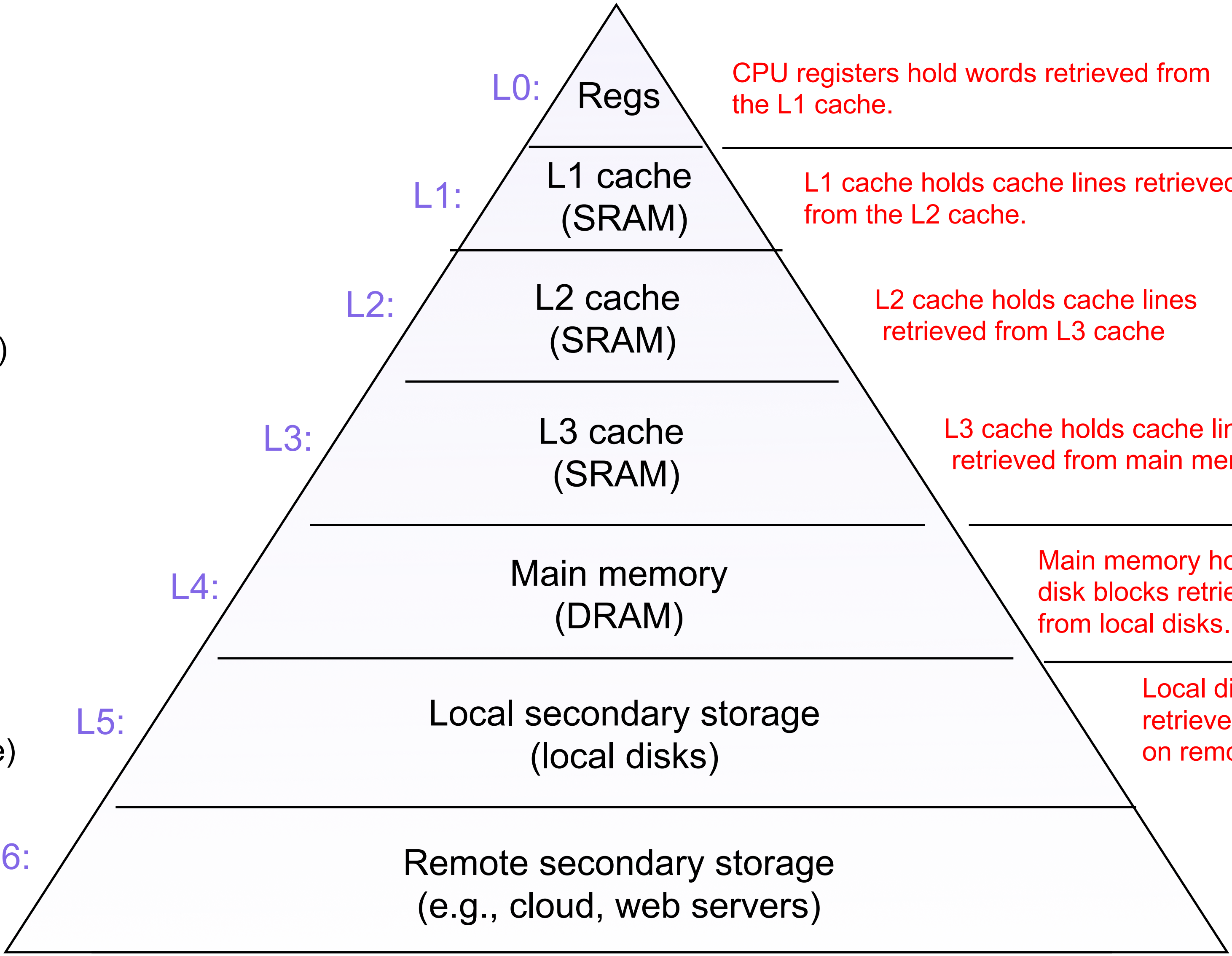
Lecture 12: Dynamic Memory

CS 105

October 15, 2019

Memory Hierarchy

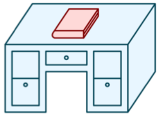


Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices

L0: Regs

CPU registers hold words retrieved from the L1 cache.

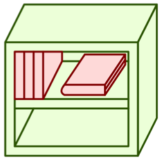


L1: L1 cache (SRAM)

L1 cache holds cache lines retrieved from the L2 cache.

L2: L2 cache (SRAM)

L2 cache holds cache lines retrieved from L3 cache

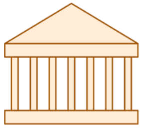


L3: L3 cache (SRAM)

L3 cache holds cache lines retrieved from main memory.

L4: Main memory (DRAM)

Main memory holds disk blocks retrieved from local disks.



