Lecture 16: Dynamic Memory

CS 105

March 24, 2019

Virtual Memory

- Each process has as much memory as it needs
  - … within limits of the hardware, architecture. and operating system

- Each process has exclusive access to its memory
  - … with a few exceptions
  - Supports multitasking

- Disk is used as a backup for memory
  - … or physical memory is a “cache” for the pages on disk

- address translation is managed by hardware

Virtual Memory

- Memory is managed by *pages*
  - For us, a page is a 4KB block of memory
  - Could be other sizes, or even mixed sizes

- An address is composed of
  - Offset within page (lower bits, here 12 bits)
  - Page number (upper bits—at most 52 bits, actually fewer)

- Each process has its own mapping from *virtual* page numbers to *physical* page numbers
  - Some pages are in physical memory
  - Other pages are stored on the disk

Problems with Virtual Memory

- What happens when there is a
  - TLB miss?
  - Page fault?
  - Context switch?

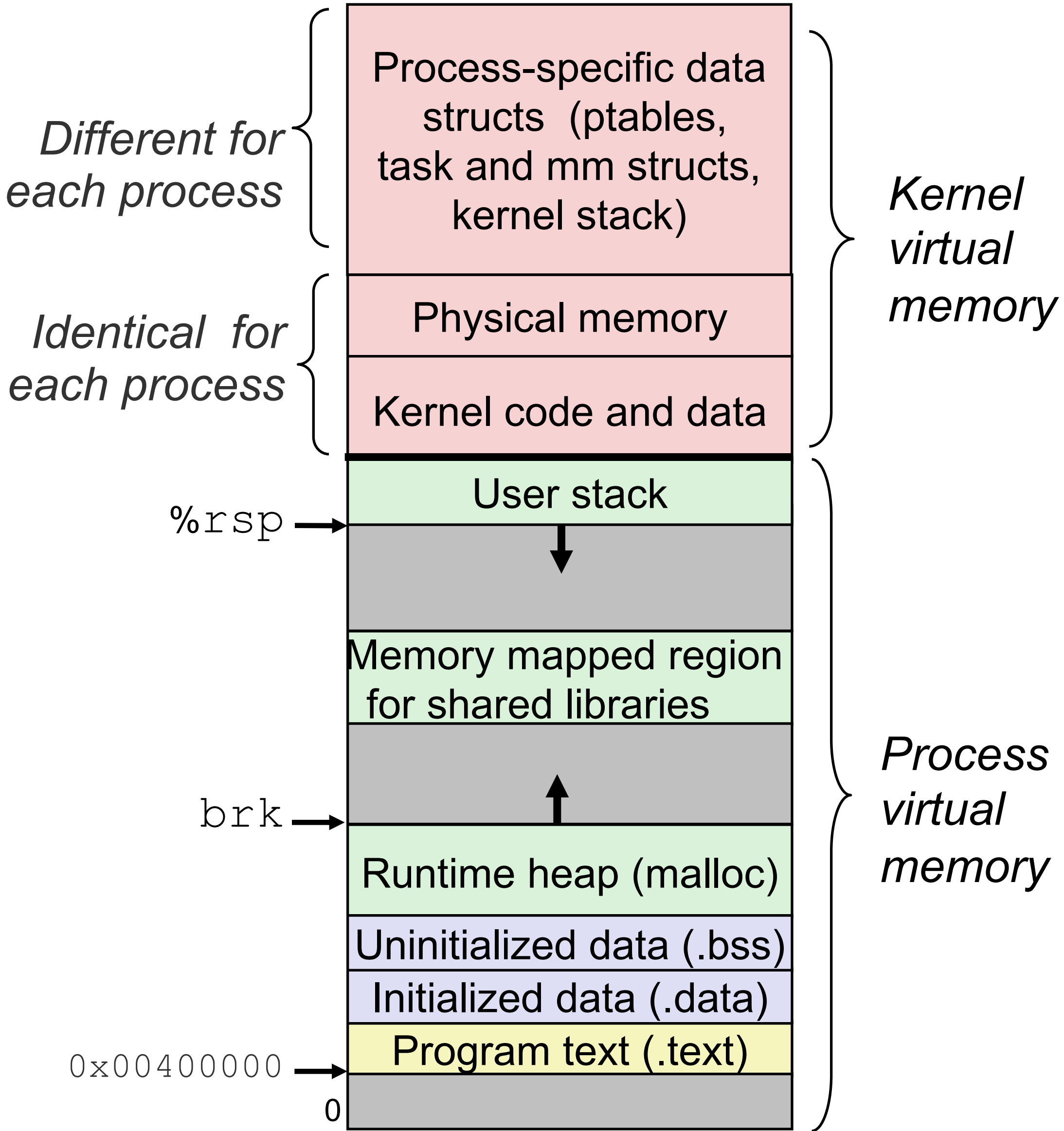
The Operating System

- an **operating system** is a layer of software interposed between the hardware and application programs
  - protects the hardware from misuse
  - provides applications with simple and uniform mechanisms for manipulating low-level hardware devices
- the operating system **kernel** is the portion of the operating system code that is always in memory.
- kernel implements handlers for exceptions (e.g., faults, interrupts)
- application programs transfer control to the kernel by executing special **system call** instructions

