Project 2: Memory Allocator

Shinwoo Kim
Teaching Assistant
shinwookim@pitt.edu
https://sites.pitt.edu/~shk148/CS0449-2234/

- *Course News!*
- *Malloc Project*
- *TA-aided Debugging*

Spring 2023, Term 2234
Friday 12 PM Recitation
5502 Sennott Square
Feb 17th, 2023

Announcements

- ~~Queue Lab~~
  - ~~Due: Thursday, February 23 2023 11:59 PM~~
- Project 2: Heap Allocator
  - Due: Friday, March 3rd 2023, 11:59 PM

The Malloc Project.

- For Project 2, you will implement your own version of `malloc()` and `free()`
  - That means you cannot call malloc() or free() anywhere in your code
    - Other than your own implementations
- TODO:
  - `mm_init()` ←This one is done for you! But you should still read it and understand
  - `mm_malloc(size_t size)` ←This is your malloc() implementation
  - `mm_free()` ← This should free the block that you mm_malloc()ed
  - And other helper functions ← Helper function makes debugging easier

Testing your implementation

- A driver program is provided to aid your testing (`mdriver`)
  - Runs various ***traces*** similar to queue lab
  - You can write your own traces (use `traces/short1-bal.rep` as an example to write your own test cases)
    - Designed to test all sorts of functionality of your implementation
    - Work on traces from easiest down, don't try to do them all at once
  - *The score from the driver != your final project score*
    - Gradescope will translate it into your final grade once you submit
      - *Practically impossible to get a 100 from the driver program*

- A `Makefile` is provided for you
  - Make sure to run `make` after every change in `mm.c`
  - This will generate several files, such as `mdriver`
- To run the program and check your implementation, use:
  - `./mdriver -f traces/nameofatracefile`
  - Start with the shorter traces(short1-bal, short2-bal)

Grading your implementation

- Your implementation will be tested for both space management and time complexity
  - How much memory are you using?
  - How fast are your operations?
  - Both must perform at a certain level in comparison to a “real” `malloc()` implementation
    - Glibc `malloc()` is used for comparison
  - Remember Space-Time tradeoff? ⇒ Essentially becomes an optimization problem

Project 2 Road Map

- Initially, we'll do a naïve implementation. (Phase 1) ← Should be done by now
  - This means we just need it to *work*
    - A working allocate and free is all that's needed
- Implement coalesce and splitting (Phase 2) done by middle of week 2
  - To start, focus on the early traces, which should require implementing
  - Should see a boost in space management once you are done
- Once everything else works, implement the explicit free list (Phase 3) done by deadline
  - Should see a massive performance improvement once you complete phase 3
    - *Why?*

- You will encounter bugs at every phase, but that doesn't mean the bug came from that phase!
  - Bugs in phase 2 may be coming from phase 1, …

Resources, if you get stuck…

- Project 2 Help Session
  - Recording and Slides posted to Discord
- Read the C Memory Management slides
  - Re-read them a couple times for good measure
  - Impossible to do well on project without a good conceptual understanding of memory allocation
- C issues: See K & R
  - An oldie but goodie
- Other places to turn to when you get stuck (Because you will probably get stuck)
  - CSAPP 3e Chapter 9.9
  - **Drawing it out on a whiteboard!**

Getting help from the TAs (and staff)

- You will most likely *get stuck* or have *bugs* in your program
  - That's normal.
  - Learning to identify and fix bugs is one of the learning objectives of this course
    - **Debugging**: Learning tools to help debug program crashes and break down existing programs.
- Hence, a simple *"can you look at my code?"* is not acceptable
  - Instead, narrow down where the problem is happening
    - **Use GDB!!!**
  - Isolate the region of code with the bug, and show us the code with the full error message.
- We want to help you succeed, but TAs are not compilers
  - After getting help and making the necessary changes, don't show us yourcode and ask "is this better?"
  - Instead, compile and test it! That's what programming is about: you make a thing, you test the thing, you fix the bugs.

Project 2 Help Session

Appended here for your convenience

Malloc Night 2023

Shinwoo Kim + Jake Kasper
CS 0449 Teaching Assistants

Spring 2023, Term 2234
5502 SENSQ
Feb 22nd, 2023

Agenda

- Conceptual Overview with Shinwoo
    - The standard `malloc()` interface
- Project Logistics
- Implementation Details with Jake
    - Designing your own `malloc()`!
- Debugging Tips and Tricks with GDB
- Q&A + TA-**aid**ed Debugging
    - **Game-plan:** Finish Phase 1 by the end of the night