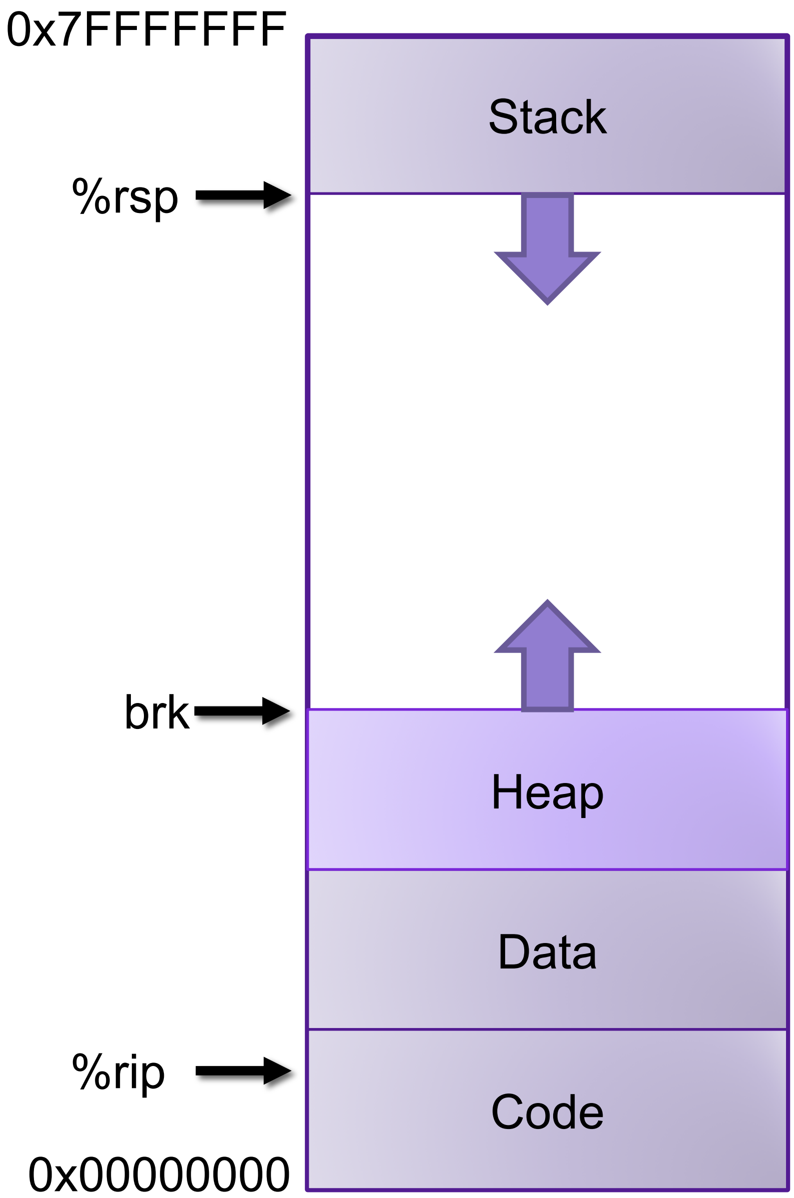
L5: Local secondary storage (local disks)

Local disks hold files retrieved from disks on remote servers

L6: Remote secondary storage (e.g., cloud, web servers)

Memory

- the heap is an area of memory maintained by a dynamic memory allocator
- programmers can use the dynamic memory allocator to acquire additional memory at run time
  - e.g., for data structures whose size is not known at compile time
- the operating system kernel maintains a variable brk that points to the top of the heap



Dynamic Memory Allocation

Dynamic memory allocator

- Manages the heap
  - organizes the heap as a collection of (variable-size) **blocks**, each of which is either **allocated** or **free**
  - allocates and deallocates memory
  - may ask OS for additional heap space
- Part of the process's runtime system
  - Linked into program

Example dynamic memory allocators

- **`malloc`** and **`free`** in C
- **`new`** and **`delete`** in C++

explicit allocators

- object creation & garbage collection in Java
- object creation & garbage collection in Python

implicit allocators

Allocation Example using `malloc`

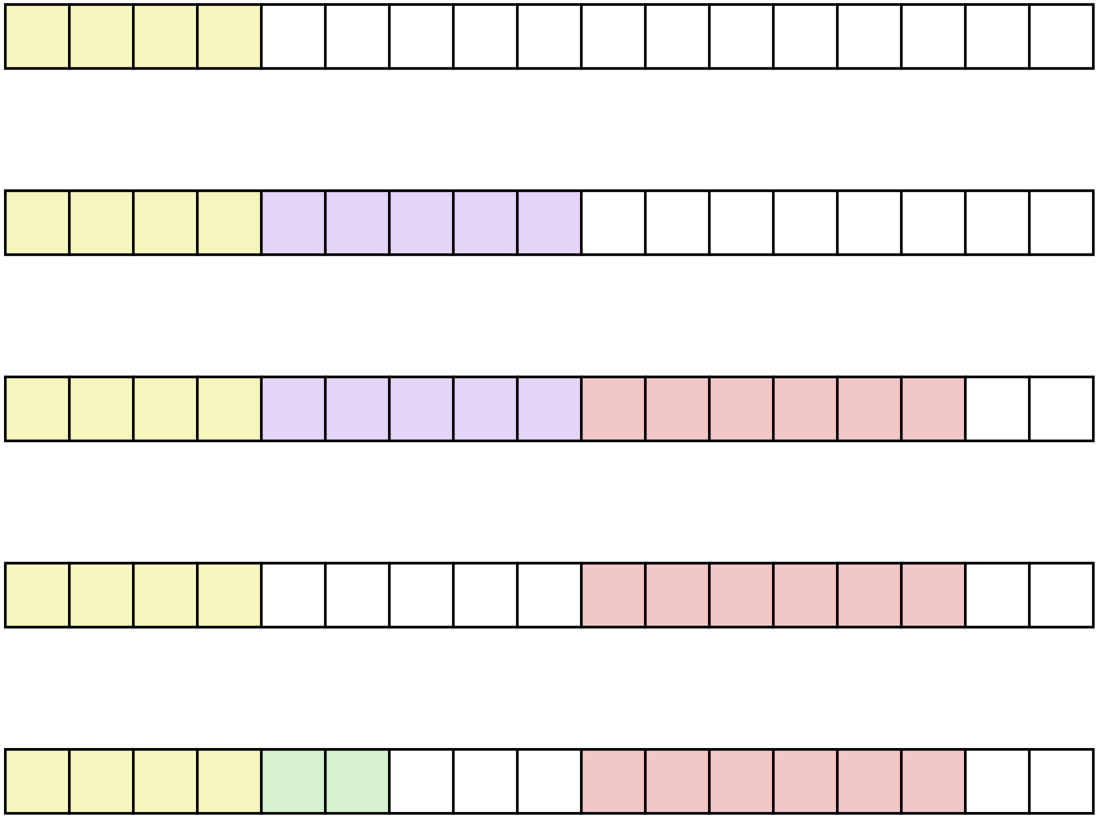
```
#include <stdio.h>
#include <stdlib.h>
void foo(int n) {
    int i, *p;

    /* 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)`

Allocator Requirements

- **Must handle arbitrary request sequences:**
  - cannot control number, size, or order of requests
  - (but we'll assume that each free request corresponds to an allocated block)
- **Must respond immediately:**
  - no reordering or buffering requests
- **Must not modify allocated blocks:**
  - can only allocate from free memory on the heap
  - cannot modify or move blocks once they are allocated
- **Must align blocks:**
  - 8-byte (x86) or 16-byte (x86-64) alignment on Linux
  - Ensures that allocated blocks can hold any type of data
- **Must only use the heap:**
  - any data structures used by the allocator must be stored in the heap

First Example: A Simple Allocator

