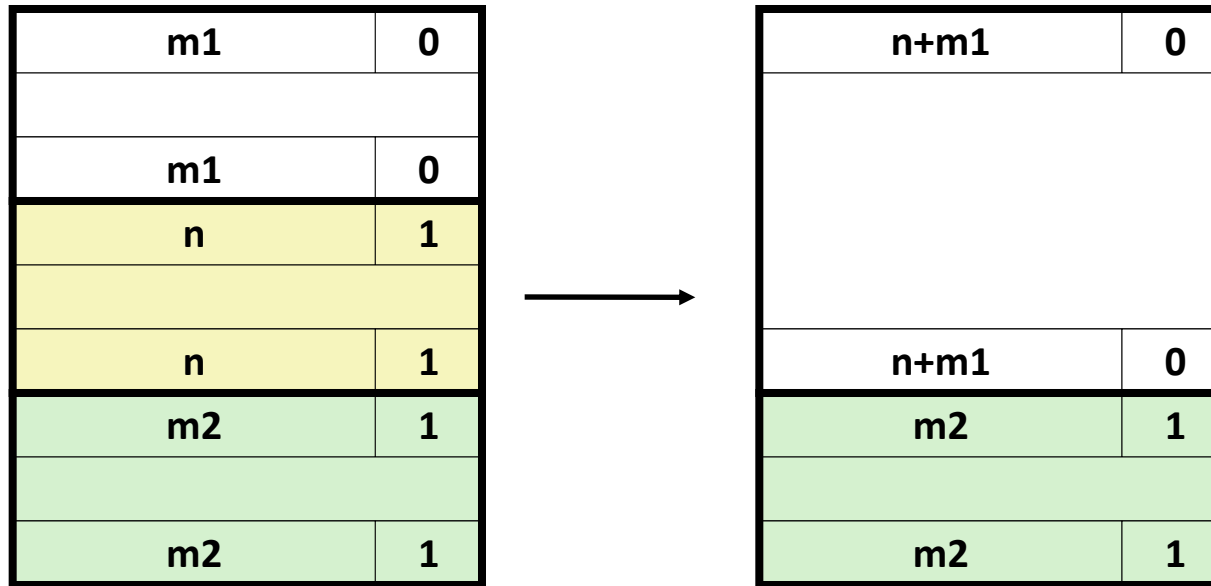
Constant Time Coalescing (Case 2)

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

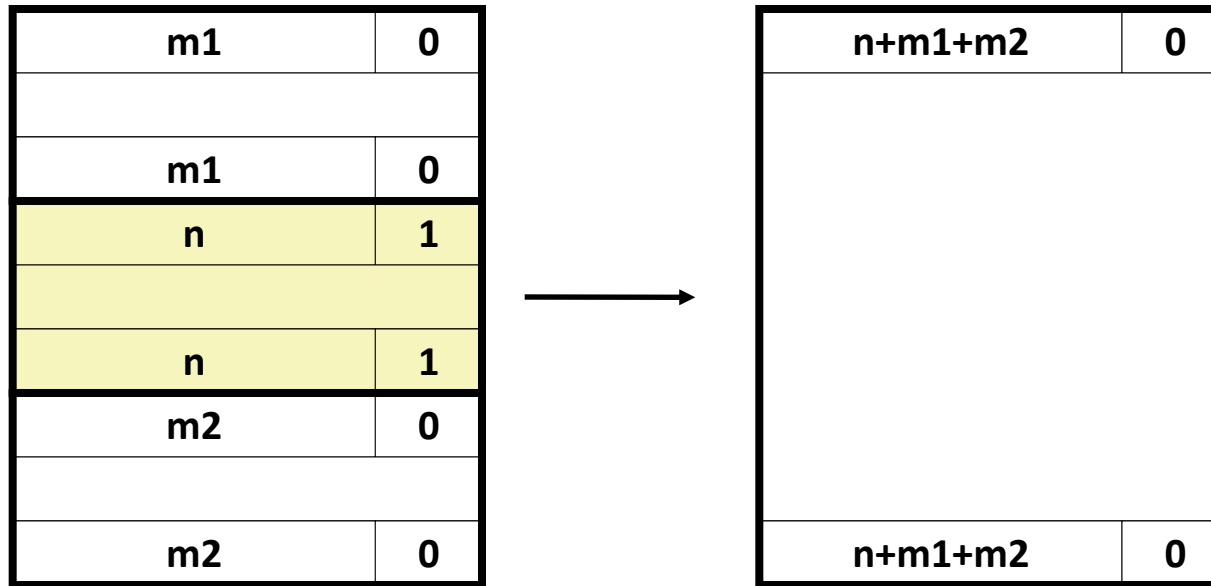


m1	1
m1	1
n+m2	0
n+m2	0

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?

