

November 16–18th, 2022

**Independent
Systems Programming
Conference**

Introduction to Memory Allocation: Design and Implementation

Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube: [www.youtube.com/c/MikeShah](#)

17:00–18:00, Wed, 16th November 2022

<https://handmade-seattle.com/>

60 minutes | Introductory/Intermediate Audience

Please do not redistribute slides without prior permission.

Goal(s) for today

What you're going to learn

- For our audience
 - Learn about memory allocation
 - Understand how we would implement a memory allocator from scratch
 - Look at different types of allocators
 - Look at the design of some of the more popular allocators
- The target experience level for this talk is probably more beginner level
 - I think beginners will get a lot out of the examples and resources in one place
 - I *hope* this talk encourages intermediate users to think and try building an allocator
 - (This is the handmade community after all, I assume folks will want to build at least a simple implementation at some point :))
 - I think experts will have strong opinions already on this topic already

Your Tour Guide for Today

by Mike Shah (he/him)

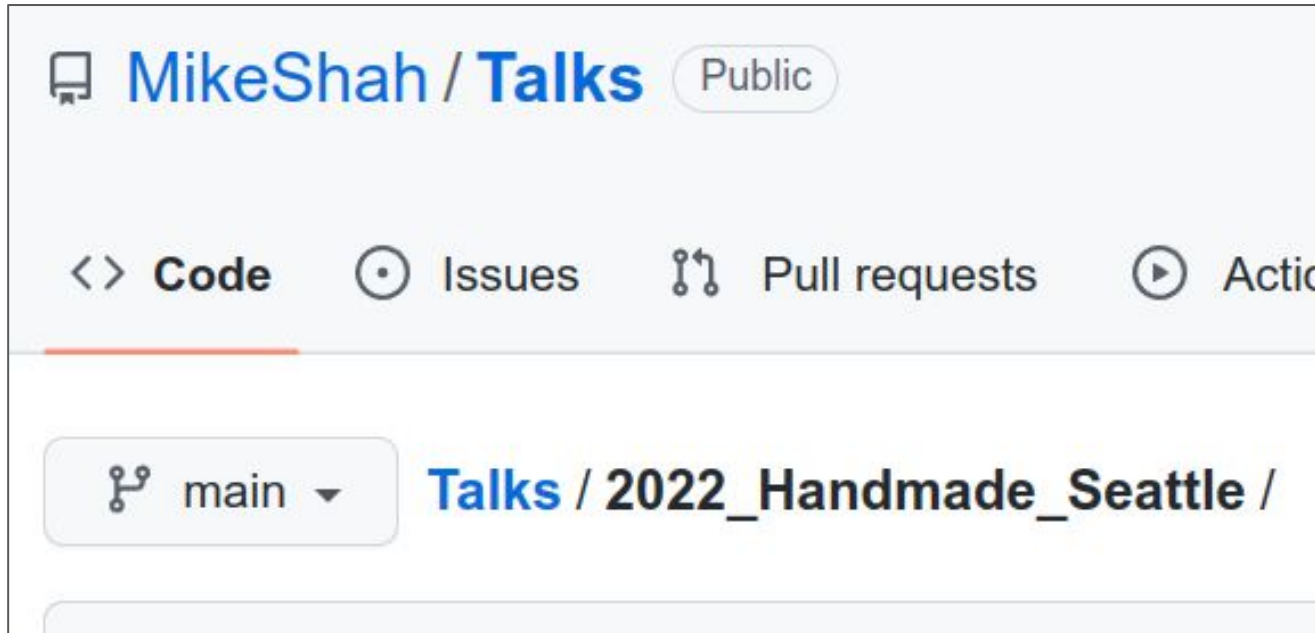
- **Associate Teaching Professor** at Northeastern University in Boston, Massachusetts.
 - I teach courses in computer systems, computer graphics, and game engine development.
 - My **research** in program analysis is related to **performance** building static/dynamic analysis and software visualization tools.
- I do **consulting** and technical training on modern C++, Concurrency, OpenGL, and Vulkan projects
 - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of **computer graphics**, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io
- More online training coming at courses.mshah.io



Code for the talk

- Located here:

https://github.com/MikeShah/Talks/tree/main/2022_Handmade_Seattle



The abstract that you read and enticed you to join me is here!