```
void *brk;  // top of heap

void *malloc (size_t size) {
  void *p = brk;
  brk += align(size);
  return p;
}

void free (void *ptr) {
  // do nothing
}
```

Advantages
- Blazing fast
- Simple

Disadvantages
- Memory is never recycled

Performance Goals

- **Throughput** and **Memory Utilization**
  - These goals are often conflicting

- Throughput
  - Number of completed requests per unit time
  - Example: if your allocator processes 5,000 **`malloc`** calls and 5,000 **`free`** calls in 10 seconds then throughput is 1,000 operations/second

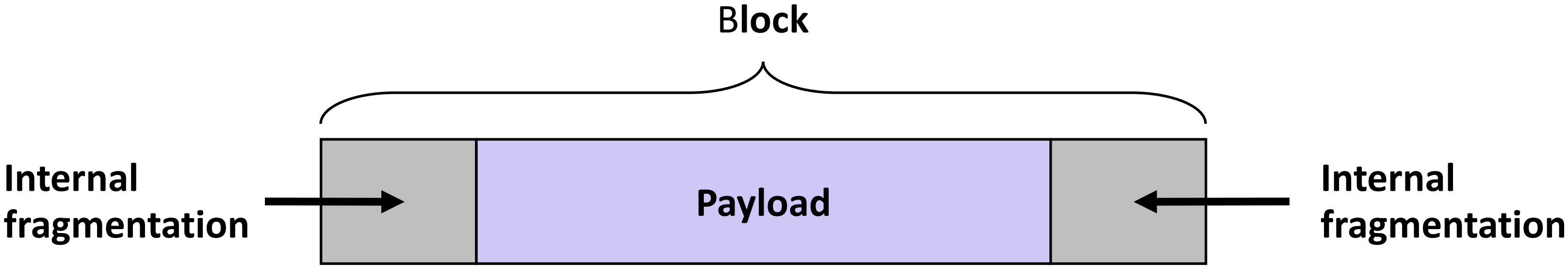
- Peak Memory Utilization
  - Minimize wasted space

Peak Memory Utilization

- Given some sequence of **`malloc`** and **`free`** requests $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$ the **peak memory utilization** after request k is $U_k = \frac{\max_{i \le k} P_i}{H_k}$
  - P_i, is the aggregate payload, i.e., the sum of the currently allocated payloads after request i, where the payload of **`malloc(p)`** is **`p`** bytes
  - H_k is the current heap size
    - Assume H_k is monotonically nondecreasing

Utilization Blocker: Internal Fragmentation

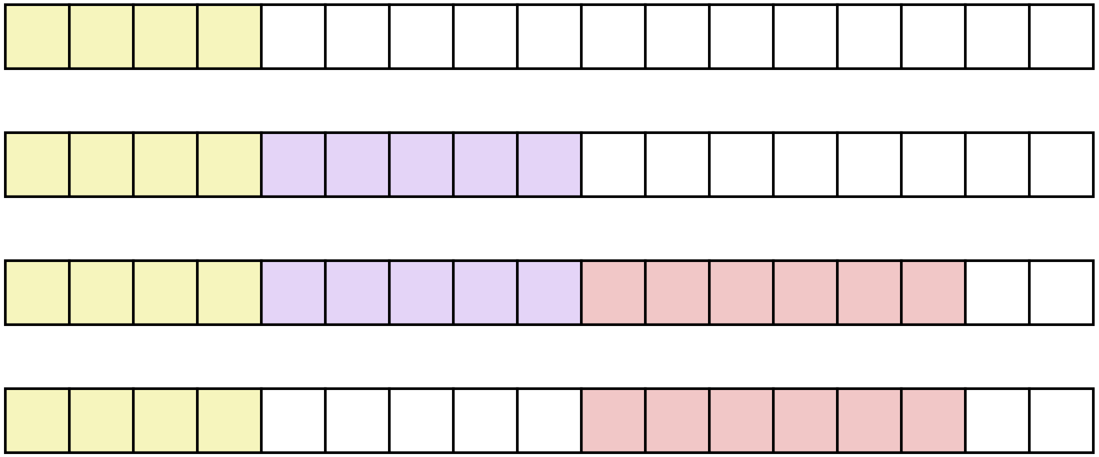
- For a given block, ***internal fragmentation*** occurs if payload is smaller than block size



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

Utilization Blocker: 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

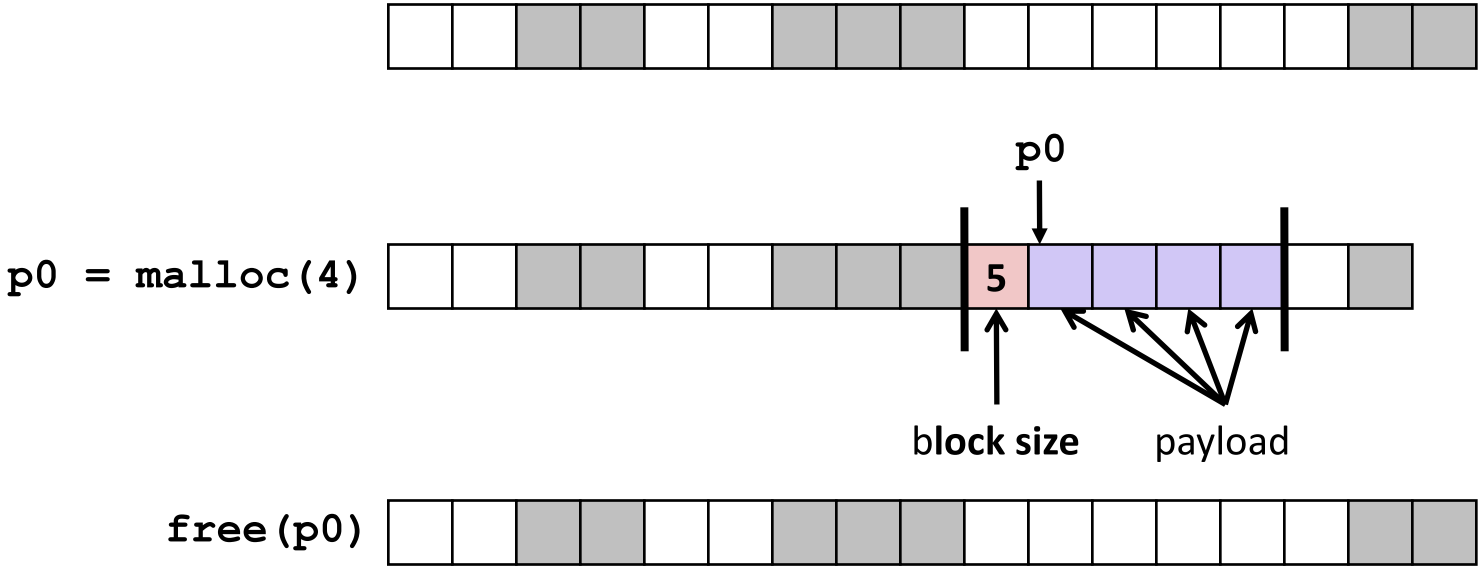
Challenges

- Strategic: maximize throughput and peak memory utilization