Slides to be shared at end of the night ⇒ Discord

Conceptual Overview

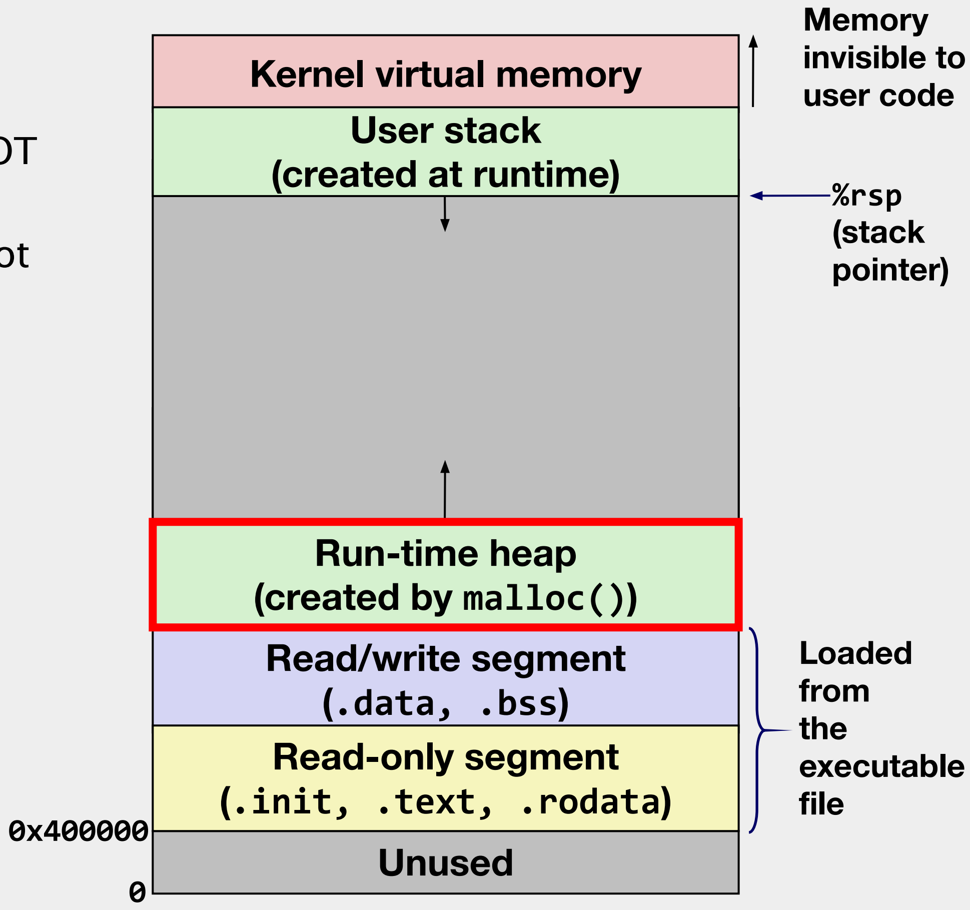
Shinwoo Kim

Dynamic Memory Allocation

Assigning memory at execution time

Dynamic memory allocation

- Used when
  - Data structures whose size is NOT known at compile-time
  - Particular chunk of memory is not needed for the entire run
    - Can reuse that memory for storing other things later
- Dynamic memory allocators manage an area of process address space known as the heap



m[emory] alloc[ator]

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Types of allocators
  - Explicit allocator manages memory for you
  - Implicit allocator requires the programmer to directly manage memory

Implicit allocator (Java)

```
String my_string = new String(“hello”);
```

Explicit allocator (C)

```
char *my_string = (char*)
malloc(10*sizeof(char));
strcpy(my_string, “hello”);
…
free(my_string);
```

The glib `malloc()` package

```
#include <stdlib.h>
void *malloc(size_t size)
```

- Successful:
  - Returns a **void** pointer to a memory block of at least size bytes
  - aligned to a 16-byte boundary (on x86-64)
  - 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()`, `calloc()`, 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

A `malloc()` example

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

void foo(long n)
{
    /* Allocate a block of n longs */
    long *p = (long *) malloc(n * sizeof(long));
    if (p == NULL) { // always check return value of malloc()
        perror("malloc"); // print error message
        exit(0);
    }