Example system calls in Linux x86-64

Number	Name	Description
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
9	`mmap`	Map memory page to file
12	`brk`	Reset top of heap
39	`getpid`	Get process id
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process

Virtual Address Space of a Linux Process



Lots of unused areas—
more than is shown

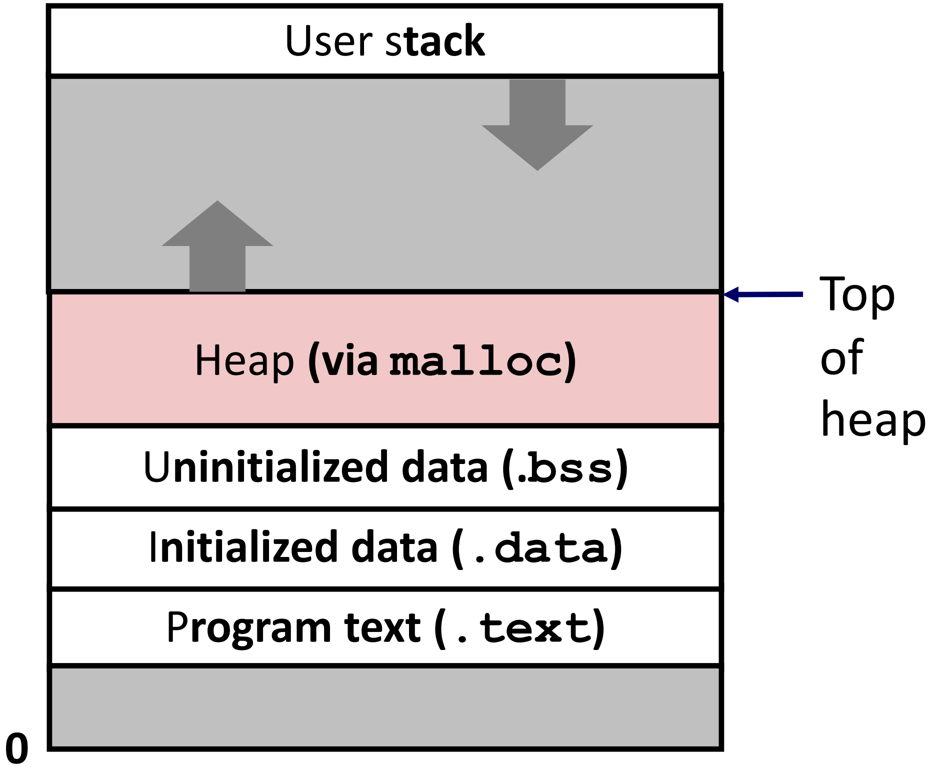
Dynamic Memory Allocation

Dynamic memory allocator

- Part of the process’s runtime system
  - Linked into program
- Manages the heap—within the process’s VM
  - May ask OS for additional heap space

Dynamic Memory Allocators

- **`malloc`** and **`free`** in C
- **`new`** and **`delete`** in C++
- Manage the *heap*, an area of process virtual memory
- For data structures whose size is only known at runtime.



Dynamic Memory Allocators

- Maintains the heap as collection of variable sized ***blocks***, which are either ***allocated*** or ***free***

- ***Explicit allocator***: application allocates and frees space
  - **`malloc`** and **`free`** in C; **`new`** and **`delete`** in C++
  - Discussed today

- ***Implicit allocator:*** application allocates, but does not free space
  - Garbage collection in Java, SML, and Lisp

Example using `malloc`

```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);
}
```

First Example: A Simple Allocator

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

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

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

Advantages
- Blazing fast
- Simple

Disadvantages
- Memory is never recycled
- No alignment

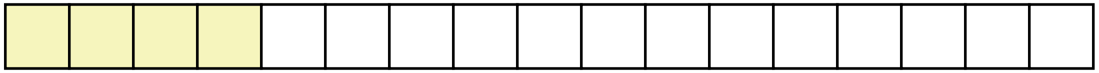
Desiderata
- Speed
- Alignment
- Efficient use of memory