- Implementation:
  - How do we know how much memory to free given just a pointer?

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 (4 byte) word for every allocated block



`p0 = malloc(4)`

`free(p0)`

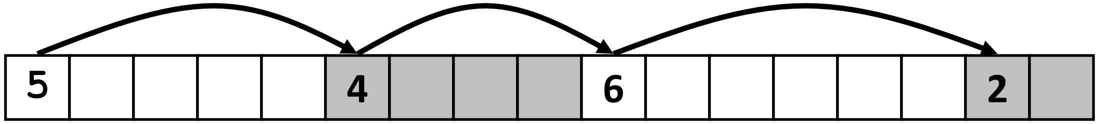
Challenges

- Strategic: maximize throughput and peak memory utilization

- Implementation:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?

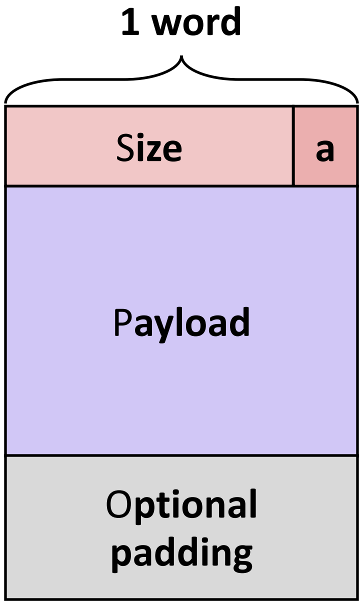
Keeping Track of Free Blocks

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



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



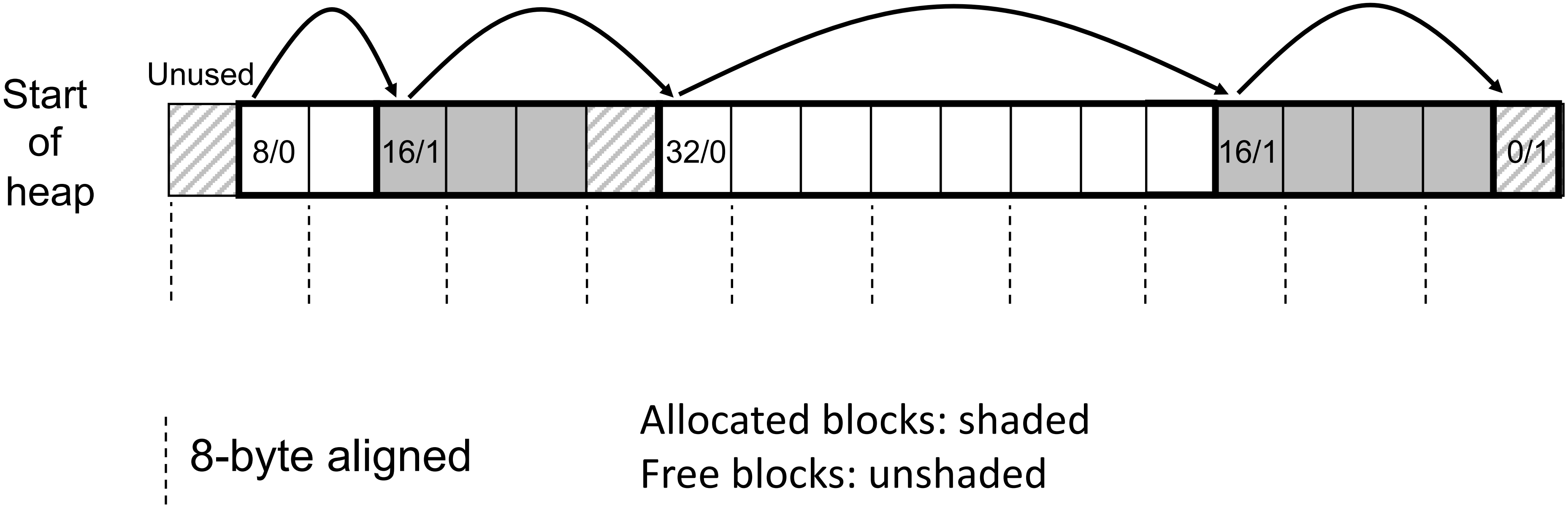
Format of allocated and free blocks

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

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



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

Challenges

- Strategic: maximize throughput and peak memory utilization

- Implementation:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation—many might fit?

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

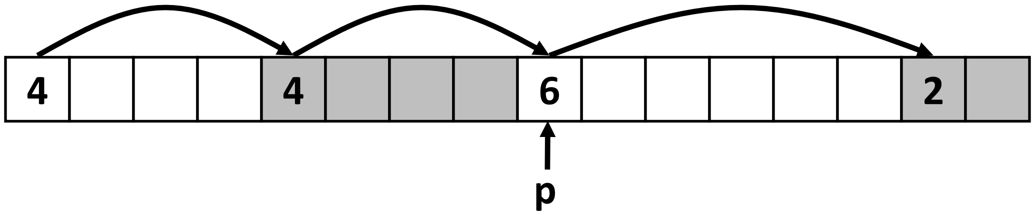
Challenges

- Strategic: maximize throughput and peak memory utilization

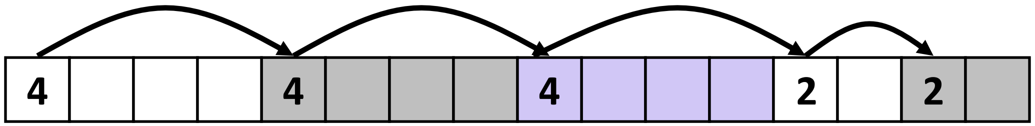
- Implementation:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation—many might fit?
  - What do we do with the extra space when allocating a structure that is smaller than the free block it is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

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
```

