Lecture 15: `malloc()` under the hood

CSE 29: Systems Programming and Software Tools

Olivia Weng

Review: Pointer arithmetic

- General rule: `ptr + n` = `ptr + n * sizeof(type)`

In C

C interprets it like this

```
char str[] = "Hi CSE29!";
char str1[] = "30?";

strncpy(str + 6, str1, strlen(str1));

printf("%s\n", str);
```

123456

3

↳ Hi CS630?

```
int arr[] = {0, 1, 2};
int *ptr = arr; // 0x100
```

ptr + 2

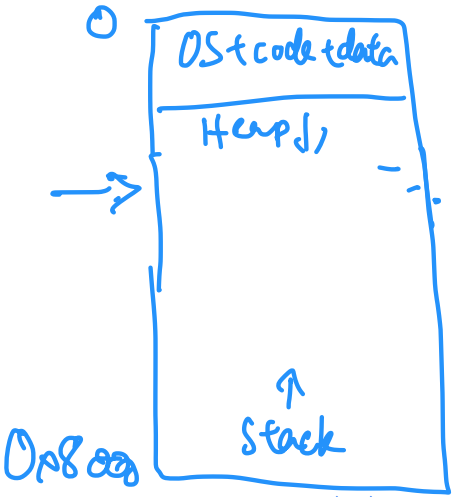
0x 108

?

Announcements

- Sign up for Exam 3 on prairietest.com
  - Can even sign up for makeup!

- Problem set 4 will be released today



How do `malloc()` and `free()` work?

What problem are `malloc()` and `free()` trying to solve?

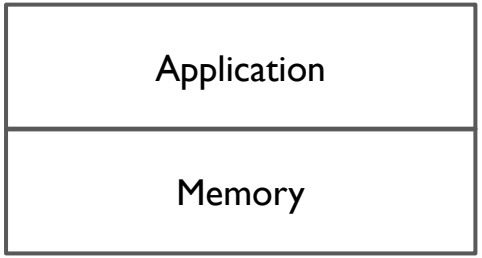
malloc()

- Solves: allocating memory of any size for data that exists longer than a function call

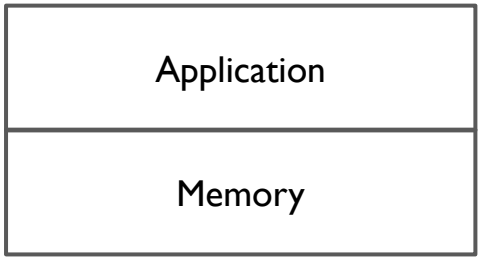
- Why?
  - So we can access the data as long as we need it, beyond the function it was created in