Constraints

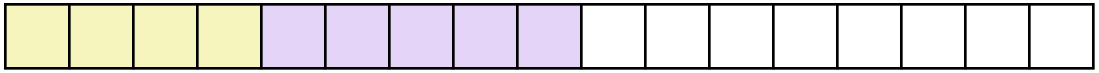
- Applications
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - `free` request must be to a `malloc`’d block
- Allocators
  - Cannot control number or size of allocated blocks
  - Must respond immediately to `malloc` requests
    - Cannot reorder or buffer requests
  - Must allocate blocks from free memory
  - Must align blocks so they satisfy alignment requirements
    - 8-byte (x86) or 16-byte (x86-64) alignment on Linux
  - Cannot move the allocated blocks once they are `malloc`’d
    - Compaction is not allowed

Allocation Example

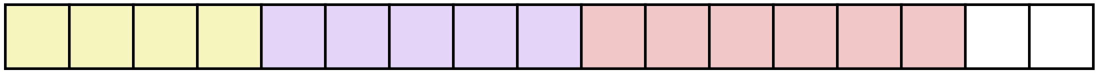
`p1 = malloc(4)`



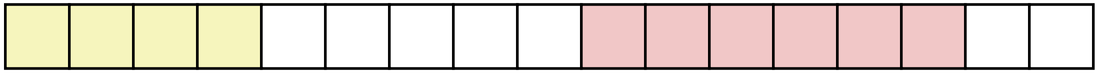
`p2 = malloc(5)`



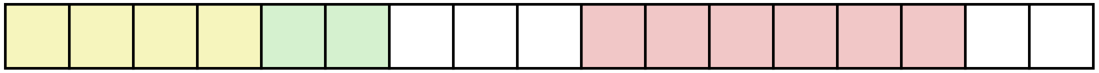
`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(2)`



Performance Goals

- **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

- Peak Memory Utilization
  - Minimize wasted space

Peak Memory Utilization

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

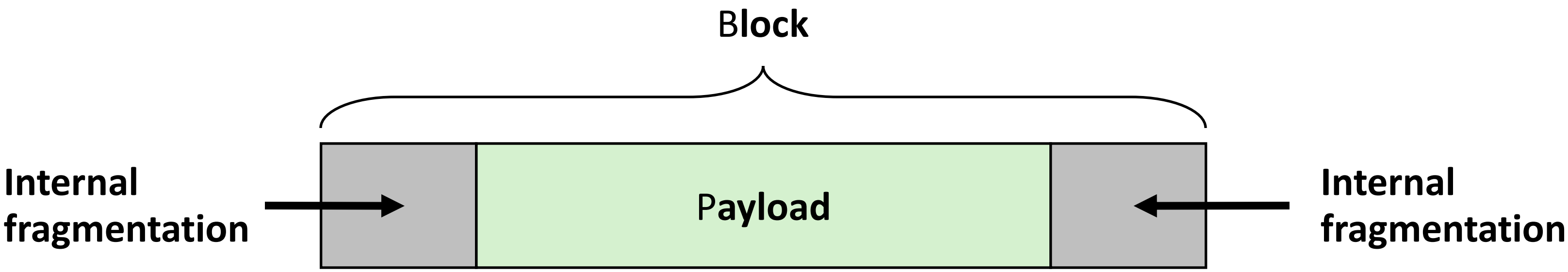
- *Def: Aggregate payload* P_k
  - **`malloc(p)`** results in a block with a ***payload*** of **`p`** bytes
  - After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads

- *Def: Current heap size* H_k
  - Assume H_k is monotonically nondecreasing

- *Def: Peak memory utilization after k+1 requests*
  - $U_k = (\max_{i<=k} P_i) \;/\; H_k$

Utilization Blocker: Internal Fragmentation

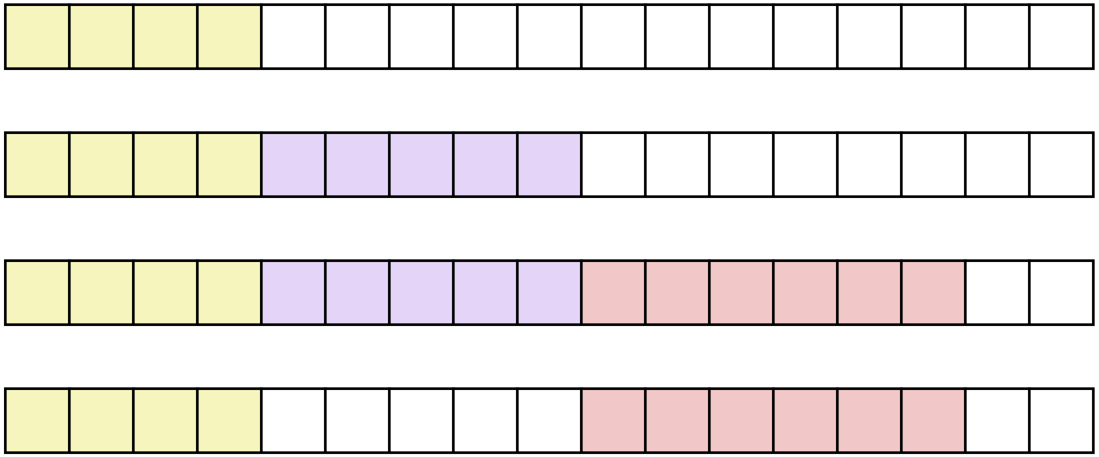
- For a given block, ***internal fragmentation*** occurs if payload is smaller than block size