    /* Initialize allocated block */
    for (long i=0; i<n; i++)
    p[i] = i;
    /* Do something with p */
    . . .
    /* Return allocated block to the heap */
    free(p); // Always free after malloc()
}
```

For an array of n chars, we need to allocate `n*sizeof(chars)`

malloc returns a void pointer, so we need to cast it to the type of pointer we want

Note: `malloc()` only makes room for your data, it does NOT initialize your data!

https://sites.pitt.edu/~shk148/teaching/CS0449-2234/code/malloc_example.c

Project logistics

Your roadmap to success!

The malloc project.

- Now that we've seen `malloc()` work…
  - We get to make our own 🥳
    - That means you cannot call `malloc()` or `free()` anywhere in your code
      - Other than your own implementations

Design Goals

TODO:

- `mm_init()` ←This one is done for you! But you should still read it and understand it
- `mm_malloc(size_t size)` ←This is your `malloc()` implementation
- `mm_free()` ← This should free the block that you `mm_malloc()`ed
- And other helper functions ← Helper function makes debugging easier

Perform well in both time and space complexity
⇒ Essentially becomes an optimization problem

Malloc Lab Roadmap

- Initially, we'll do a naïve implementation. (Phase 1) done by end of week 1
  - This means we just need it to *work*
    - A working allocate and free is all that's needed
- Implement coalesce and splitting (Phase 2) done by middle of week 2
  - To start, focus on the early traces, which should require implementing
- Once everything else works, implement the explicit free list (Phase 3) done by deadline
  - Should see a massive performance improvement once you complete phase 3

Project logistics

- This project is <u>NOT</u> meant to be done is one sitting
  - If one of the TAs or staff sat down to do this lab from scratch, it would still take at least a week
- Plan ahead, leave plenty of time for *design*
  - Measure twice, cut once
- Work in small blocks of time
  - One or two hours, then take a break!
    - Your brain can keep working subconsciously
  - Leave enough time for your "*Eureka!*" moments
- Start the project and come to office hours as early as you can
  - Debugging this project is hard, even for TAs
  - Errors are generally too complicated to look at a little code in Discord and see the mistake
  - Asking for help last minute will likely not succeed

Modularity and Design

- Good style shouldn't be an afterthought
  - Hard to debug if you can't read your own code
  - Easier to explain to/get help from TAs with cleaner code
- *That is to say…*

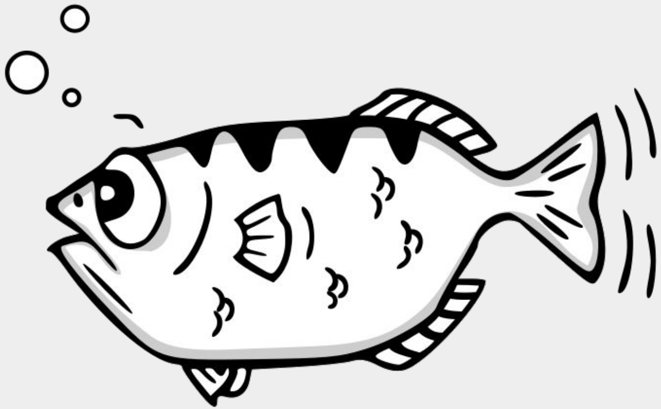
1. Avoid long if-else chains (may be a loop? switch?)
2. Think carefully about what exactly each function should do
   - Don't put everything in a single function
3. Descriptive comments!
   - comments as you go
   - Especially useful for **check-off meetings**

Testing and getting help

- The driver will run *traces* on your program
  - Similar to Queue Lab
  - Designed to test all sorts of functionality of your implementation
  - Work on traces from easiest down, don’t try to do them all at once
- Portion of grade is on performance
  - Based on how much *time* it took
  - Must perform at a certain level in comparison to a “real” `malloc()` implementation
- Use whiteboards or notebook paper
  - Do a lot of drawing of your free lists

Debugging

- You're given a couple of functions to debug your heap
  - `examine_heap()` - prints out the heap's contents
  - `check_heap()` - checks to make sure the heap is in a consistent state
  - Especially useful early on
- **GNU Debugger**
  - See Lab 0 for GDB walkthrough
    - Cheatsheet in Discord
  - No more `printf()` debugging
  - GDB is **huge** for debugging, especially on this lab



Getting help from the TAs (and staff)

- You will most likely *get stuck* or have *bugs* in your program
  - That’s normal.
  - Learning to identify and fix bugs is one of the learning objectives of this course
    - **Debugging**: Learning tools to help debug program crashes and break down existing programs.
- Hence, a simple *“can you look at my code?”* is not acceptable
  - Instead, narrow down where the problem is happening
    - **Use GDB!!!**
  - Isolate the region of code with the bug, and show us the code with the <u>full</u> error message.
- We want to help you succeed, but TAs are not compilers
  - After getting help and making the necessary changes, don’t show us yourcode and ask “is this better?”
  - Instead, compile and test it! That’s what programming is about: you make a thing, you test the thing, you fix the bugs.

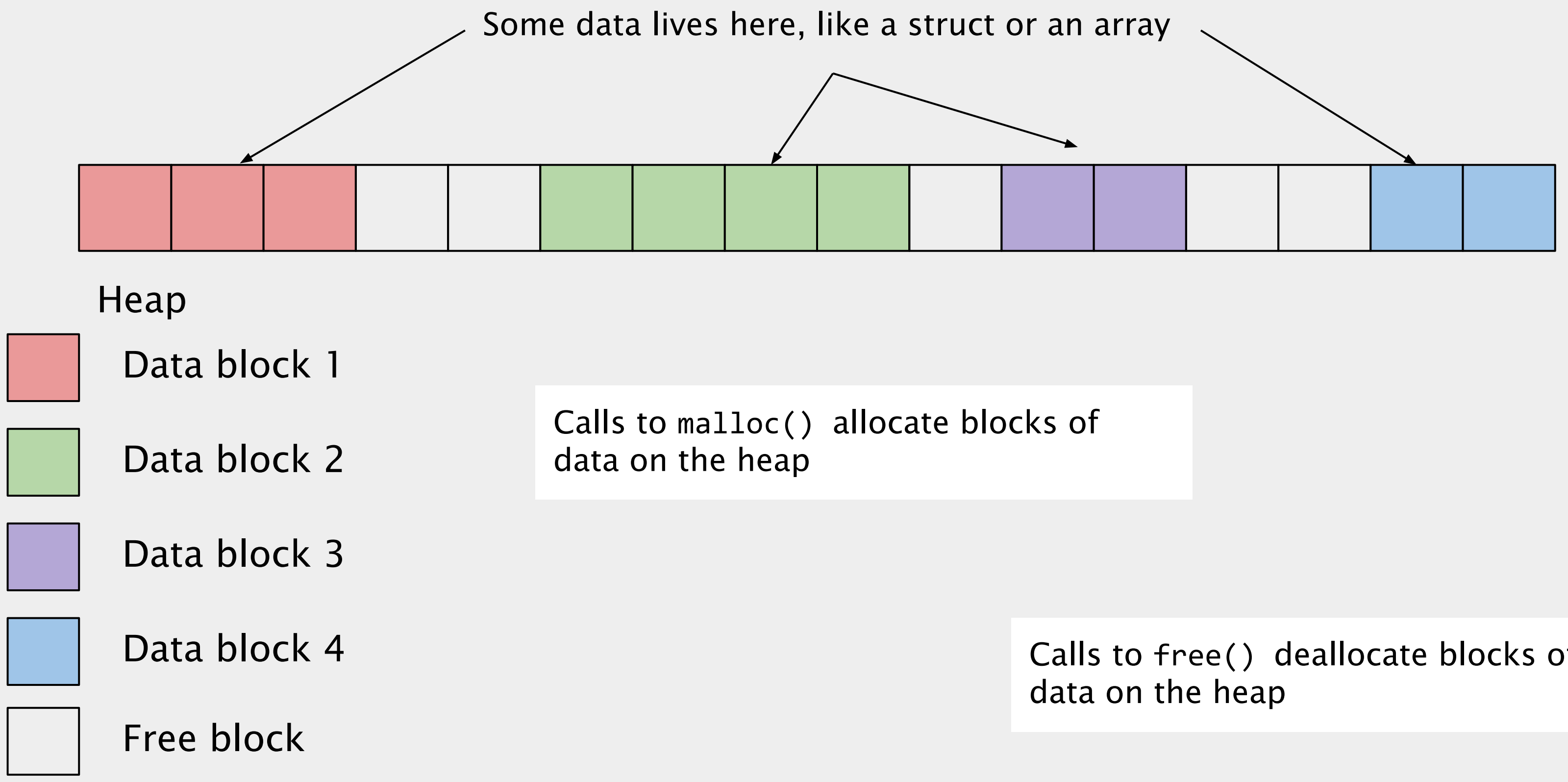
Other *non-breathing* resources

- Read the C Memory Management slides
  - Re-read them a couple times for good measure
  - Impossible to do well on project without a good conceptual understanding of memory allocation
- C issues: See K & R
  - An oldie but goodie
- Other places to turn to when you get stuck (Because you will probably get stuck)
  - CSAPP 3e Chapter 9.9
  - **Drawing it out on a whiteboard!**

Implementation Details

Jake Kasper

The Heap

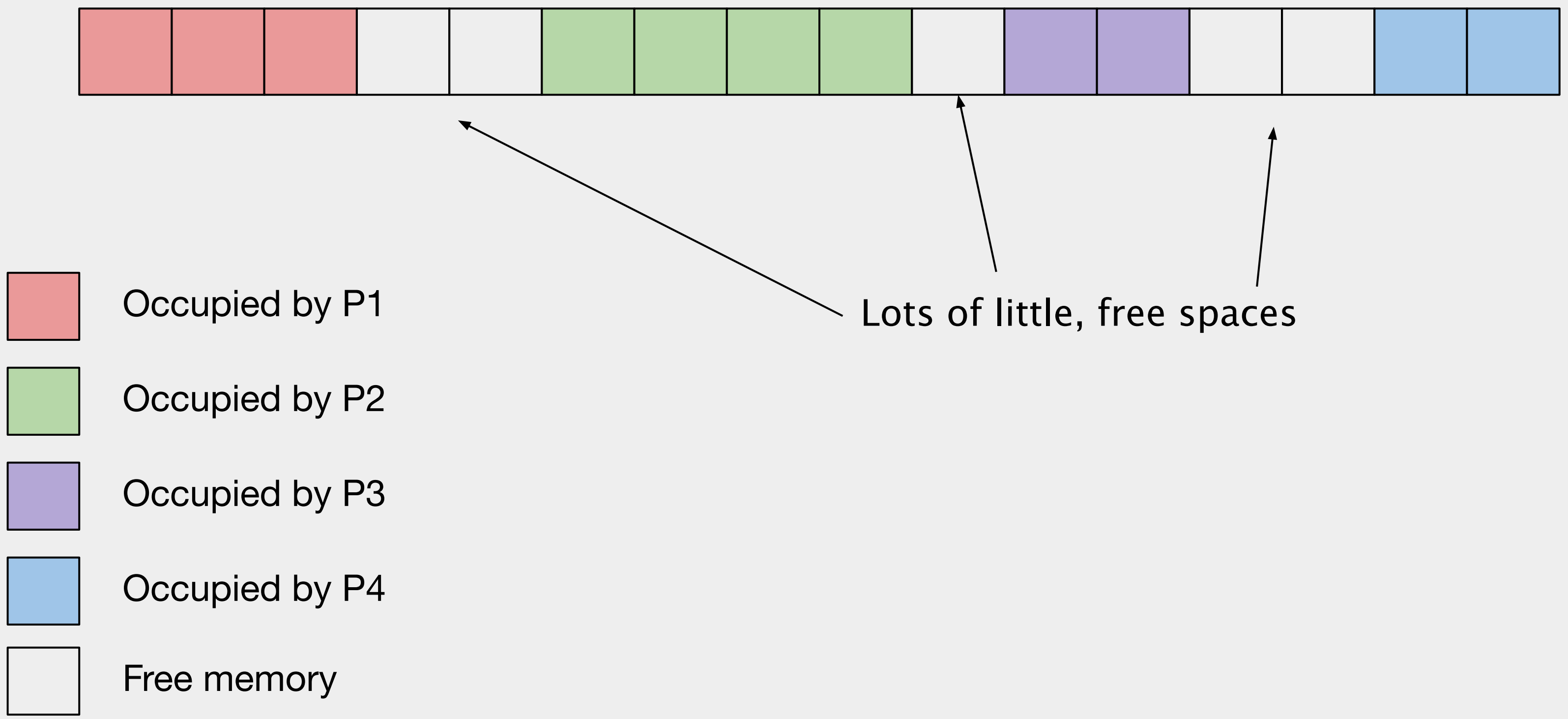


Calls to `malloc()` allocate blocks of data on the heap

Calls to `free()` deallocate blocks of data on the heap

Fragmentation

External



Struct Information

- The **BlockInfo** struct will contain the metadata for each heap block
  - **size** refers to the number of bytes this block contains
    - The allocated bytes for FreeBlockInfo are **included** in size
  - **prev** points to the previous block in the heap
- The **FreeBlockInfo** struct will contain the metadata for each free block (once we implement Phase 3, more later)
  - **nextFree** points to the next free block in the list of free blocks
  - **prevFree** points to the previous free block in The list of free blocks
- The **Block** struct is what makes up a block
  - The block's metadata is stored in **info** and, if it's a free block, **freeNode** will maintain its location in the free block list