Memory Allocation

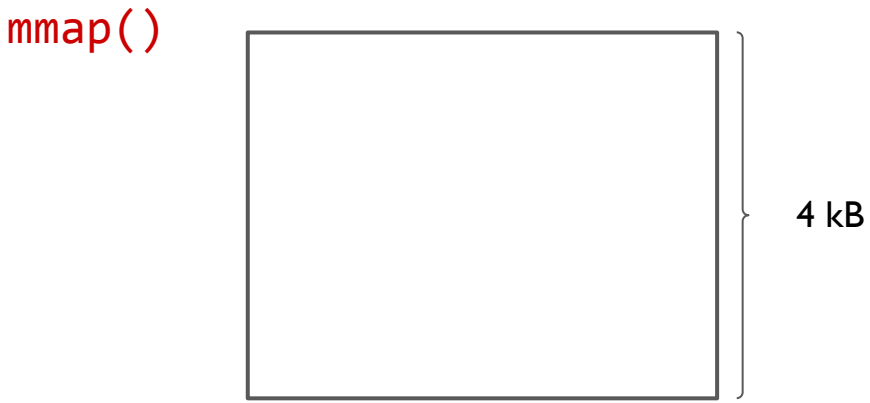
- Application wants memory for its heap



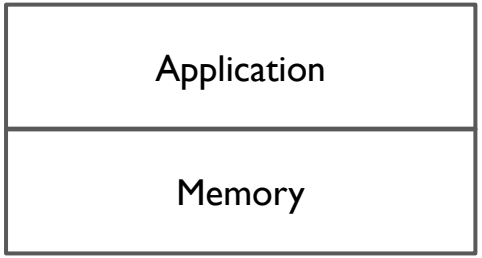
Memory Allocation



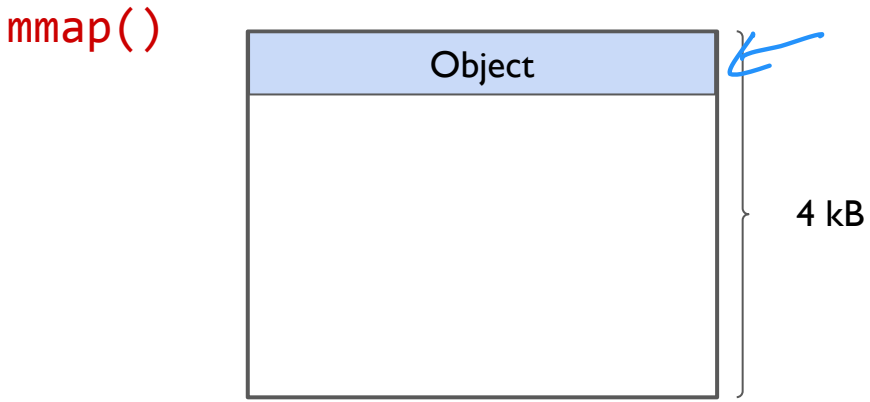
- Application wants memory for its heap
  - OS provides chunks of memory via `mmap()`



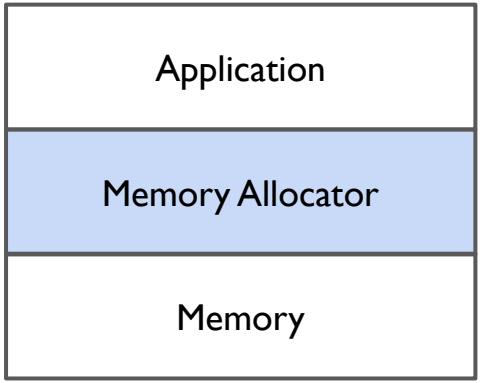
Memory Allocation



- Application wants memory for its heap
  - OS provides chunks of memory via `mmap()`
  - But, application objects are typically smaller than these chunks

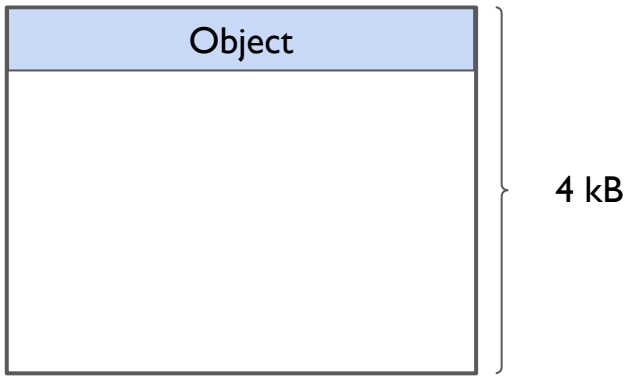


Memory Allocation

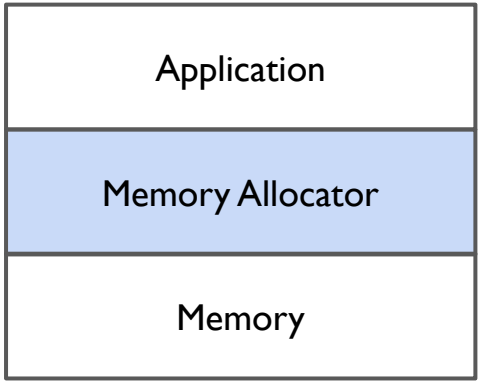


- Application wants memory for its heap
  - OS provides chunks of memory via `mmap()`
  - But, application objects are typically smaller than these chunks
- Memory allocator: manages objects within these chunks