Summary of Key Allocator Policies

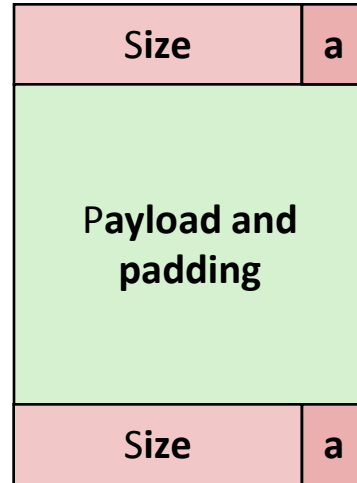
- Placement policy:
 - First-fit, next-fit, best-fit, etc.
 - Trades off lower throughput for less fragmentation
 - **Interesting observation:** segregated free lists (next lecture)
approximate a best fit placement policy without having to search entire free list
- Splitting policy:
 - When do we go 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

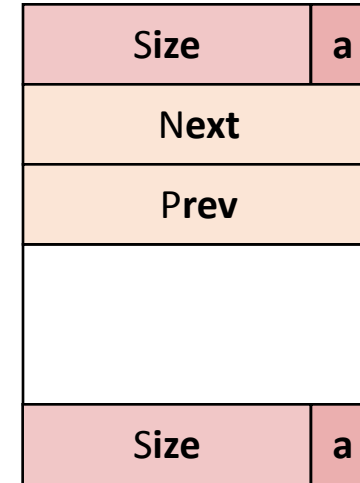
- Implementation: very simple
- Allocate cost:
 - linear time worst case
- Free cost:
 - constant time worst case
 - even with coalescing
- Memory usage:
 - will depend on placement policy
 - First-fit, next-fit or best-fit
- Not used in practice for `malloc/free` because of linear-time allocation
 -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

Method2: Explicit Free Lists

Allocated



Free



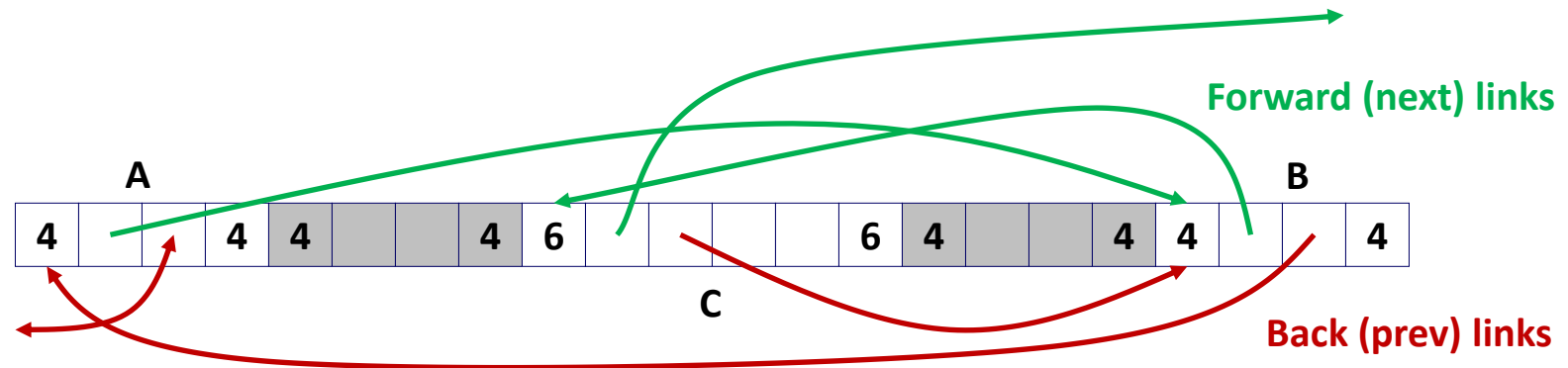
- Maintain list(s) of *free* blocks, not *all* blocks
 - The “next” free block could be anywhere
 - Need to store forward/back pointers, not just sizes
 - Still need boundary tags for coalescing
 - We track only free blocks, so we can use payload area

Explicit Free Lists

- Logically:



- Physically: blocks can be in any order



Explicit Free Lists: Insertion Policy

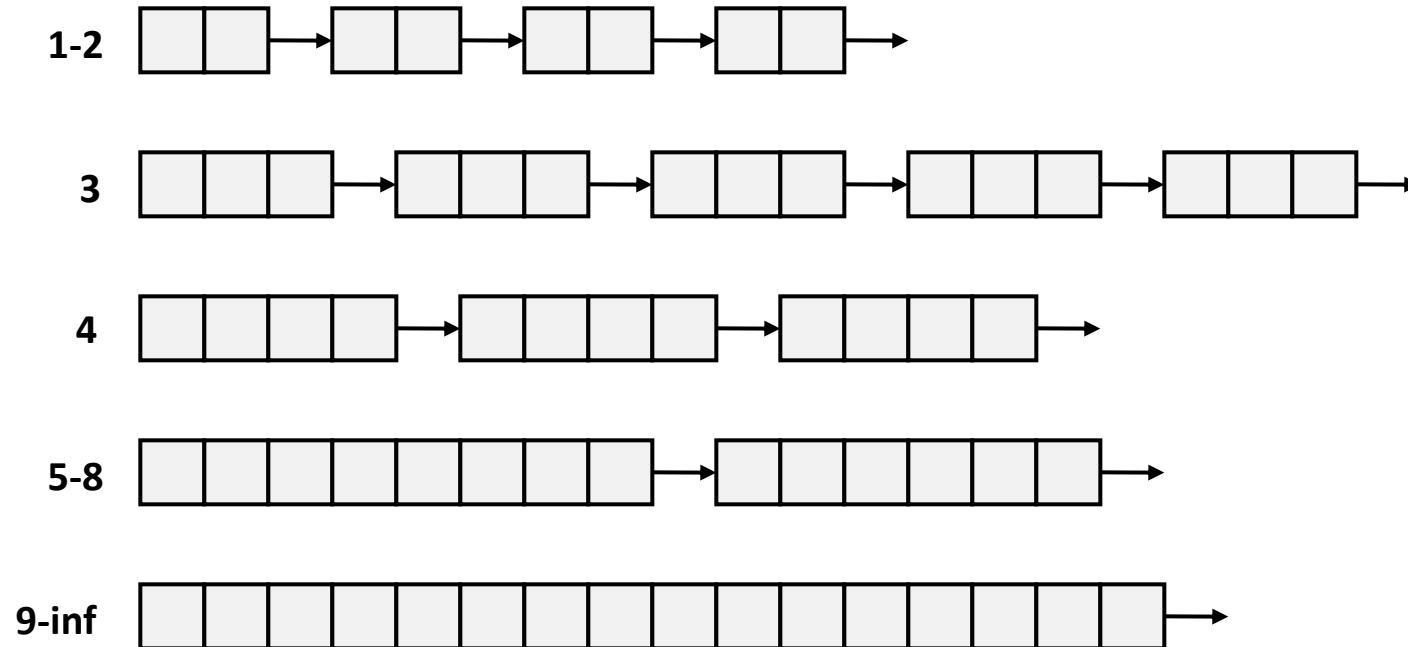
- **Insertion policy:** Where in the free list do you put a newly freed block?
- **LIFO (last-in-first-out) policy**
 - Insert freed block at the beginning of the free list
 - **Pro:** simple and constant time
 - **Con:** studies suggest fragmentation is worse than address ordered
- **Address-ordered policy**
 - Insert freed blocks so that free list blocks are always in address order:
 $addr(prev) < addr(curr) < addr(next)$
 - **Con:** requires search
 - **Pro:** studies suggest fragmentation is lower than LIFO

Explicit Free Lists: Summary

- Comparison to implicit list:
 - Allocate is linear time in number of *free* blocks instead of *all* blocks
 - *Much faster* when most of the memory is full
 - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
 - Some extra space for the links (2 extra words needed for each block)
 - Does this increase internal fragmentation?
- Most common use of linked lists is in conjunction with segregated free lists
 - Keep multiple linked lists of different size classes, or possibly for different types of objects

Method 3: Segregated List Allocators

- Each *size class* of blocks has its own free list



- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

Segregated List Allocators

- Given an array of free lists, each one for some size class
- To allocate a block of size n :
 - Search appropriate free list for block of size $m > n$
 - If an appropriate block is found:
 - Split block and place fragment on appropriate list (optional)
 - If no block is found, try next larger class
 - Repeat until block is found
- If no block is found:
 - Request additional heap memory from OS (using `sbrk()`)
 - Allocate block of n bytes from this new memory
 - Place remainder as a single free block in largest size class.

Segregated List Allocators

- To free a block:
 - Coalesce and place on appropriate list
- Advantages of segregated list allocators:
 - Higher throughput
 - log time for power-of-two size classes
 - Better memory utilization
 - First-fit search of segregated free list approximates a best-fit search of entire heap
 - Extreme case: giving each block its own size class is equivalent to best-fit

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
```