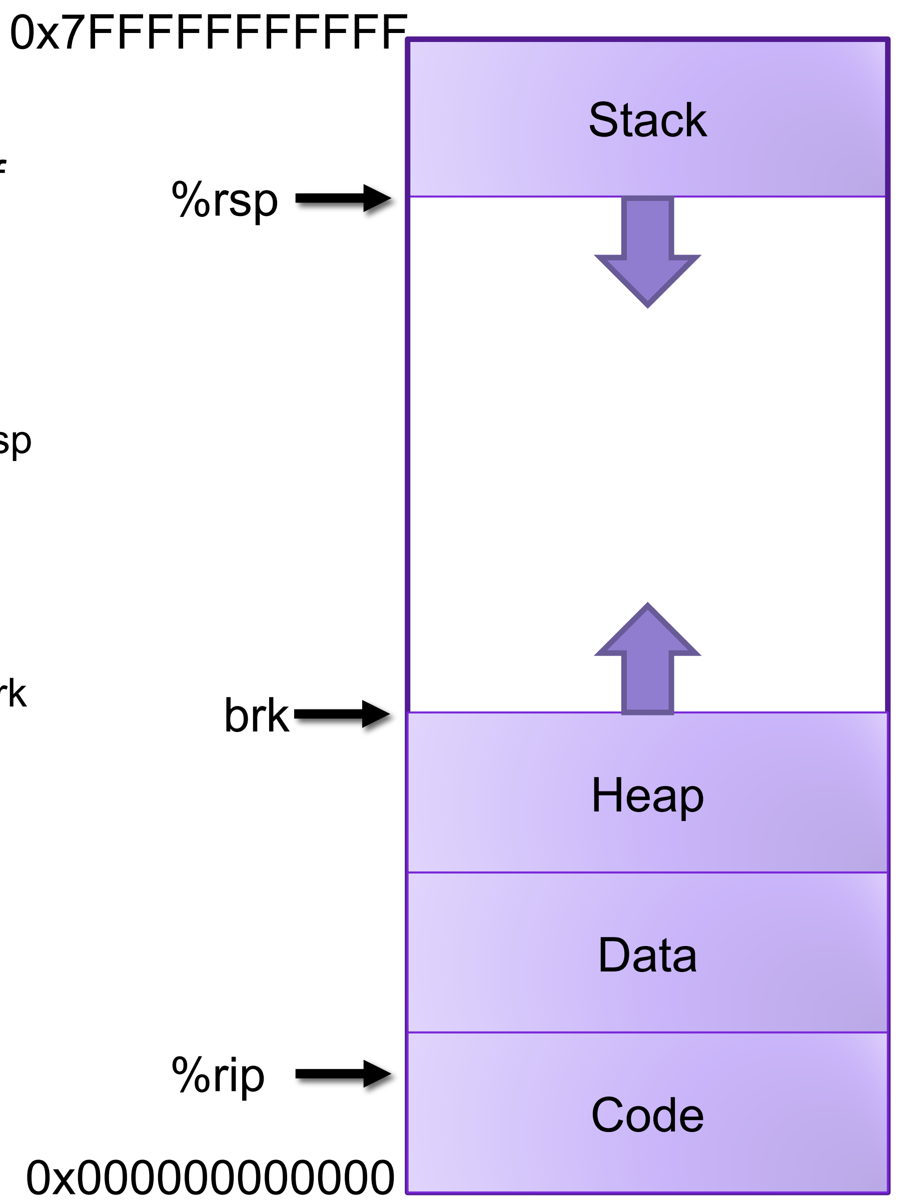
Lecture 11: Dynamic Memory

CS 105

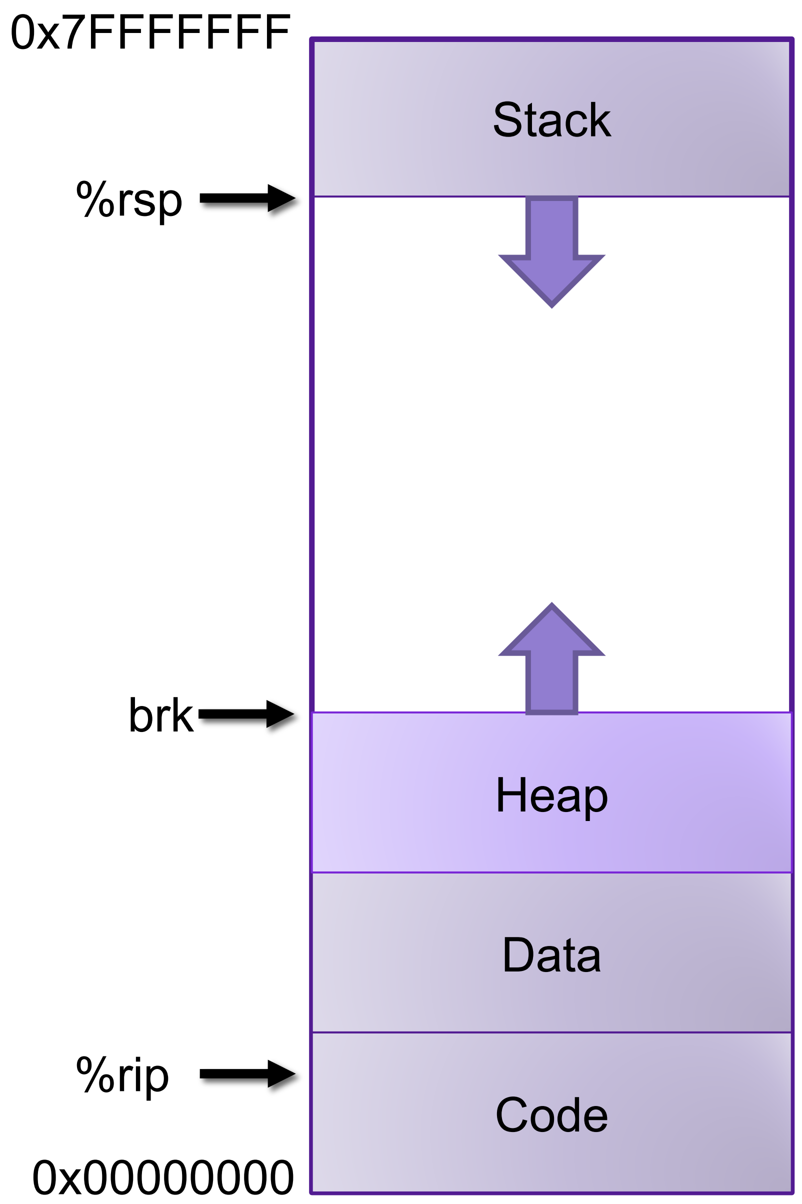
Memory

- byte addressable array made up of four logical segments
- **stack** provides local storage for procedures
  - "top" of the stack stored in register %rsp
- **heap** is an area of memory maintained by a dynamic memory allocator
  - operating system maintains variable brk that points to the top of heap
- **data** stores global variables
- **code** stores program instructions
- attempt to access uninitialized address results in exception (segfault)



The Heap

- the heap is an area of memory for dynamic memory allocation
- programmers can use a dynamic memory allocator to acquire additional memory at run time
- programmers can use a system call to modify brk (e.g., extend 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 using system call sbrk()
- 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

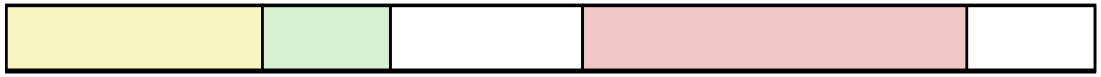
- object creation & garbage collection in Java
- object creation & garbage collection in Python

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

1) **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)

2) **Must respond immediately:**
   - no reordering or buffering requests

3) **Must not modify allocated blocks:**
   - can only allocate from free memory on the heap
   - cannot modify or move blocks once they are allocated

4) **Must align blocks:**
   - 8-byte (x86) or 16-byte (x86-64) alignment on Linux
   - Ensures that allocated blocks can hold any type of data