`mmap()`

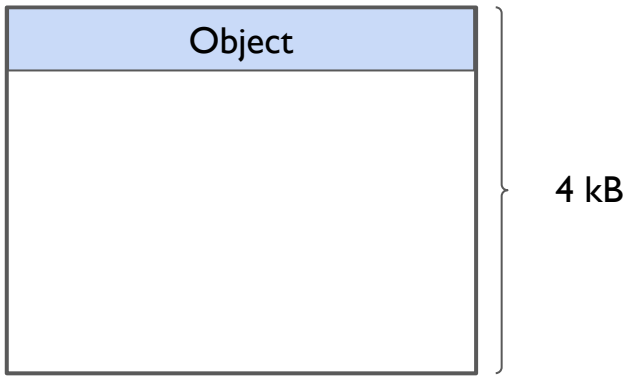


Memory Allocation



- Application wants memory for its heap
  - OS provides chunks of memory via `mmap()`
  - But, application objects are typically smaller than these chunks
- Memory allocator: manages objects within these chunks

`mmap()`



`malloc()` decides:

- Where to place object
- How to manage free memory
- When to call `mmap()`
- And more...

`malloc()` Requirements

- `#include <stdlib.h>`

- `void *malloc(size_t size)`
  - Returns pointer to memory of at least `size` bytes, *aligned to (typically, on Linux) 8 bytes*

- `void free(void *ptr)`
  - Returns the memory pointed to by `ptr` to pool of free memory space
  - `ptr` must have come from a previous call to `malloc()`

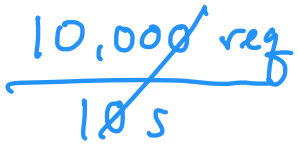
Performance Goals: `malloc()` & `free()`

Performance Goals: `malloc()` & `free()`

- Maximize **Throughput**
  - **Throughput**: Number of completed requests per unit time

Performance Goals: `malloc()` & `free()`

- Maximize **Throughput**
  - **Throughput**: Number of completed requests per unit time
  - Ex:
    - 5,000 `malloc()` calls and 5,000 `free()` calls in 10 seconds
    - Throughput = 1,000 memory operations / second



Performance Goals: `malloc()` & `free()`

- Maximize **Throughput**
  - **Throughput**: Number of completed requests per unit time
  - Ex:
    - 5,000 `malloc()` calls and 5,000 `free()` calls in 10 seconds
    - Throughput = 1,000 memory operations / second

- Maximize **Peak Memory Utilization**
  - **Memory Utilization**: Size of currently allocated memory / Size of all requested `mmap()` memory

Performance Goals: `malloc()` & `free()`

- Maximize **Throughput**
  - **Throughput**: Number of completed requests per unit time
  - Ex:
    - 5,000 `malloc()` calls and 5,000 `free()` calls in 10 seconds
    - Throughput = 1,000 memory operations / second

- Maximize **Peak Memory Utilization**
  - **Memory Utilization**: Size of currently allocated memory / Size of all requested `mmap()` memory
  - **Peak Memory Utilization**: How well you are using the memory you've requested
    - Highest utilization possible = 1

Poor memory utilization

- Fragmentation: When free chunks of memory are only available in small fragments
  - This is bad because it becomes difficult to allocate big contiguous chunks of memory

Good memory utilization

- Not a lot of fragmentation

How to maximize throughput?

- Throughput = # operations / second

- Minimize average time it takes to complete an operation, either `malloc()` or `free()`

How to maximize memory utilization?

- Memory utilization =
  - Size of currently allocated memory / Size of requested `mmap()` memory

How to maximize memory utilization?

- Memory utilization =
  - Size of currently allocated memory / Size of requested `mmap()` memory

- Use up as much of the memory that you already have

How to maximize memory utilization?

- Memory utilization =
  - Size of currently allocated memory / Size of requested `mmap()` memory

- Use up as much of the memory that you already have

- Consider:
  - Can I use/reuse free memory or must I ask the OS for more memory via `mmap()`?

Maximizing throughput often conflicts with maximizing memory utilization

Implementation Issues

- How to know how much memory is being `free()`'d when we're only given a pointer (and no length)?
- How to keep track of free memory blocks?
- How to pick which free memory chunks to use for allocation?
  - Many viable options
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?

Implementation Issues

- How to know how much memory is being `free()`'d when we're only given a pointer (and no length)?
- How to keep track of free memory blocks?
- How to pick which free memory chunks to use for allocation?
  - Many viable options
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?