```
typdef struct _BlockInfo{
  long int size;
  struct _Block* prev;
} BlockInfo
```

```
typdef struct _FreeBlockInfo{
  struct _Block* nextFree;
  struct _Block* prevFree;
} FreeBlockInfo
```

```
typdef struct _Block{
  BlockInfo info;
  FreeBlockInfo freeNode;
} Block
```

mm_init()

- The mm_init() initializes the allocator
  - subsequent calls reset the allocator
- Nothing to modify
  - But you still need to understand it
- Tells us we need to manage:
  - free_list - we will ignore this until phase 3
  - malloc_list_tail - points to the last block in the heap
    - Used to actually ensure the validity of your heap, so it's important to maintain
      - examine_heap()
      - check_heap()
  - heap_size - the actual byte size of your heap
    - Your implementation should NOT manually change this value, but you should understand how it's changed in requestMoreSpace()
- This function is called by the other source files we won't be editing

```c
int mm_init() {
  free_list_head = NULL;
  malloc_list_tail = NULL;
  heap_size = 0;
  return 0;
}
```

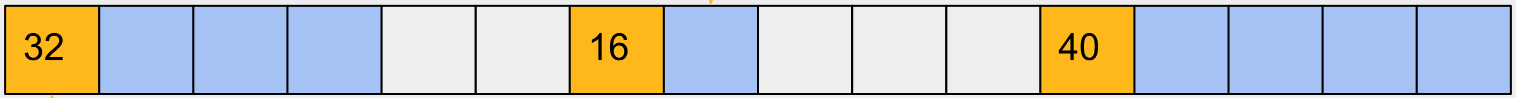
mm_malloc(size_t size)

- Allocates a block of memory of the given size

```
void* mm_malloc(size_t size) {
  Block* ptrFreeBlock = NULL;
  Block* splitBlock = NULL;
  long int reqSize;

  // Zero-size requests get NULL.
  if (size == 0) {
    return NULL;
  }

  // Determine the amount of memory we want to allocate
  reqSize = size;