5) **Must only use the heap:**
   - any data structures used by the allocator must be stored in the heap

First Example: A Simple Allocator

```
void *malloc (size_t size) {
  return sbrk(align(size));
}

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

Advantages
- Simple
- Blazing fast

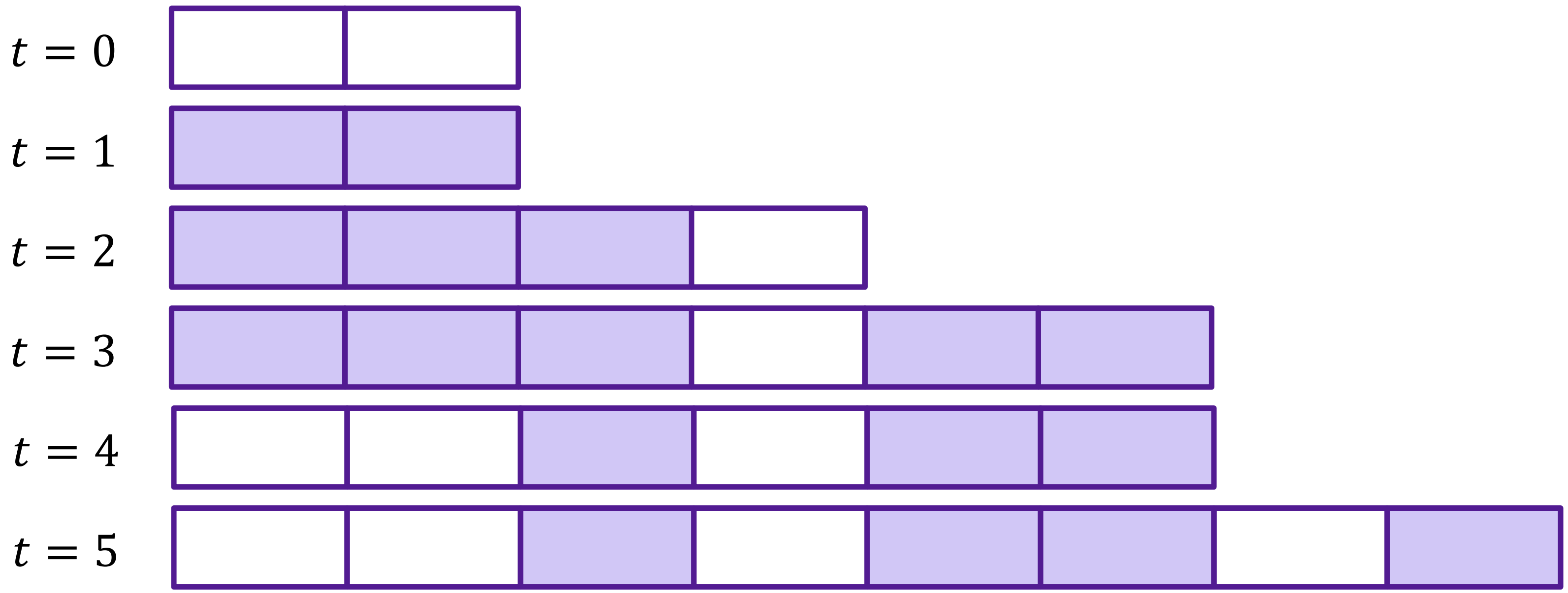
Disadvantages
- Memory is never recycled
- Wastes a lot of space

Allocator Goals

- **Throughput:** number of requests completed per time unit
  - Make allocator efficient
  - Example: if your allocator processes 5,000 **`malloc`** calls and 5,000 **`free`** calls in 10 seconds then throughput is 1,000 operations/second
- **Memory Utilization:** fraction of heap memory allocated
  - Minimize wasted space
  - Peak Memory Utilization $U_t = \dfrac{\max\limits_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$

Exercise 0: Memory Utilization

- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} \textit{space allocated at time } i}{\textit{size of heap at time } t}$



- What is the Peak Memory Utilization at time $t = 2$? $\mathbf{3/4}$