Assumptions made in this lecture

- Memory is **4-byte** addressed and **4-byte aligned**
- Allocated bytes make up an **allocated block** of memory
- Free bytes make up a **free block** of memory

Knowing How Much to Free

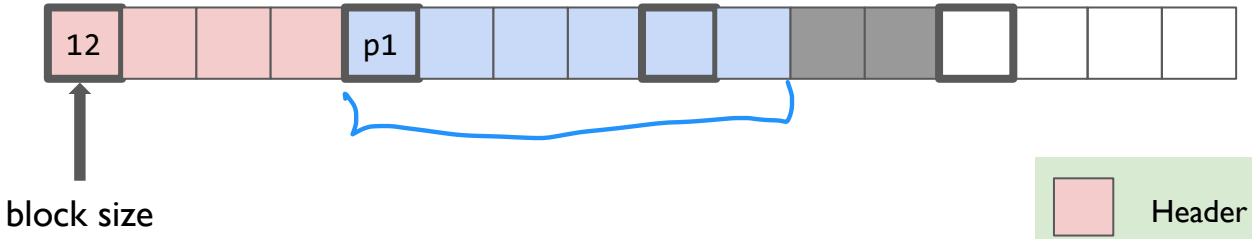
- Keep the length of allocated memory in the preceding aligned block
  - This is often called the ***header field*** or ***header***

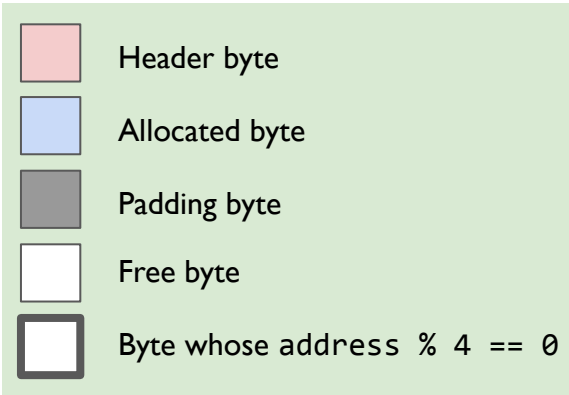
Knowing How Much to Free

- Keep the length of allocated memory in the preceding aligned block
  - This is often called the ***header field*** or ***header***
  - Since we are 4-byte addressed, we require 4 extra bytes to store header for every allocated block

Knowing How Much to Free

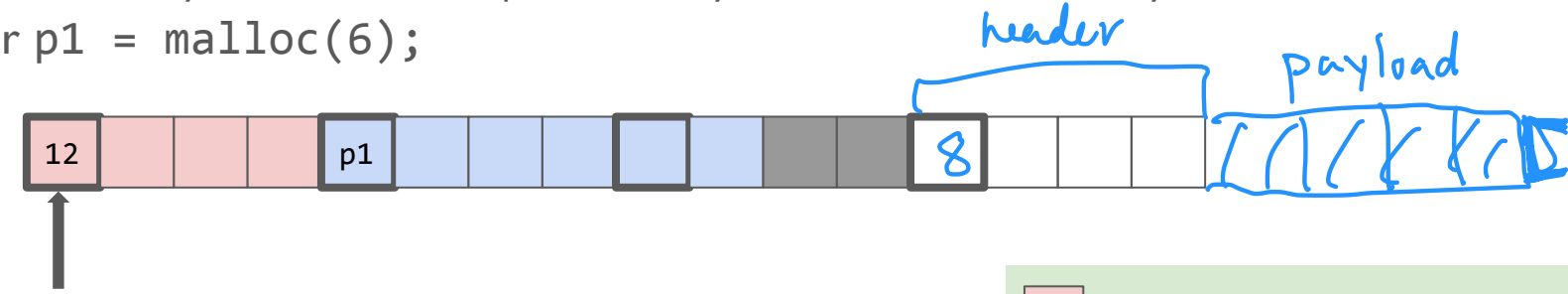
- Keep the length of allocated memory in the preceding aligned block
  - This is often called the ***header field*** or ***header***
  - Since we are 4-byte addressed, we require 4 extra bytes to store header for every allocated block
- Consider `p1 = malloc(6);`