Abstract

Has anyone told you that memory allocation is too slow? Do your colleagues *shutter* when you say malloc? Is malloc actually slow? How would we know if allocation is too slow? In this talk I will provide an introduction to stack and heap based memory allocation strategies and trade-offs. We'll understand the difference between stack and heap based memory, and then move on to different implementations of stack and heap based allocators and where they might be used. Throughout the talk, I will also show you how to build a simple heap memory allocator to demonstrate by example some of the design decisions that you have to make.

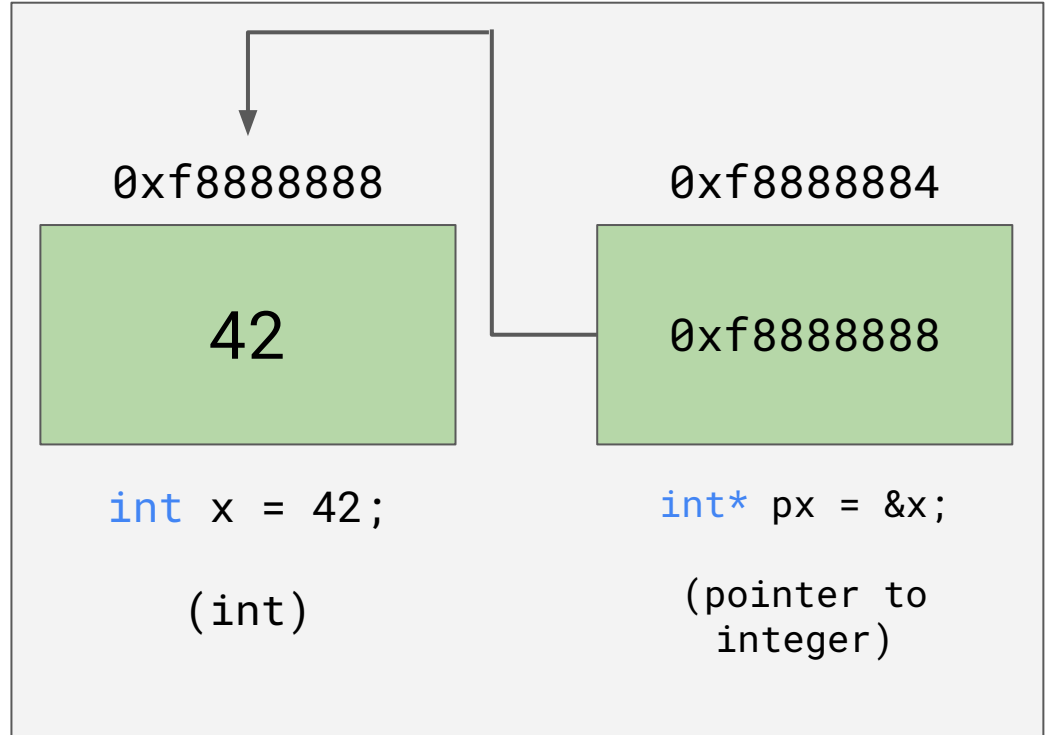
Prerequisite Knowledge for this Presentation

With a few links so you can get up to speed if needed

Note: I'm going to flip through these quite fast -- our audience in attendance is likely well versed, but for you folks in the future watching this I hope this is helpful.

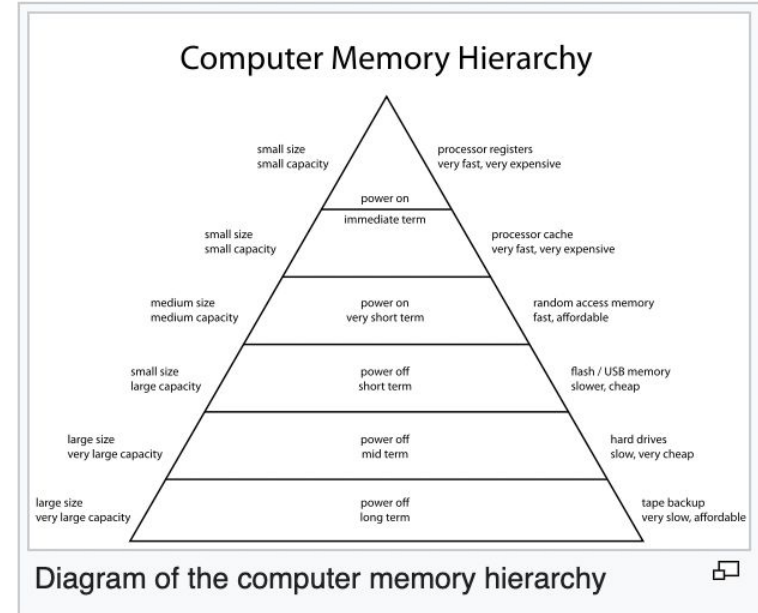
Prereq 1/4 - What is a raw pointer?