- 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

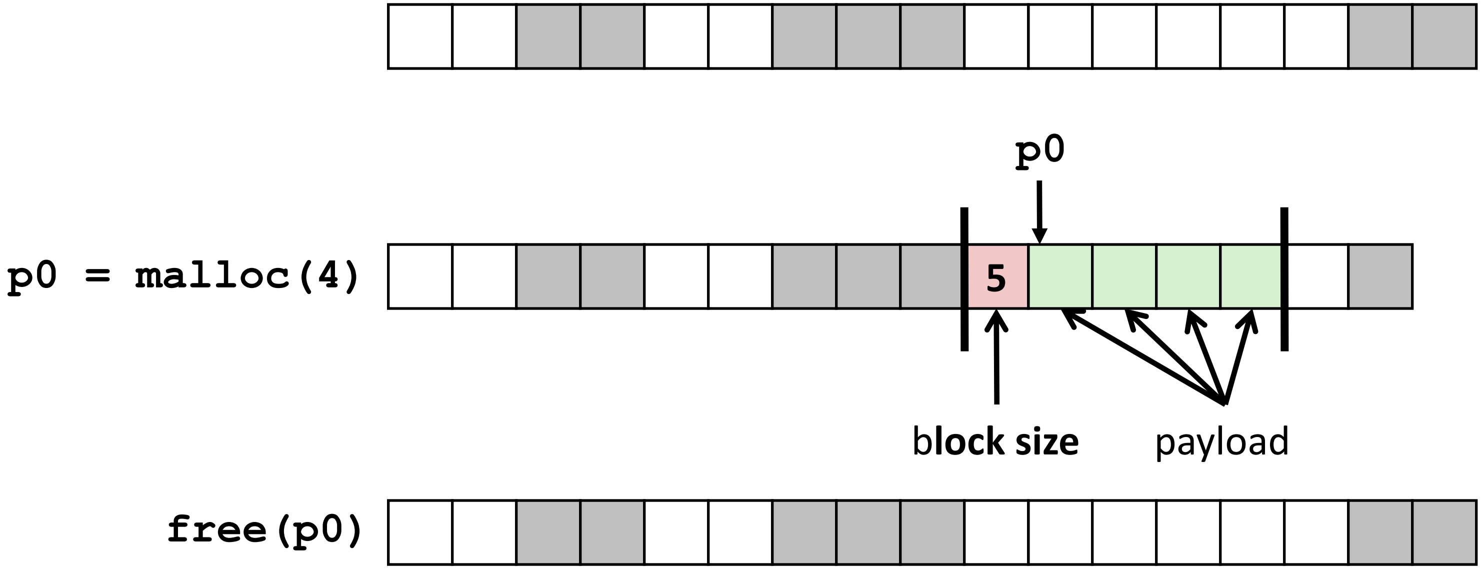
Challenges

- Strategic: maximize throughput and peak memory utilization

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

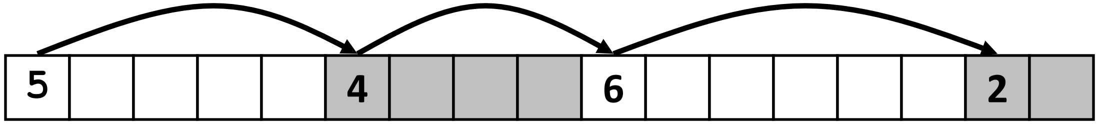
Knowing How Much to Free

- 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

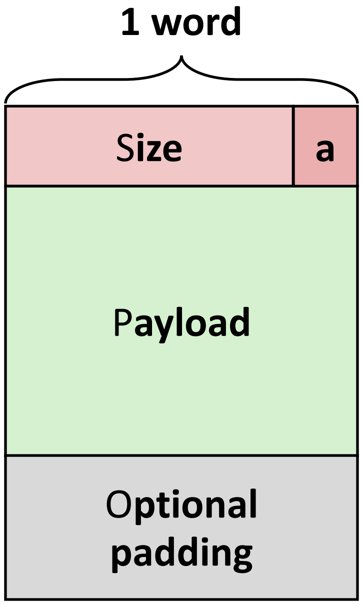
- Method 1: ***Implicit list*** using length—links all blocks