  // Round up for correct alignment
  reqSize = ALIGNMENT * ((reqSize + ALIGNMENT - 1) / ALIGNMENT);
  ptrFreeBlock = searchList(reqSize);
```

`malloc()` and `free()` have to consider...

This block only needed one word for memory, but the added header doubled the amount of allocated space it needed



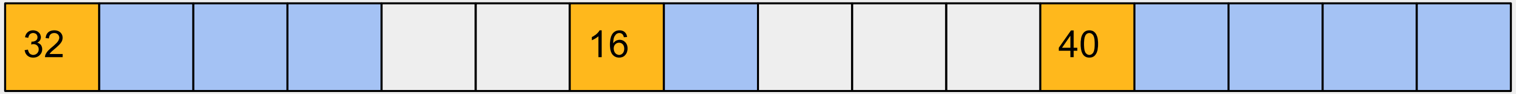
Header before block contains block length

Block header contains block size

Memory allocator looks at block header and frees that many subsequent bytes

`malloc()` and `free()` have to consider…

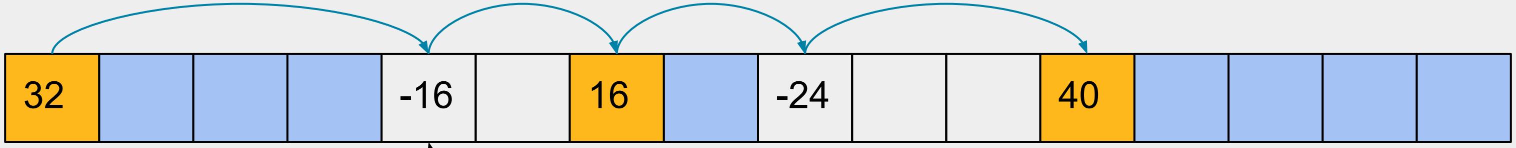
- **4 ways of keeping track of free memory**
  - Implicit List ← Start Here
  - Explicit List ← Final implementation of project
  - Segregated List
  - Sort blocks by size



Implicit Free List

```
malloc(8);
```

Keep a list of ALL blocks

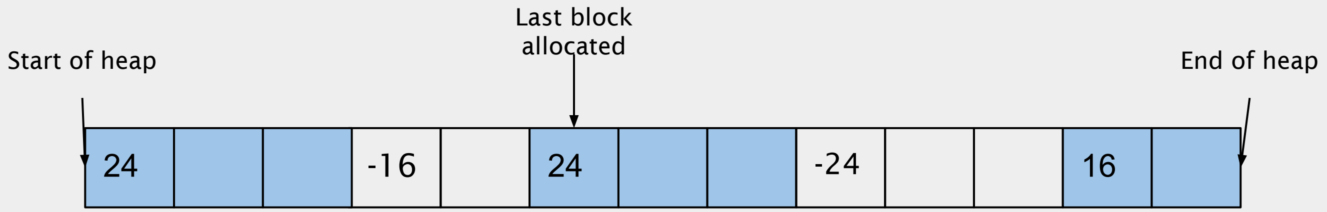


Header of a free block will contain the size of the free block (negative to denote that block is free)

`mm_malloc(size_t size)`

1. Determine the required size (adjust for correct alignment)
2. Search the list of blocks
   - ➢ Are any of them big enough?
     - i. Let’s give that back to the callee of `mm_malloc()`
3. What if there are no blocks big enough for our requested size?
   - ➢ We need to request more space from the system
   - ➢ `requestMoreSpace(size_t reqSize)` - grows the heap
     - i. Look at how this function is implemented and try to understand it
4. Now we have a block (either from the list or from our request)
   - ➢ How do we mark it as allocated?
5. Let’s give it our function callee

Implicit free lists: finding free blocks



`malloc(16);`

First fit:

Next fit:

Best fit:

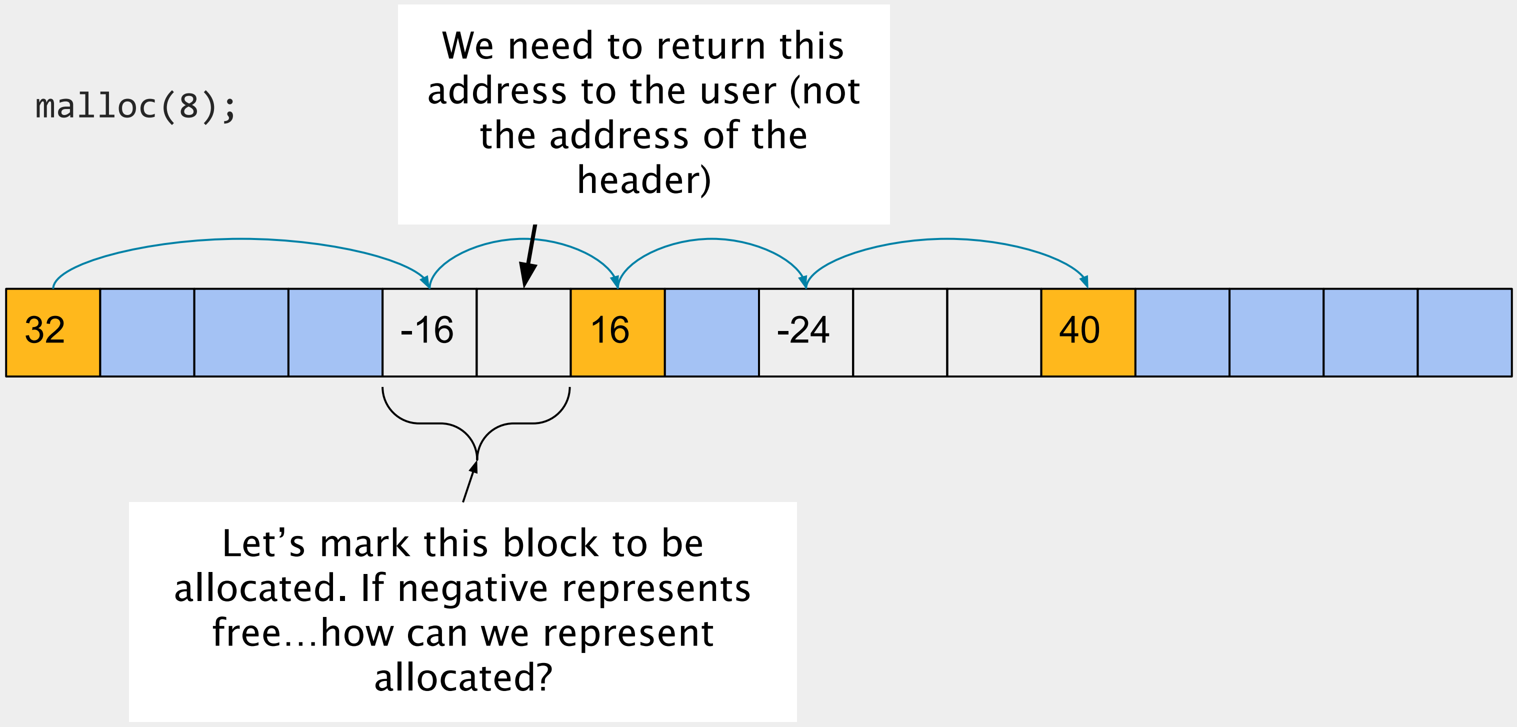
First fit: starts at beginning of heap and selects the first block that can be used

Next fit: same as first fit, but starts searching from the previously allocated block

Best fit: finds the block which fits the best (least amount of wasted space)

Implicit Free List