Knowing How Much to Free

- Keep the length of allocated memory in the preceding aligned block
  - This is often called the ***header field*** or ***header***
  - Since we are 4-byte addressed, we require 4 extra bytes to store header for every allocated block
- Consider `p1 = malloc(6);`



block size = 4 byte header
+ 6 byte object
+ 2 bytes padding

12 bytes allocated total

Header byte

Allocated byte

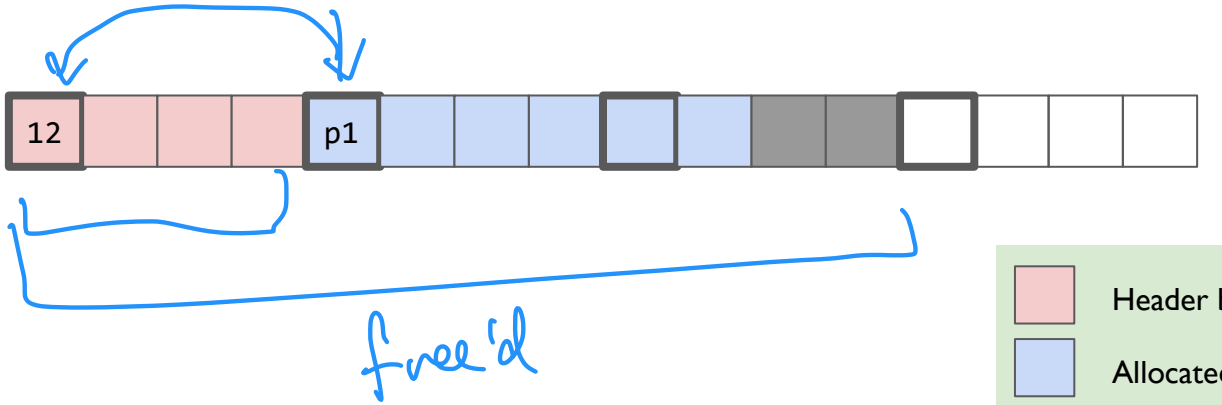
Padding byte

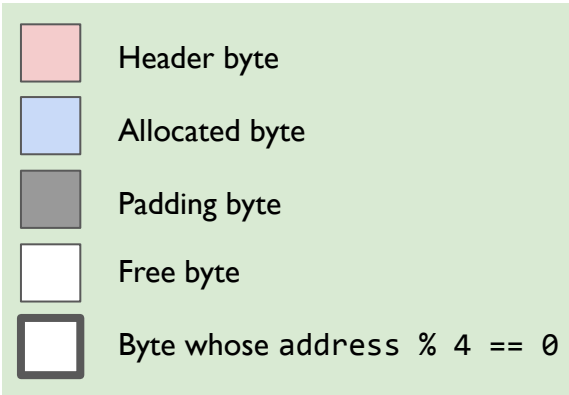
Free byte

Byte whose address % 4 == 0

Knowing How Much to Free

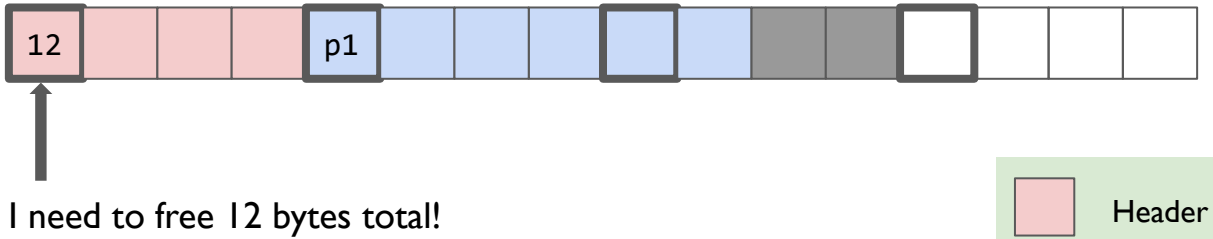
- To `free(p1)`
  - Check ***header*** in preceding aligned block to know how much memory to free