Method 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

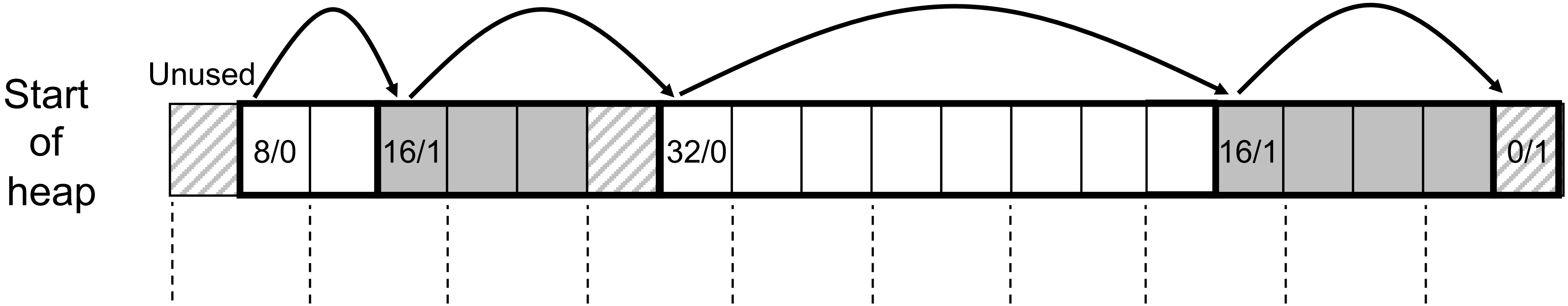


a = 1: Allocated block
a = 0: Free block

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

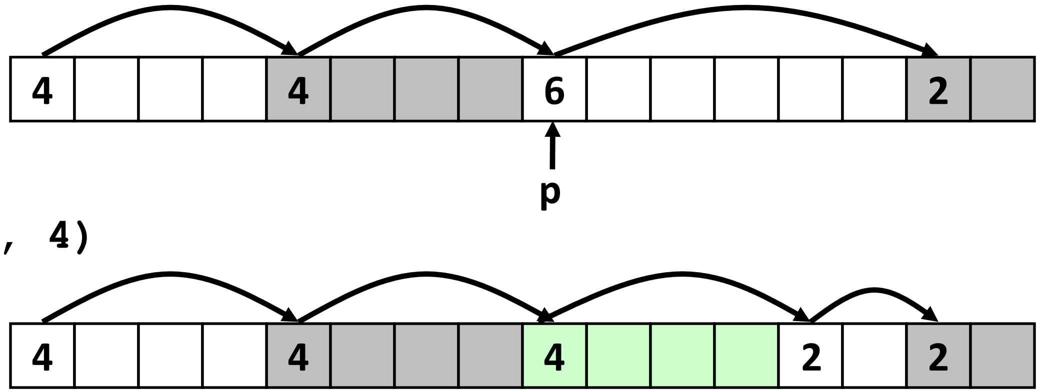
- ***First fit.*** Search list from beginning, choose ***first*** free block that fits:

```
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)`

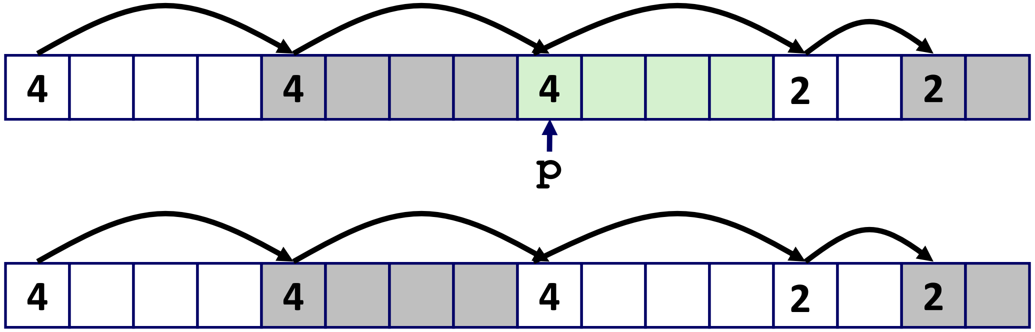
```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining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