```
malloc(8);
```



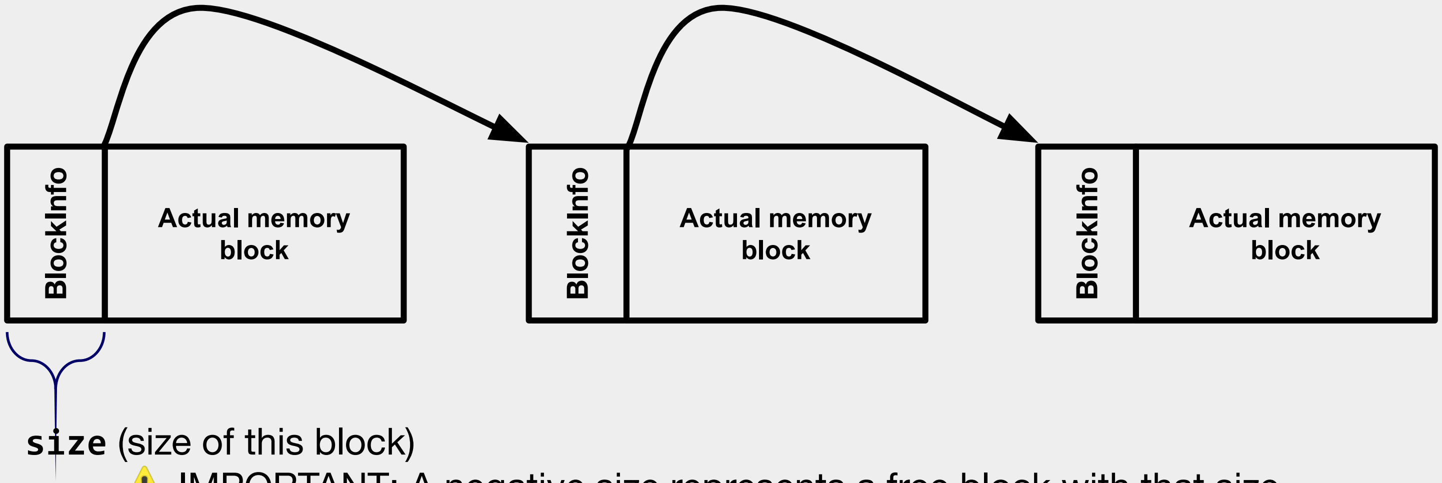
Phase 2: Splitting Free Blocks

- Currently, if we find a block that's big enough, we allocate it.
  - We are massively over-allocating memory
  - We might ask for 4B, get get 32B.
- … If we have a block that's large enough to split into 2, let's do it!
  - A block **at minimum** needs to contain the header + alignment padding
- If we run into a really big block, we could split into two parts:
  - One part just big enough to fit our requested block
  - Another part at least as big as our minimum block size
- After splitting, need to get a pointer to the newly created blocks
  - Special, unscaled pointer arithmetic can help here
  - Make sure all of your new blocks are added to the list
- **Draw out** a test case, with at least 3 to 5 blocks, trying several cases
  - Helps a lot for this function and coalesce

Phase 3: Explicit Lists

- Searching a list is slow…
  - How can we speed it up?
    - Reduce the number of elements in the list!
- Let’s create a list with just the free blocks
  - That way we’re not traversing over allocated blocks
    - Which we can’t allocated anyways
- Make sure to maintain this list upon coalescing…
  - Your implementation may not guarantee that two free blocks next to each other in the heap are also next to each other in the free list…

Currently, an implicit list

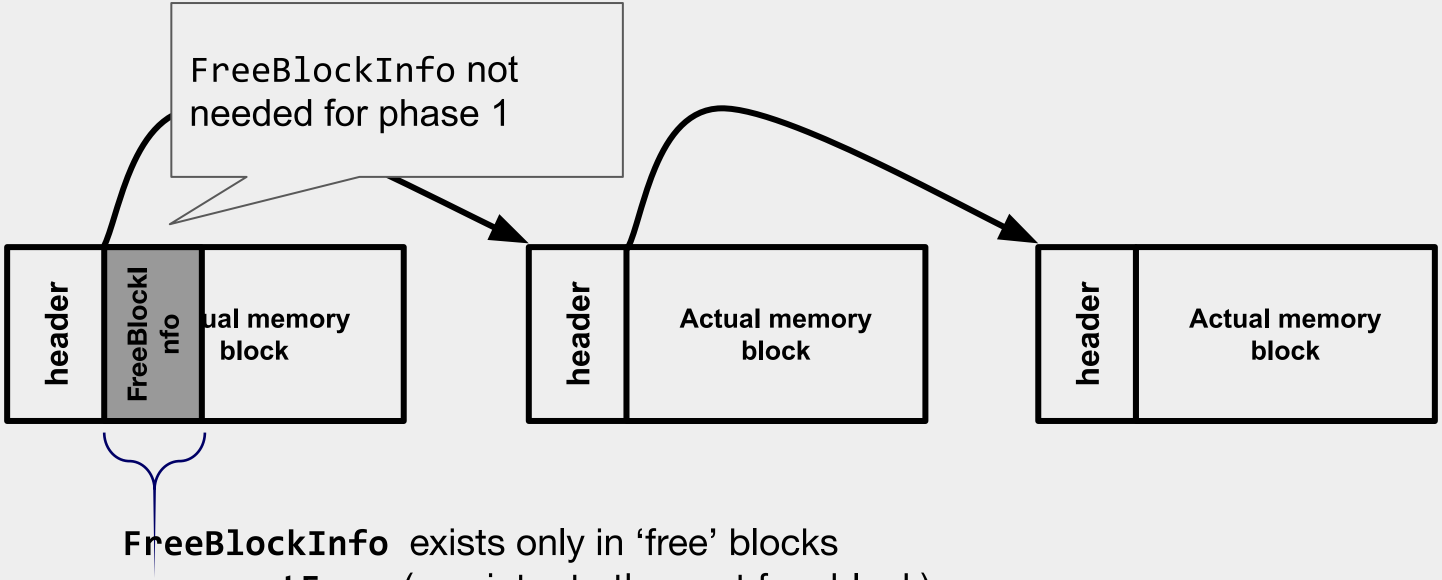


size (size of this block)

⚠️ IMPORTANT: A negative size represents a free block with that size

prev (pointer the previous block in the list)

Explicit List



FreeBlockInfo exists only in 'free' blocks

nextFree (a pointer to the next free block)

prevFree (a pointer to the last free block)

These pointers form the basis of an **explicit free list.**

- When a block is free, these will be useful
- When a block is allocated ('in-use'), you don't need to do anything about this struct, its data will simply be overwritten by the user

mm_free(void* ptr)

- Deallocate the given pointer that was previously allocated by `mm_malloc()`

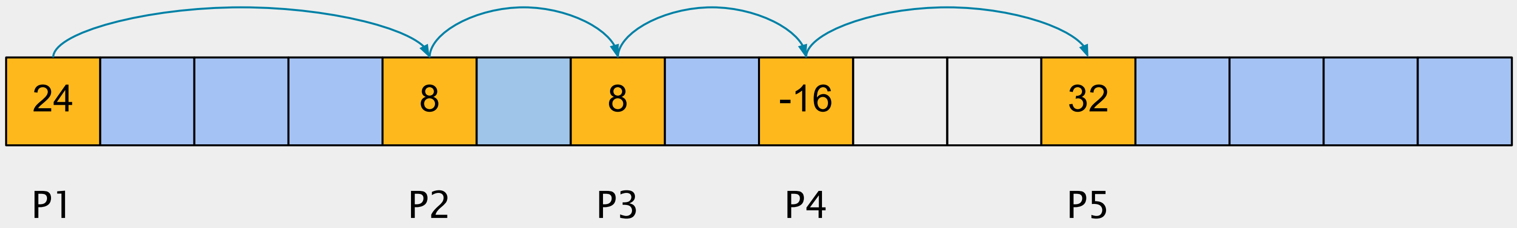
1. Get to the header of the block and mark it as free
   a. `ptr` points to the data portion of the block, not the header
   b. How can we get back to the header?
   c. How can we denote a block is free
2. **Phase 2**: implement `coalesce()`
   a. As described in lecture