- What is the Peak Memory Utilization at time $t = 5$? $\mathbf{5/8}$

Allocator Goals

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

- **Memory Utilization:** fraction of heap memory allocated
  - Minimize wasted space
  - Peak Memory Utilization $U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$

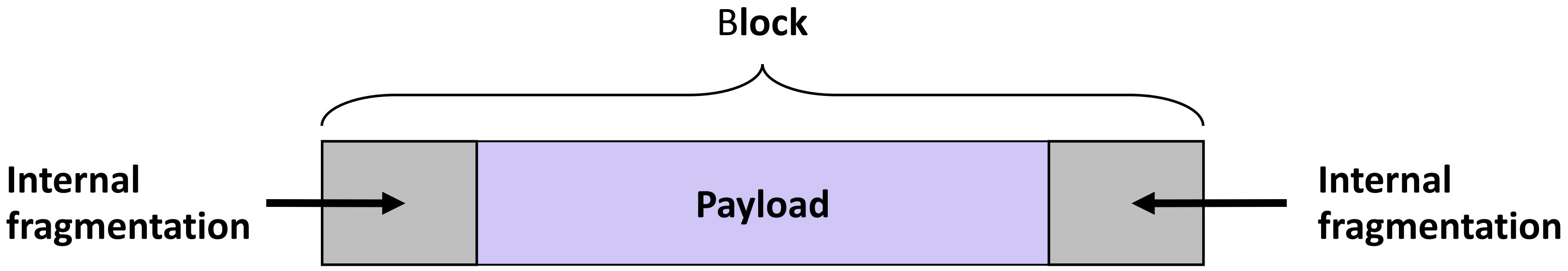
- These goals are often conflicting

Allocator Goals

- **Throughput:** number of requests completed per time unit
  - Make allocator efficient
  - Example: if your allocator processes 5,000 **`malloc`** calls and 5,000 **`free`** calls in 10 seconds then throughput is 1,000 operations/second
- **Memory Utilization:** fraction of heap memory allocated
  - Minimize wasted space
  - Peak Memory Utilization $U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$
- These goals are often conflicting

Utilization Blocker: Internal Fragmentation

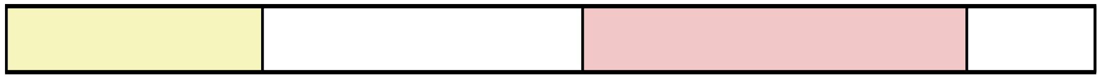
- For a given block, ***internal fragmentation*** occurs if payload is smaller than block size



- 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)
```