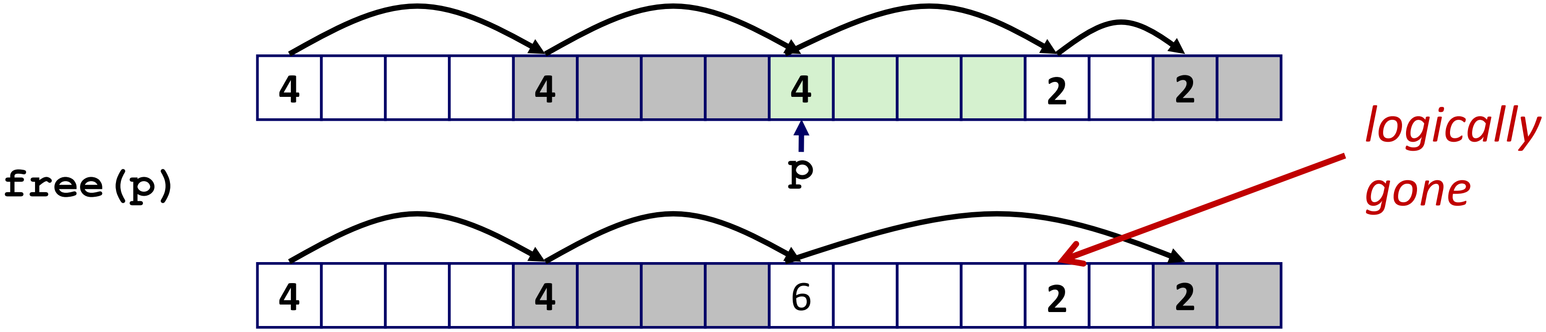
`free(p)`

`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

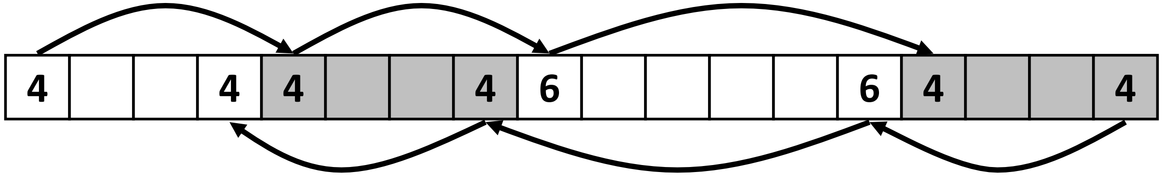


```
void free_block(ptr p) {
     *p = *p & -2;           // clear allocated flag
     next = p + *p;          // find next block
     if ((*next & 1) == 0)
       *p = *p + *next;      // add to this block if
}                            //    not allocated
```

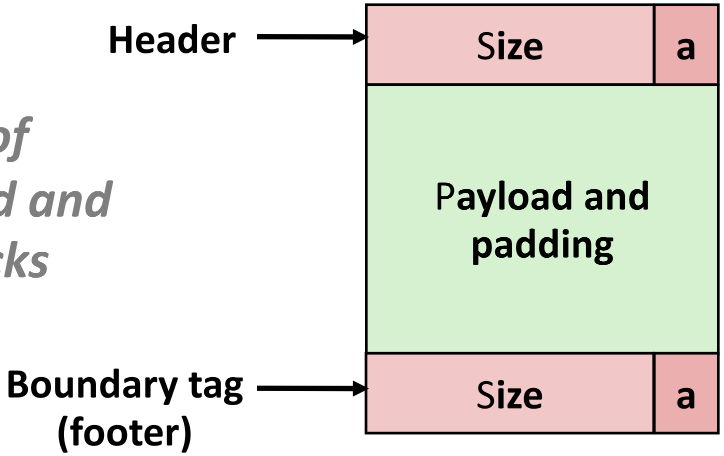
  - But how do we coalesce with *previous* block?

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



Header

Size | a

Payload and padding

Boundary tag (footer)