```
void mm_free(void* ptr) {
  Block* blockInfo = (Block*)UNSCALED_POINTER_SUB(ptr, sizeof(BlockInfo));

      // YOUR CODE

  coalesce(blockInfo);
}
```

Memory Allocation Example

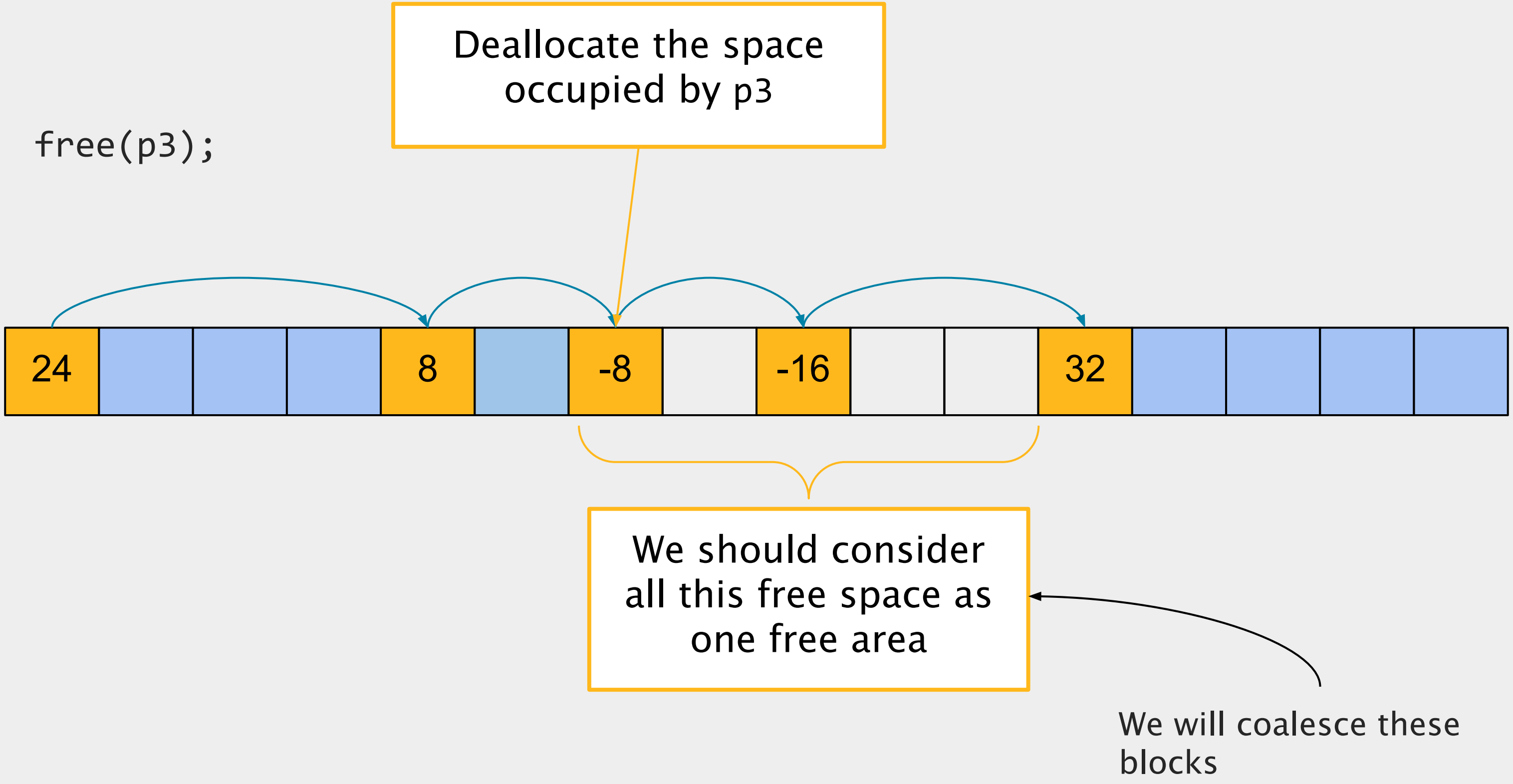
Implicit Free List - Example Freeing