- If this picture makes sense to you, and you can explain in your own words what a pointer is -- let's proceed!
- Don't have the prerequisite? Search on YouTube [Learn and understand \(almost\) everything about the fundamentals of C++ pointers in 96 minutes](#)



Prereq 2/4 - There are multiple types of physical memory

- It's useful to understand there's a memory hierarchy, with different types of memory,
 - and 'the why' for the different layers (**locality**) and how that improves performance we have it for performance
- It will be useful to know a little bit about how **virtual memory** [\[wiki\]](https://en.wikipedia.org/wiki/Virtual_memory) works
 - And why **virtual memory** allows programmers to 'think' think about memory as a contiguous segment.



https://en.wikipedia.org/wiki/Memory_hierarchy

Prereq 3/4 I'll assume you've used malloc/free found in <stdlib.h> or <malloc.h>

- **void* malloc(size_t size);**
 - On Success Returns a pointer to a memory block of at least size bytes
 - For x86 this memory is aligned to 8-byte boundaries
 - For x86-64 this memory is aligned to 16-byte boundaries
 - A size of 0 returns NULL
 - Unsuccessful allocation: returns NULL(0)
- **void free(void *p);**
 - Reclaims memory allocated by malloc, calloc, or realloc.
 - (Returns the block pointed at by p to pool of available memory)
 - p must come from previous call to malloc (or realloc)
 - Note: Never use free with **delete** (used in C++), these are different allocators!
- **void* calloc(size_t nmemb, size_t size);**
 - Similar to malloc, but initializes the allocated block to zero.
- **void* realloc(void* ptr, size_t size);**
 - Changes size of previously allocated block, contents of new block unchanged

```
1 // mallocexample.c
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(){
6
7     // Allocate a contiguous block of 50 integers
8     int* data = (int*)malloc(50*sizeof(int));
9
10    // If our allocation fails, malloc returns NULL
11    if(data==NULL){
12        exit(1); // Throw some error
13    }
14
15    // Otherwise initialize our allocated blocks
16    // We can use a loop to do this, or other
17    // commands like 'calloc' or 'memset'
18    int i;
19    for(i=0; i < 50; ++i){
20        data[i] = i;
21    }
22
23    // Return allocated block of size '50*sizeof(int))' to the heap
24    free(data);
25
26    return 0;
27 }
```

Prereq 4/4 I'll assume you've used malloc/free found in <stdlib.h> or <malloc.h>

- `void* malloc(size_t size);`
 - On Success Returns a pointer to a memory block of at least size

byte

Most example code will just use plain C or C++, and you can translate to whatever language you like.

- `void free(void* ptr);`
 - Releases a memory block previously allocated with malloc

■

◦ pm

Note: Sometimes I'll use 'malloc' or 'new' depending on what's more convenient or clear but the idea is interchangeable

- `void* calloc(size_t n, size_t size);`
 - Similar to malloc but initializes the memory to zero

- `void* realloc(void* ptr, size_t size);`
 - Changes the size of a memory block

unchanged

```
1 // mallocexample.c
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(){
```

```
    // Allocate an array of 50 integers
    int* arr = malloc(sizeof(int)*50);
    if (arr == NULL) {
        printf("malloc returns NULL\n");
        return 1;
    }

    // Initialize the array
    memset(arr, 0, sizeof(int)*50);

    // Use the array
    // ...

    // Free the memory
    free(arr);
}
```

One last note -- examples are often graphics/games (1/2)

- Most of my examples will be slanted towards 3D graphics/game programming
 - My apologies and I take no offense if you don't like or play video games!
 - Most of the material from this presentation should be able to be applied to other industries (especially where memory allocation matters!)
 - e.g. low latency trading applications, computational biology, etc.
- Also, I'm going to break every *good* powerpoint rule about the amount of text on a slide that is appropriate.
 - Why? This is somewhat my style, but I personally like having slides where I can remember what the speaker was talking about.