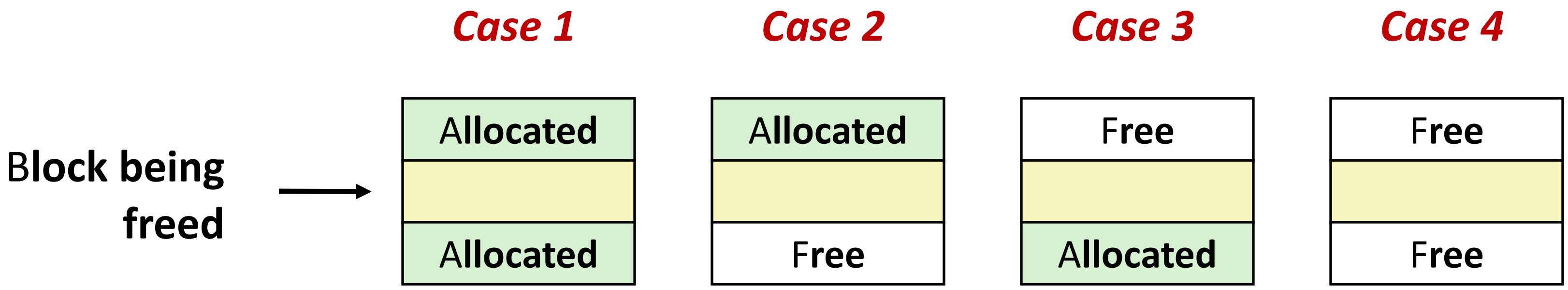
Size | a

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

Size: Total block size

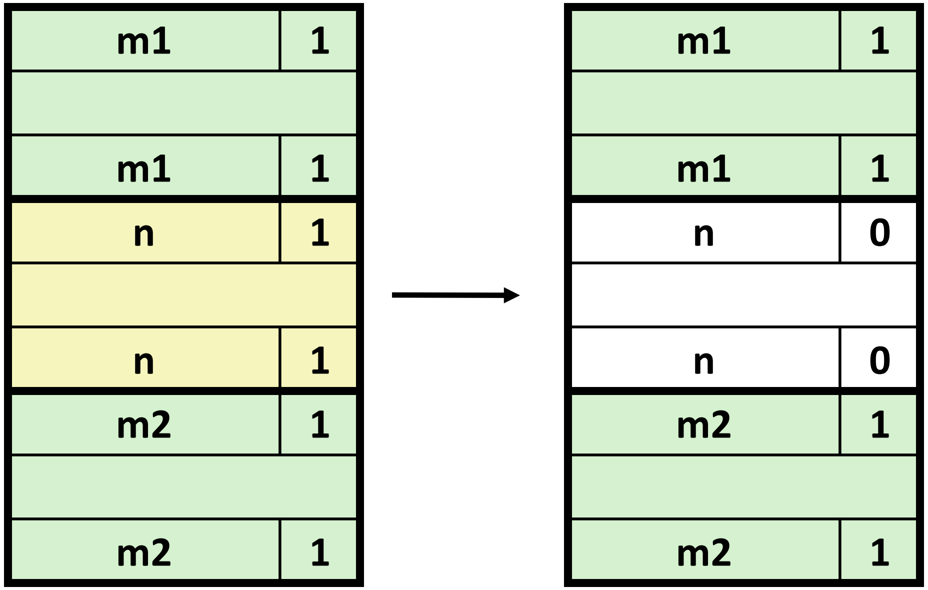
Payload: Application data (allocated blocks only)

Constant Time Coalescing

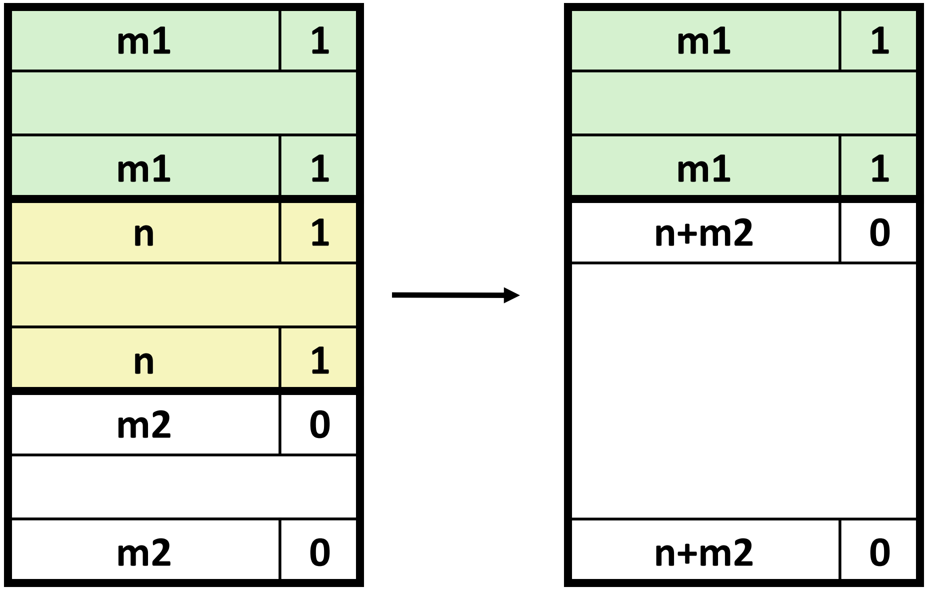


	Case 1	Case 2	Case 3	Case 4
	Allocated	Allocated	Free	Free
Block being freed →				
	Allocated	Free	Allocated	Free