Challenges

- Strategic: maximize throughput and peak memory utilization

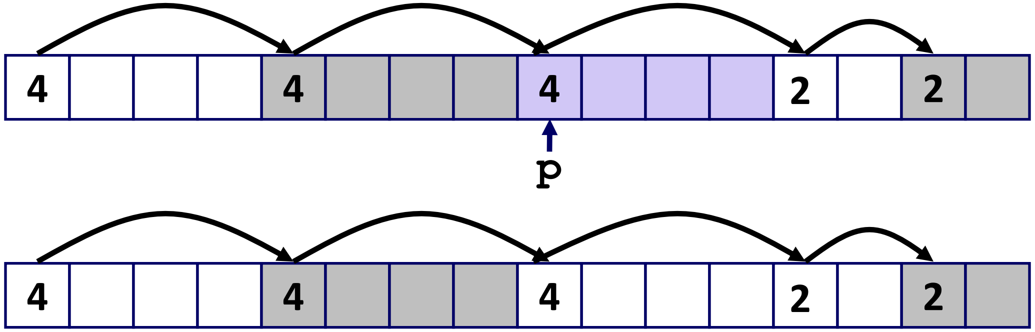
- Implementation:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation—many might fit?
  - What do we do with the extra space when allocating a structure that is smaller than the free block it is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



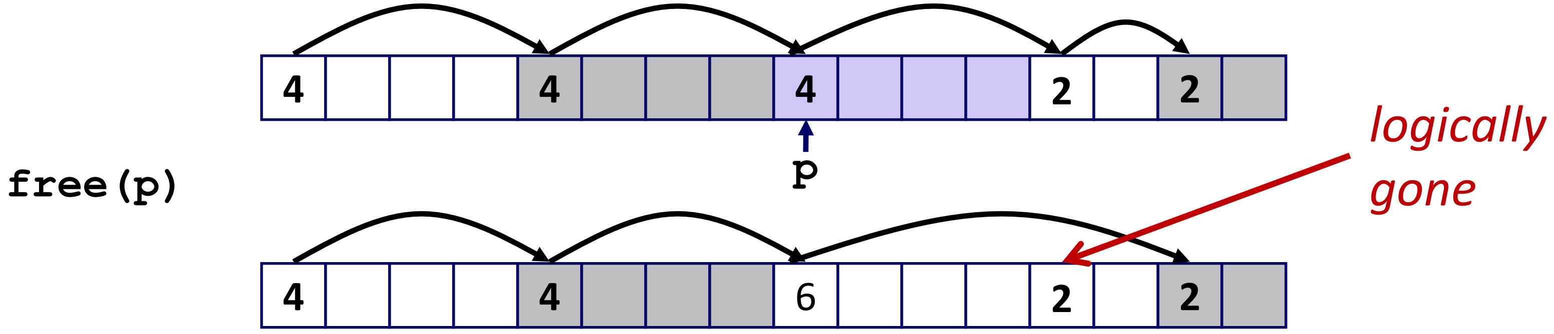
`free(p)`

`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

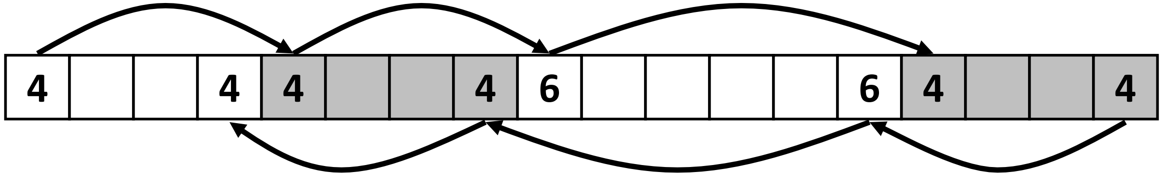


```
void free_block(ptr p) {
     *p = *p & -2;           // clear allocated flag
     next = p + *p;          // find next block
     if ((*next & 1) == 0)
       *p = *p + *next;      // add to this block if
}                            //    not allocated
```