One last note -- examples are often graphics/games (2/2)

- Most of my examples will be slanted towards 3D graphics/game programming
- My apologies and I take no offense if you don't like or play video games.
- Most of the material can be applied to other industries (e.g. low latency)

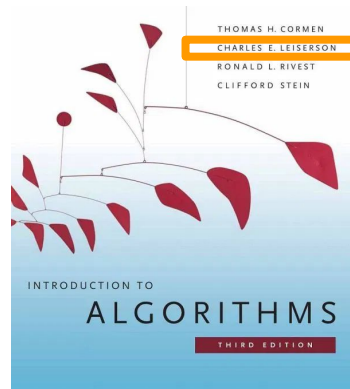
**Alright, let's talk
about memory
allocators!**

Memory Allocators

Why do we care? What are the Benefits?

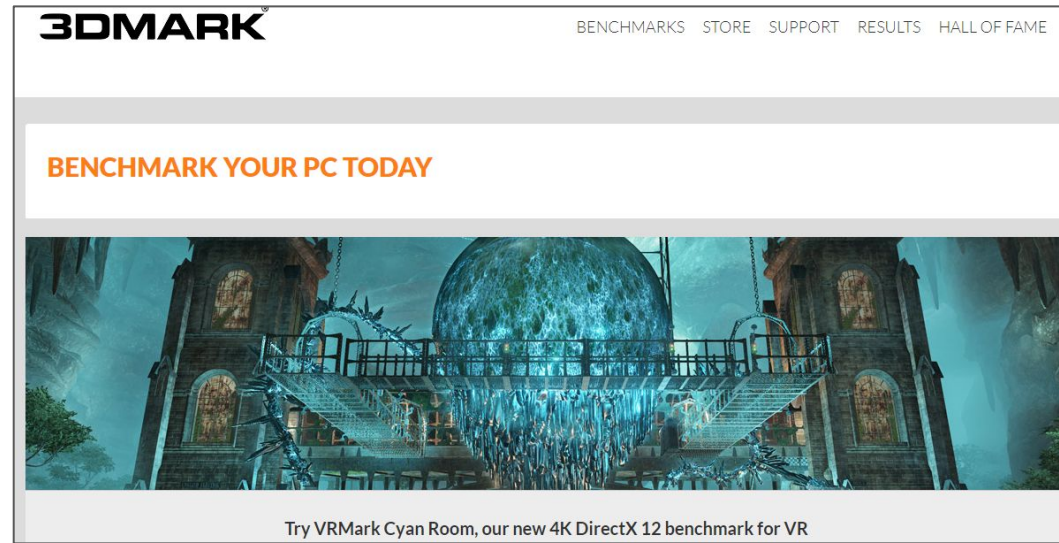
**Performance is the currency
of computing.**

“Performance is the currency of computing. You can often “buy” needed properties [of software] with performance” –
Charles Leiserson



1. Performance


- Pragmatically, better performance means you can do more, or more interesting computation elsewhere with your resources.
 - Often this means better ‘something’ which is often:
 - Graphics
 - AI
 - More precise physics
 - Gameplay
 - More battery life
 - etc.
- So when creating a memory allocator we’ll achieve performance by thinking about things like ‘locality’ and ‘contention’



<https://www.3dmark.com/>

2. Safety

- Memory allocators can provide a layer of abstraction for safety
 - <https://news.ycombinator.com/item?id=33553668>

National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

3. Economics

- Better utilization of memory, less time spent on cpus, and lower costs
- (We can probably quantify this by measuring how well our allocators prevent or reduce bugs and save engineers time)

How do you pay for AWS?



Pay-as-you-go

Pay-as-you-go allows you to easily adapt to changing business needs without overcommitting budgets and improving your responsiveness to changes. With a pay-as-you-go model, you can adapt your business depending on need and not on forecasts, reducing the risk of overprovisioning or missing capacity.

[Read more »](#)



Save when you commit

For AWS Compute and AWS Machine Learning, Savings Plans offer savings over On-Demand in exchange for a commitment to use a specific amount (measured in \$/hour) of an AWS service or a category of services, for a one- or three-year period.

[Read more »](#)



Pay less by using more

With AWS, you can get volume based discounts and realize important savings as your usage increases. For services such as S3, pricing is tiered, meaning the more you use, the less you pay per GB. AWS also gives you options to acquire services that help you address your business needs.