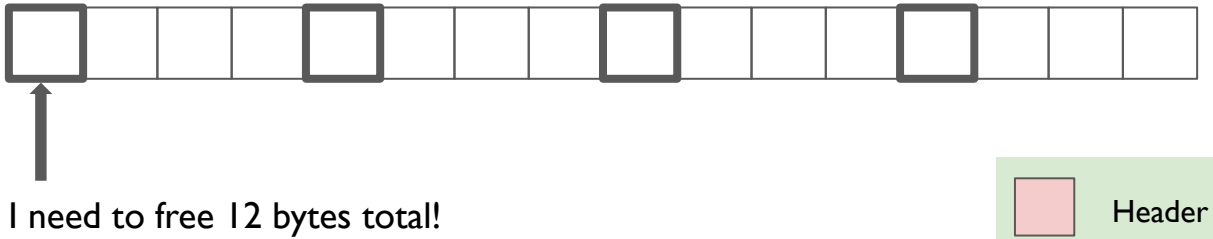
Knowing How Much to Free

- To `free(p1)`
  - Check ***header*** in preceding aligned block to know how much memory to free



Knowing How Much to Free

- To `free(p1)`
  - Check ***header*** in preceding aligned block to know how much memory to free



Implementation Issues

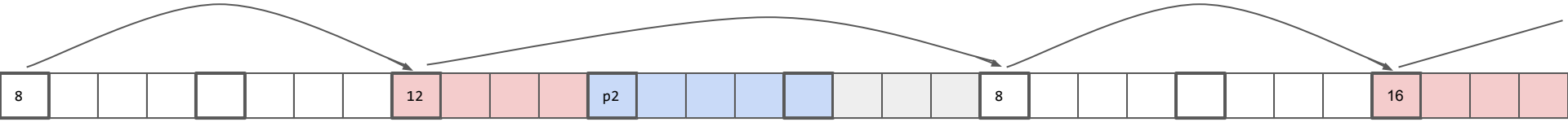
- How to know how much memory is being `free()`'d when we're only given a pointer (and no length)? **Use headers**
- How to keep track of free memory blocks?
- How to pick which free memory chunks to use for allocation?
  - Many viable options
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?

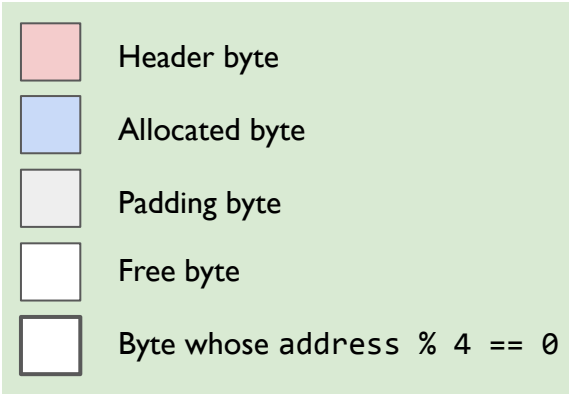
Implementation Issues

- How to know how much memory is being `free()`'d when we're only given a pointer (and no length)? **Use headers**
- How to keep track of free memory blocks?
- How to pick which free memory chunks to use for allocation?
  - Many viable options
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?

Keeping Track of Free Blocks

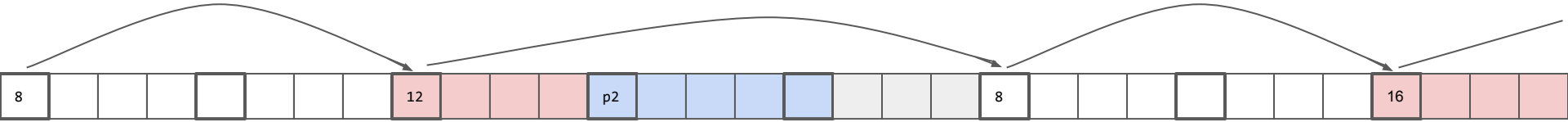
- Use pointer arithmetic to traverse the heap and find free blocks





Implicit List

- "Implicit List" == "Implicit Linked List"
  - Use the *lengths* to traverse the memory blocks via *pointer arithmetic*