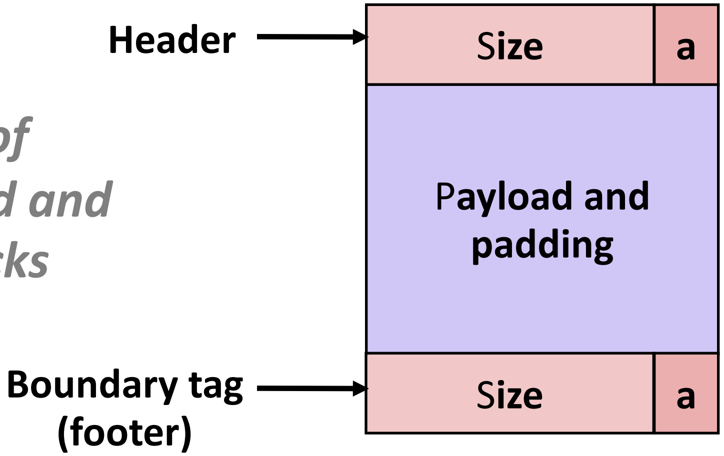
- But how do we coalesce with *previous* block?

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

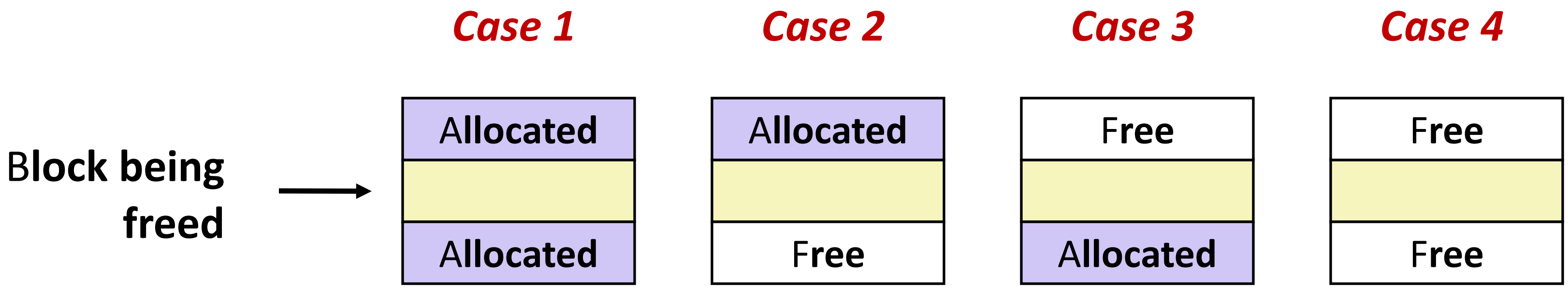


a = 1: Allocated block
a = 0: Free block

Size: Total block size

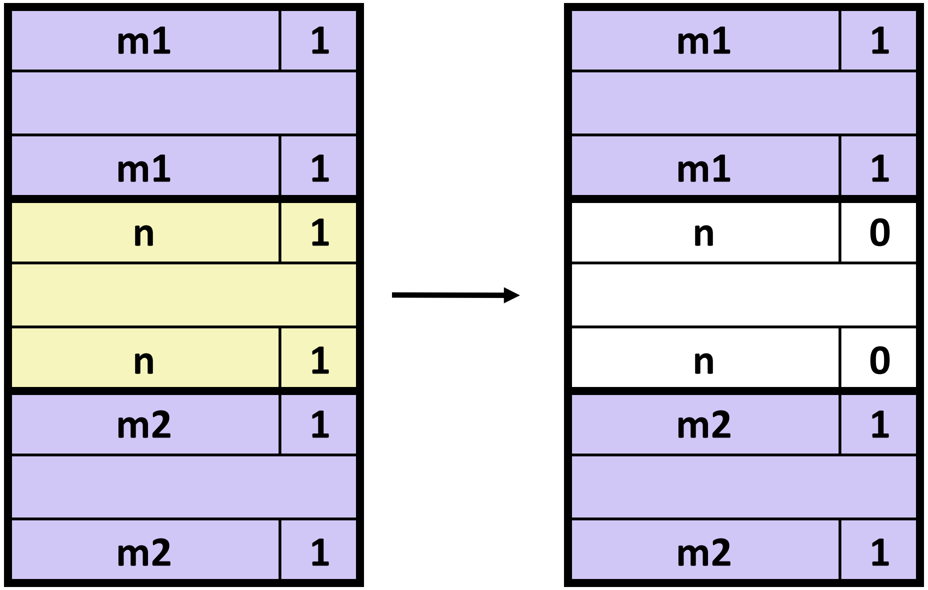
Payload: Application data (allocated blocks only)