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

Exercise 1: Utilization

Assume your heap is initially of size zero and you then run the following sequence of requests (below left) using the given allocator (below right) on a system with 4-byte alignment. What is the peak memory utilization after you complete the last request?

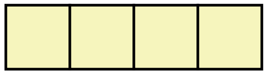
Hint: Peak Memory Utilization $U_t = \dfrac{\max_{i \le t} \mathit{space\ allocated\ at\ time\ i}}{\mathit{size\ of\ heap\ at\ time\ t}}$

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```

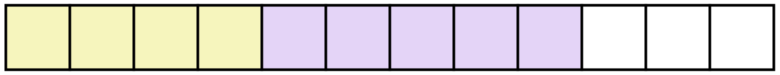
```
void *malloc (size_t size) {
  return sbrk(align(size));
}

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

Exercise 1: Utilization

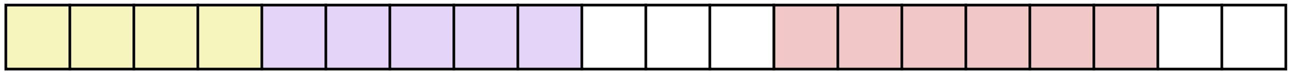
```
p1 = malloc(4)
```



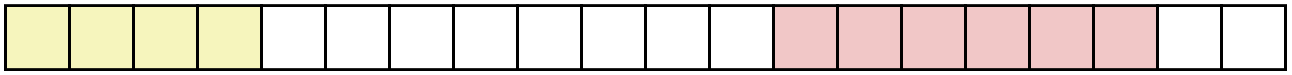
```
p2 = malloc(5)
```



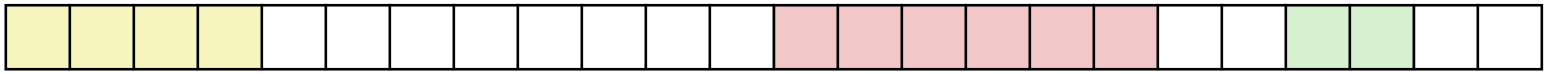
```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(2)
```



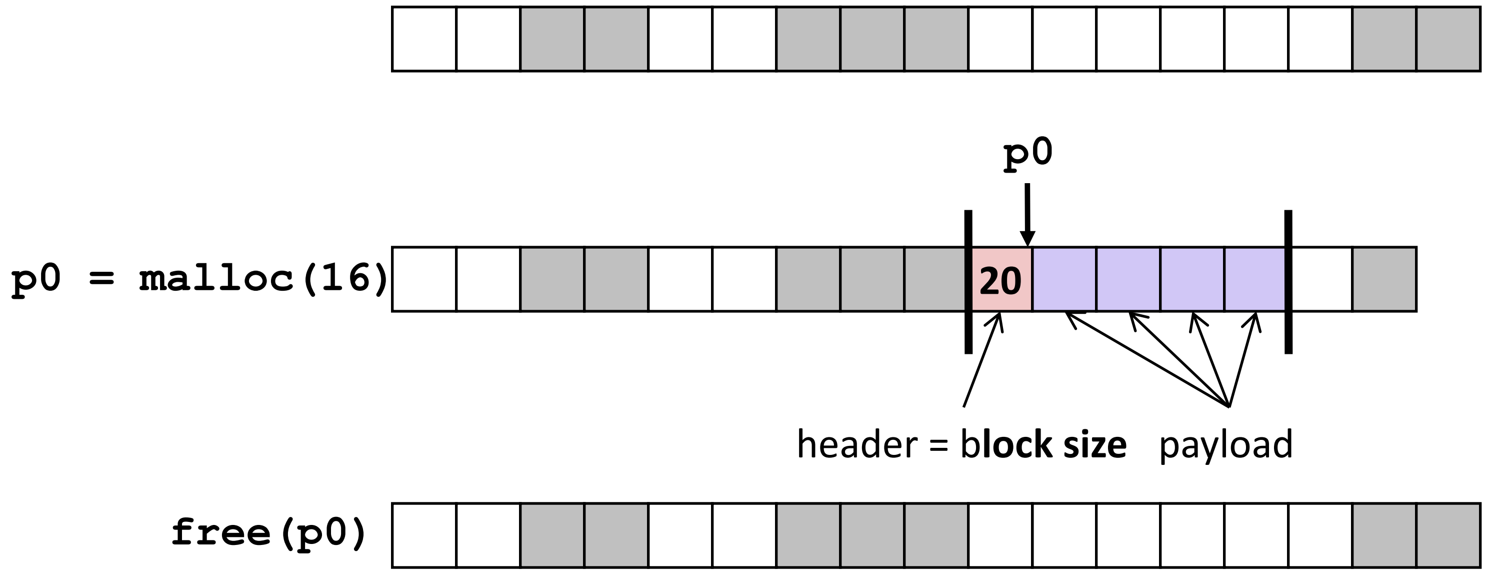
Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation challenges:
  - 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?
  - 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?

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



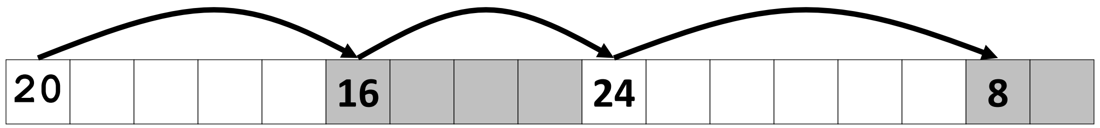
Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation Challenges:
  - 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?