```
free(p3);
```



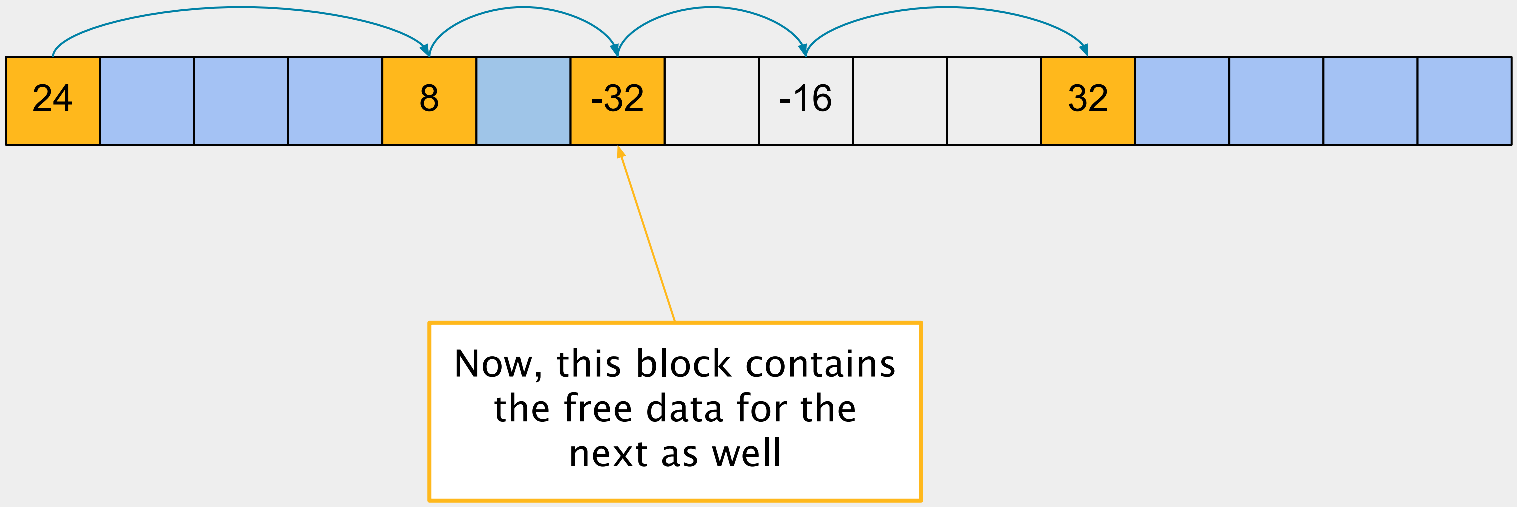
Implicit Free List - Example Freeing

```
free(p3);
```



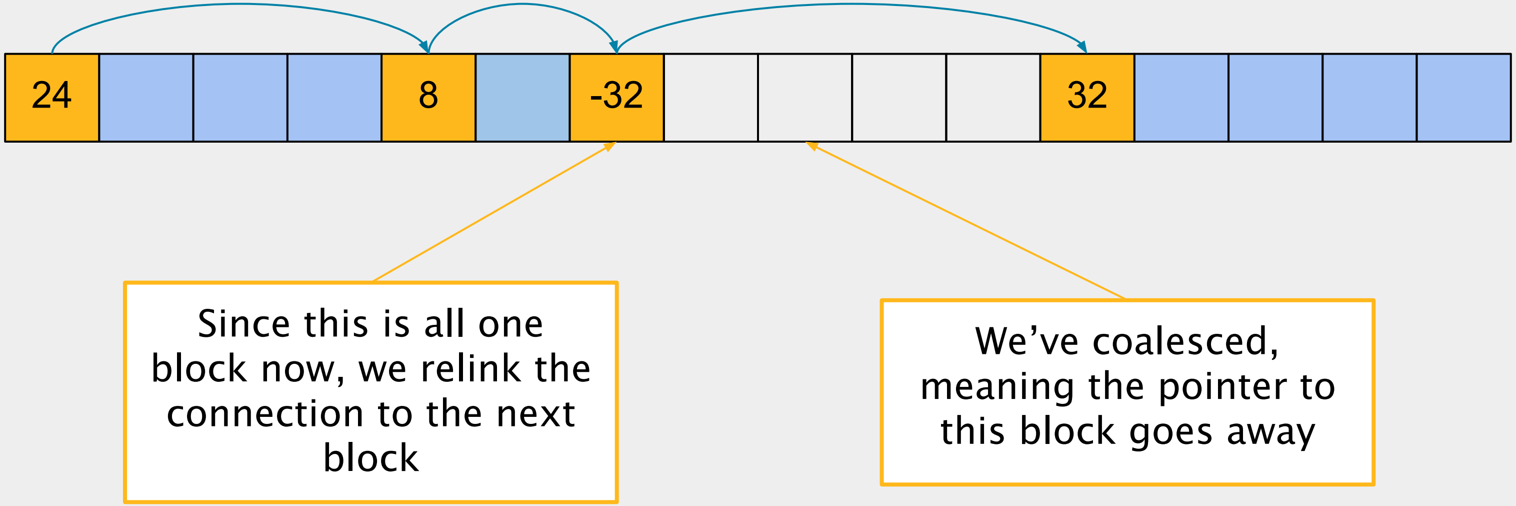
Implicit Free List - Example Freeing

```
free(p3);
```



Implicit Free List - Example Freeing

```
free(p3);
```



Works Referred

Gavin Heinrichs-Majetich’s CS 0449 Recitation Slides (Fall 2022)
Martha Dixon’s CS 0449 Recitation Slides (Fall 2020)
Randal Bryant & David R. O'Hallaron’s Computer Systems: A Programmer's Perspective
Carnegie Mellon University’s 15-213: *Introduction to Computer Systems* (Fall 2017)