Constant Time Coalescing

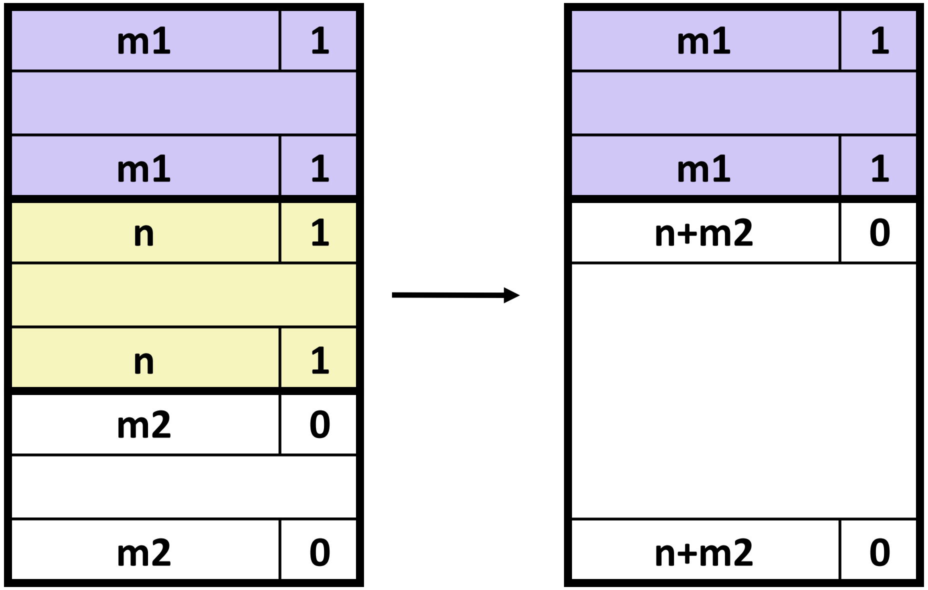


	Case 1	*Case 2*	*Case 3*	*Case 4*
	Allocated	Allocated	Free	Free
Block being freed →				
	Allocated	Free	Allocated	Free

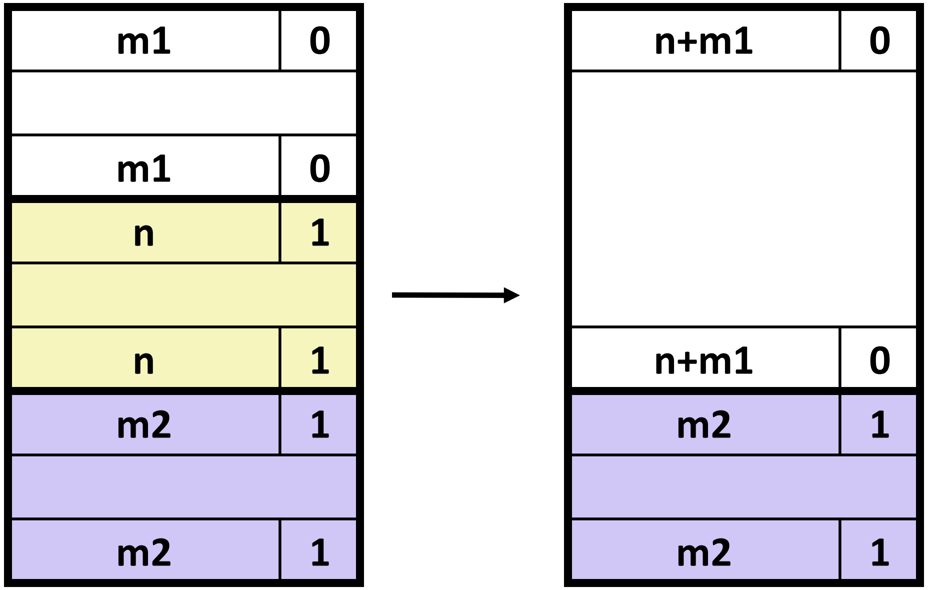
Constant Time Coalescing (Case 1)



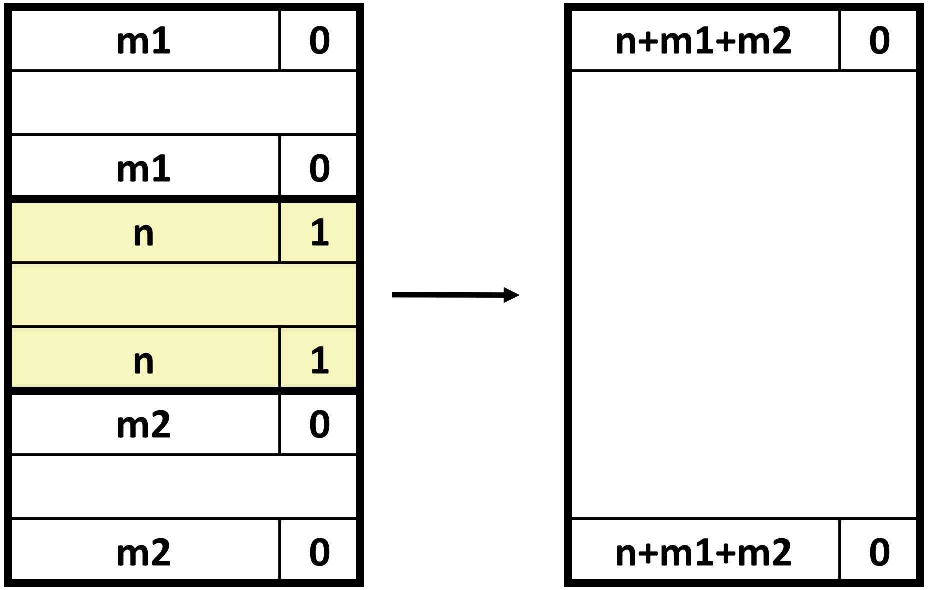
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



m1 0

m1 0

n 1

n 1

m2 0

m2 0

n+m1+m2 0

n+m1+m2 0

Implicit Lists: Summary

- Implementation: very simple

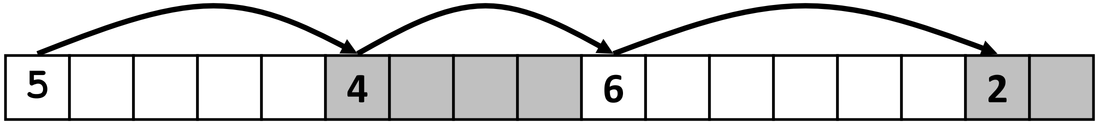
- Allocate cost: linear time in the worst case
- Free cost: constant time worst case–even with coalescing
- Memory usage: depends on the placement policy
  - First-fit, next-fit, or best-fit

- Not used in practice for **`malloc`**/**`free`** because of linear-time allocation
  - used in many special purpose applications