Header byte

Allocated byte

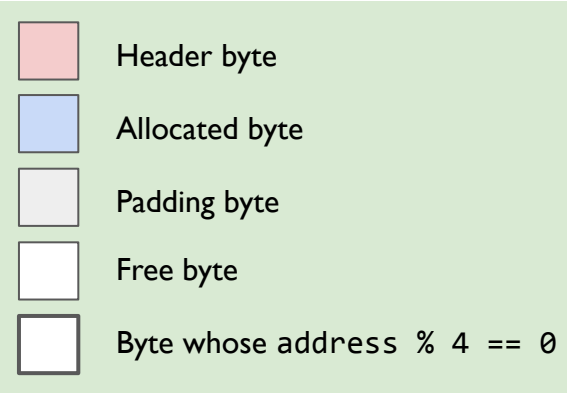
Padding byte

Free byte

Byte whose address % 4 == 0

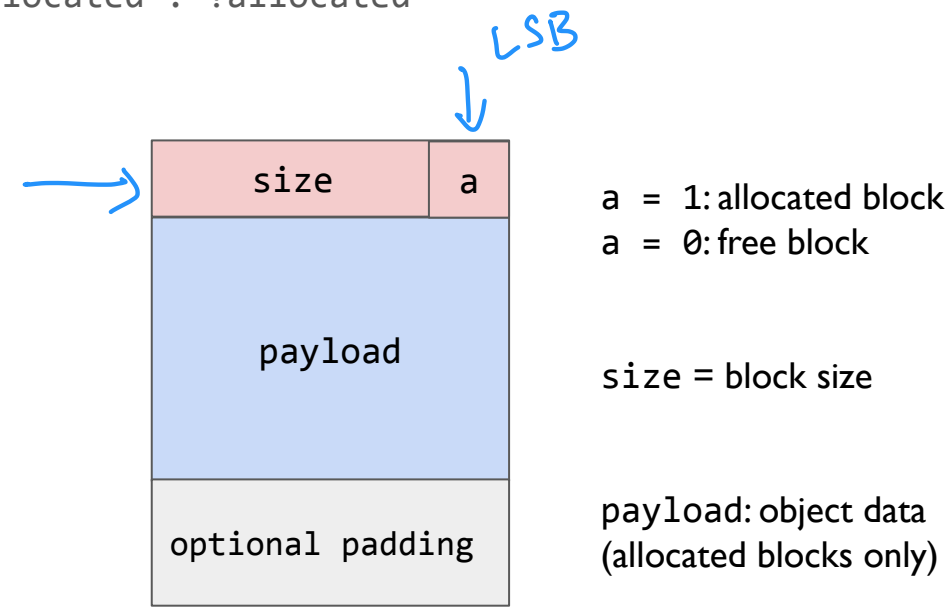
Implicit List

- For each memory allocation, we need:
  - Length
  - Is-allocated?
- Could store this information in the preceding 4-byte aligned block
  - This is wasteful!



Implicit List

- Standard trick
  - Since memory is 4-byte aligned, the 2 lowest-order address bits are always 0
  - `LSB == 1 ? allocated : !allocated`



What is the block size and allocated status?

- Header = 0xC1

192 = block size
1 = is allocated

1100 0001 = 193

1100 0000 = block size

0xC0 => 192 = blocksize
0 = not allocated / free

What is the block size and allocated status?

- Header = 0xC1
  - Block size = 192
  - allocated: 1

How can I get the size in C?

- Header = 0xC1 => 0x000000C1
  - Assume 64-bit addresses

block

E = 1110

& (~1)

32

& 0xFFFFFFFE

0x000000C0 = block size

How can I get the size in C?

- Header = 0xC1 => 0x000000C1
  - Assume 64-bit addresses

- Bit masking!
  - Mask = 0xFFFFFFFE or (~0x1)

How can I get the is-allocated status in C?

- Header = 0xC1 => 0x000000C1
  - Assume 64-bit addresses

& 0x1 = bit masking

& 0x0000001

0x0000001

Implementation Issues

- How to know how much memory is being `free()`'d when we're only given a pointer (and no length)? **Use headers**
- How to keep track of free memory blocks? **Implicit list + is-allocated bit**
- How to pick which free memory chunks to use for allocation?
  - Many viable options
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?

Implementation Issues

- How to know how much memory is being `free()`'d when we're only given a pointer (and no length)? **Use headers**
- How to keep track of free memory blocks? **Implicit list + is-allocated bit**
- How to pick which free memory chunks to use for allocation?
  - Many viable options
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?

Implicit List: Finding a Free Block

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