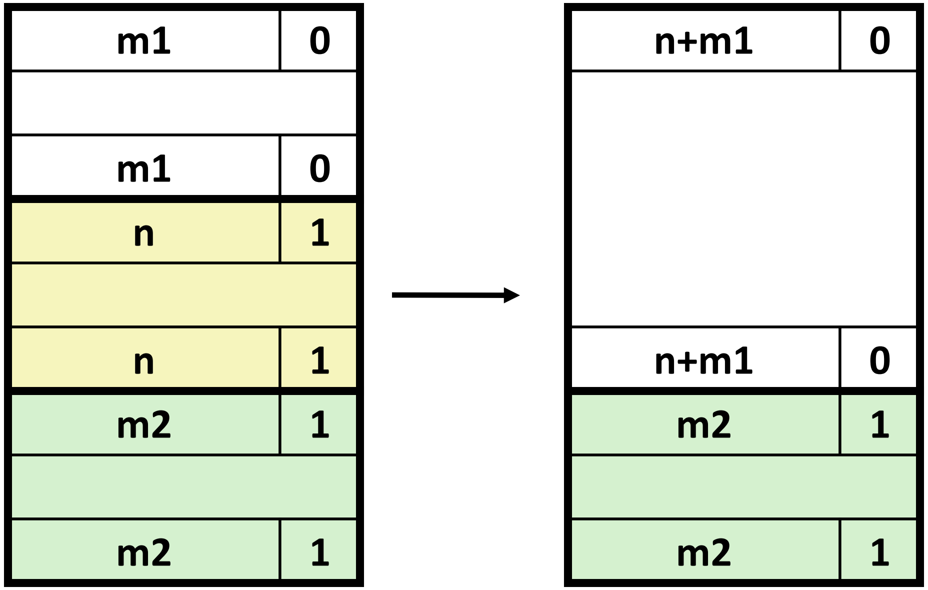
Constant Time Coalescing (Case 1)



Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



m1 | 0

m1 | 0

n | 1

n | 1

m2 | 1

m2 | 1

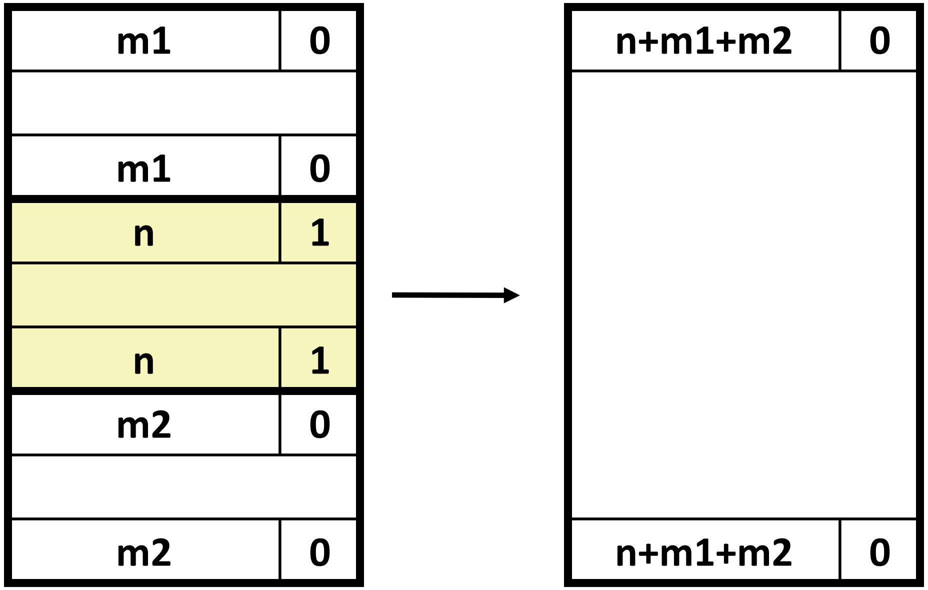
n+m1 | 0

n+m1 | 0

m2 | 1

m2 | 1

Constant Time Coalescing (Case 4)



Implicit Lists: Summary

- Implementation: very simple

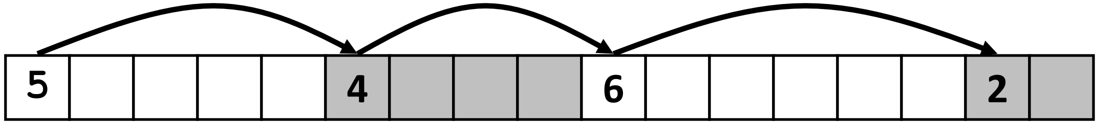
- Allocate cost: linear time in the worst case
- Free cost: constant time worst case–even with coalescing
- Memory usage: depends on the placement policy
  - First-fit, next-fit, or best-fit

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

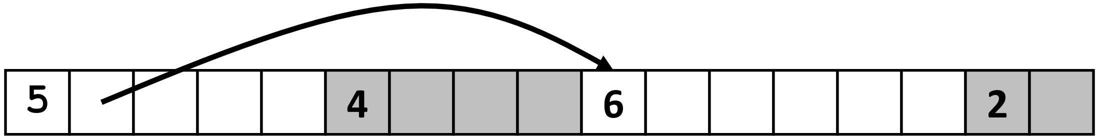
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators

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

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 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