  - 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?

Keeping Track of Free Blocks

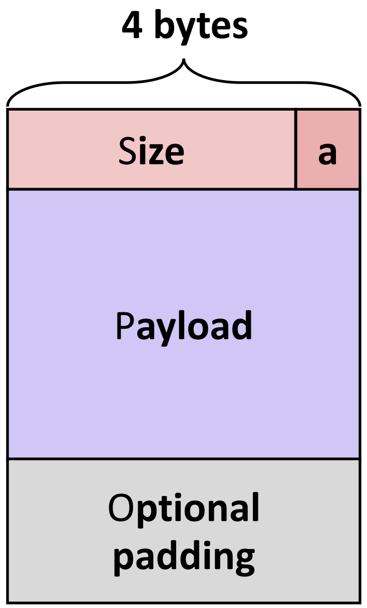
- Method 1: ***Implicit list*** using length—links all blocks



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

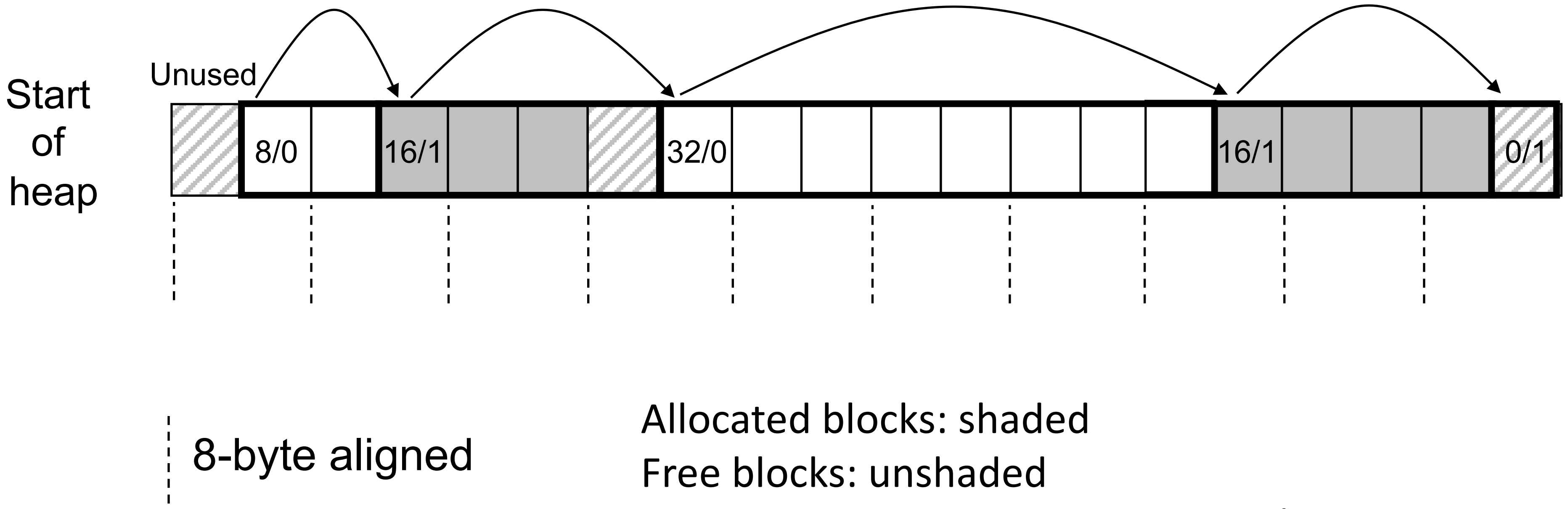


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example

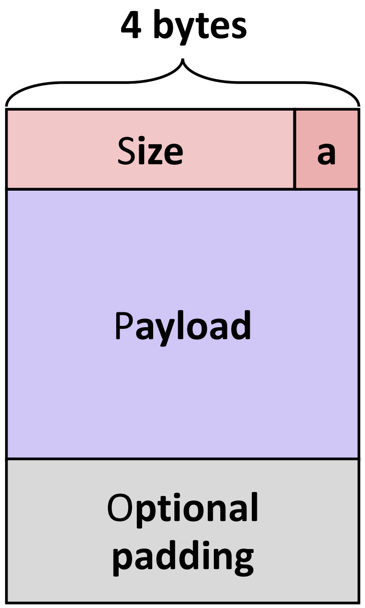


Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

Exercise 2: Block Headers

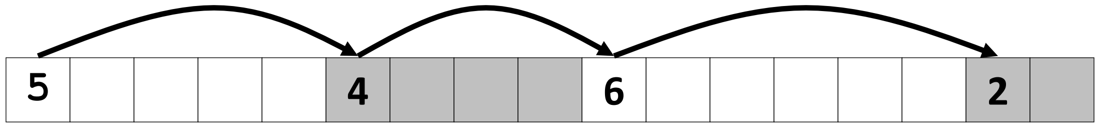
- Determine the block sizes and header values that would result from the following sequence of malloc requests. Assume that the allocator uses an implicit list implementation with the block format just described and maintains 8-byte alignment.

Request	Block size (decimal)	Block header (hex)
malloc(1)	8	0x00000009
malloc(5)	16	0x00000011
malloc(12)	16	0x00000011

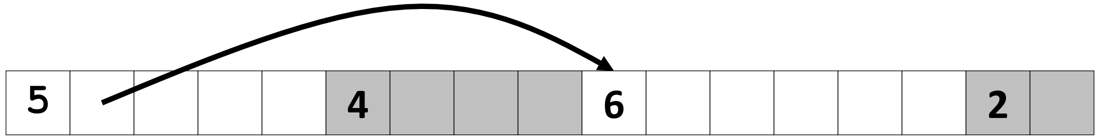


Keeping Track of Free Blocks

- 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

Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation Challenges:
  - 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?
  - 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: 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

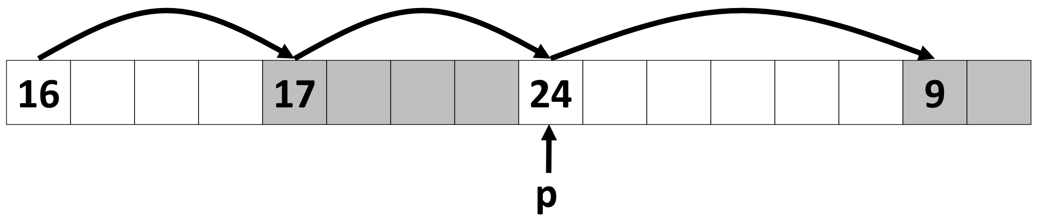
Challenges

- Goal: maximize throughput and peak memory utilization

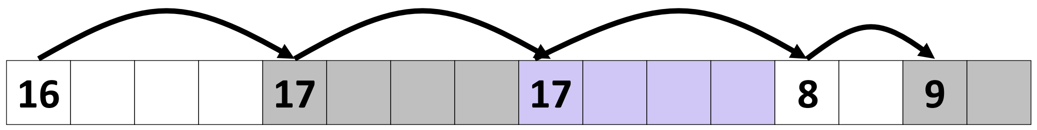
- Implementation Challenges:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation?
  - What do we do with the extra space when allocating a structure that is smaller than the free block it is placed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

Implicit List: 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
```

Summary of Key Allocator Policies

- Free-block storage policy:
  - Implicit lists, with boundary tags (nice and simple)
  - Explicit lists, exclude free blocks (faster, but more overhead)
  - Segregated lists (different lists for different sized blocks)
  - Fancy data structures (red-black trees, for example)

- Placement policy:
  - First-fit (simple, but lower throughput and higher fragmentation)
  - Next-fit (higher throughput, higher fragmentation)
  - Best-fit (lower throughput, lower 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?

Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation Challenges:
  - 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?
  - 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?

Exercise 4: Feedback

1. Rate how well you think this recorded lecture worked
    1. Better than an in-person class
    2. About as well as an in-person class
    3. Less well than an in-person class, but you still learned something
    4. Total waste of time, you didn't learn anything

2. How much time did you spend on this video (including exercises)?

3. Do you have any particular questions you'd like me to address in this week's problem session?

4. Do you have any other comments or feedback?