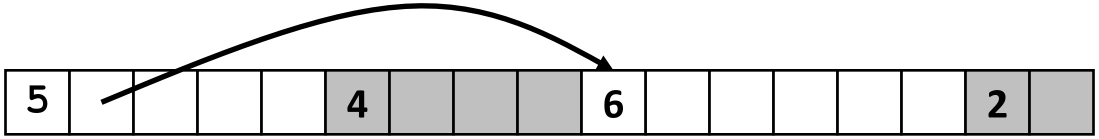
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators

Keeping Track of Free Blocks

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



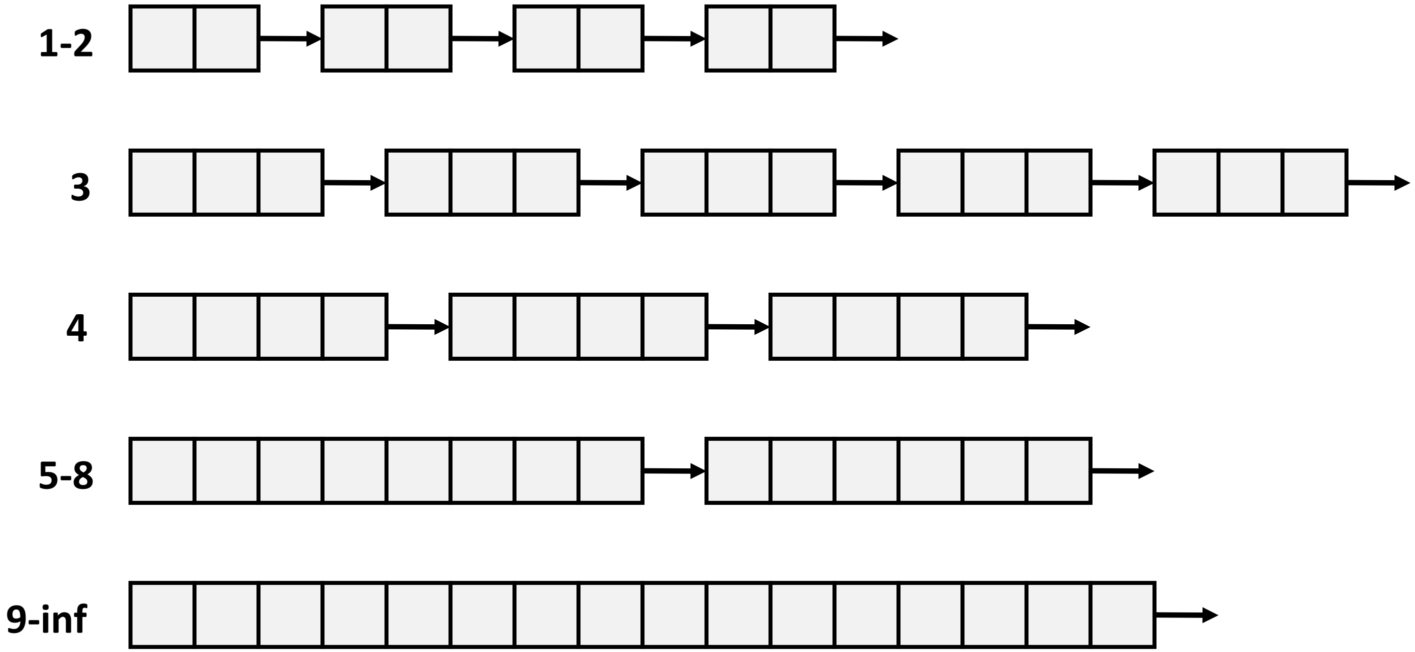
- Method 2: ***Explicit list*** among the free blocks using pointers



- 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

Segregated Lists

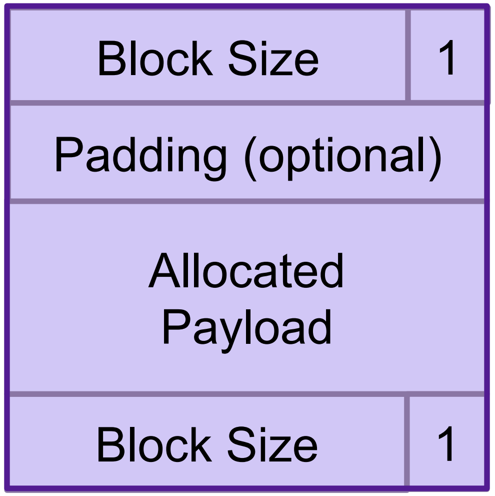
- Each **size class** of blocks has its own free list



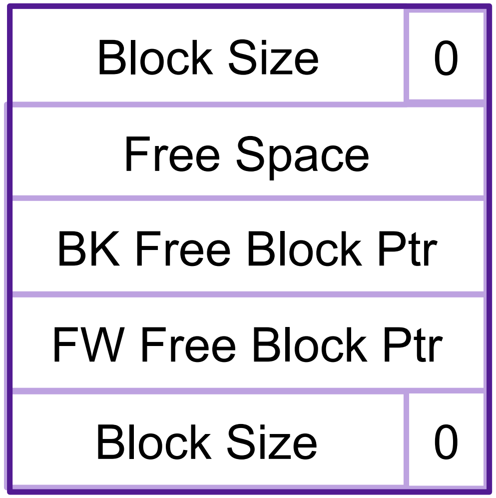
- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

Segregated List Blocks

Allocated Blocks


Block Size | 1
Padding (optional)
Allocated Payload
Block Size | 1

Free Blocks


Block Size | 0
Free Space
BK Free Block Ptr
FW Free Block Ptr
Block Size | 0

Seglist Allocator

- 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 in any list:
    - 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.

Seglist Allocator (cont.)

- To free a block:
  - Coalesce and place on appropriate list

- Advantages of seglist 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.

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - segregated free lists 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