[Read more »](#)

https://aws.amazon.com/pricing/?aws-products-pricing.sort-by=item.additionalFields.productNameLowercase&aws-products-pricing.sort-order=asc&awsf.Free%20Tier%20Type=*all&awsf.tech-category=*all

Only pay for what you use

With Google Cloud's pay-as-you-go pricing structure, you only pay for the services you use. No up-front fees. No termination charges. Pricing varies by product and usage—[view detailed price list.](#)

https://cloud.google.com/pricing/?gclid=Cj0KCQiApb2bBhDYARIsAChHC9tIUZxgL_AnVfEhB_EzfKLV8SUhbzKzbK1S9g9v2bNGhyHVXyKoh2flaAkUcEALw_wcB&gclidsrc=aw.ds

Should I *still* write my own allocator? (1/2)

- A simple one (*which also might be the right one*) at the least will give you an appreciation I think of memory.
 - You will have to think about trade-offs (which is what different allocators offer) which is a good thing.
 - For newer programmers (especially when using purely C) I think it may also give you a ‘new mental model’
 - C to me is a ‘data layout language’
 - We’re just accessing (reading/writing) ‘bytes’ from a giant array of memory.

Should I *still* write my own allocator? (2/2)

- A simple one (*which also might be the right one*) at the least will give you an appreciation of the thing.

- You will learn to think about things.
- For newer people, it's a good mental model.
- C to me
- We

To the handmade community

- I think we just like to know how things work
 - Sometimes it's the right thing to do
 - i.e. build a tool/library that solves your exact problem given your input.
- But importantly, it helps us innovate later.
- And as I said, it may give you another way to **think** about computer science, abstraction, and models of computation
- (psst...and it's fun!)

Why should you not write an allocator?

- You don't have the budget (time & money) to spend on this
 - Creating an allocator requires some amount of thinking and design before starting
 - It's possible that a new allocator introduces complexity that is not needed in your project beyond what well purposed global allocators (e.g. malloc or new) offer.
- You don't really need better performance beyond what a general allocator (e.g. 'malloc' in C or 'new' in C++) provide.

(Quick Review)

Hardware -- Our “Working Memory”

(Also called ‘main memory’ or specifically DRAM - Dynamic Random Access Memory)



We have many types of physical memory

- The goal of memory is to store 'data'
 - The duration of that storage could vary depending on the storage medium
 - (e.g. a hard drive or cloud storage *should* store information indefinitely)
- (Aside: As an expert, you're probably thinking more about the mediums, allocators, allocation size, where the memory lives, data access patterns, data lifetime and various 'misses' that can occur in different caches.)



Programmers View of working (or 'main') memory (1/3)

- So roughly speaking we have a contiguous block of memory that looks something like this.



Address (in Hex)	Value
0x1000000B	
0x1000000A	
0x10000009	
0x10000008	
0x10000007	
0x10000006	
0x10000005	
0x10000004	
0x10000003	
0x10000002	
0x10000001	
0x10000000	

Programmers View of working (or 'main') memory (2/3)

- So roughly speaking we have a contiguous block of memory that looks something like this.
- When we create a variable, a certain amount of that storage is allocated for that variable.



Address (in Hex)	Value
0x1000000B	
0x1000000A	
0x10000009	
0x10000008	
0x10000007	
0x10000006	
0x10000005	
0x10000004	
0x10000003	
0x10000002	
0x10000001	
0x10000000	

Programmers View of working (or 'main') memory (3/3)

- So roughly speaking we have a contiguous block of memory that looks something like this.
- When we create a variable, a certain amount of that storage is allocated for that variable.
 - for example
 - `int x = 42;`
 - (int is usually 4 bytes, thus 4 bytes taken in the illustration where 1 box = 1 byte of memory)



Address (in Hex)	Value
0x1000000B	42
0x1000000A	
0x10000009	
0x10000008	
0x10000007	
0x10000006	
0x10000005	
0x10000004	
0x10000003	
0x10000002	
0x10000001	
0x10000000	

Operating System View of Processes (1/3)

- Now keep in mind we have many programs running at once
 - So per process our operating system has given some memory allocated to each process.



Address (in Hex)	Value
0x10000000 + N bytes	
...	
...	
...	
...	
...	
...	
...	
...	
...	
0x10000000	

Operating System View of Processes (2/3)

- Now keep in mind we have many programs running at once
 - So per process our operating system has given some memory allocated to each process.
 - Here are two processes for example



Address (in Hex)	Value
0x20000000 + M	Process B
...	
...	
0x20000000	
...	
...	
...	
...	
0x10000000 + N	Process A
...	
...	
0x10000000	

Operating System View of Processes (3/3)

- Now keep in mind we have many programs running at once
 - So per process our operating system has given some memory allocated to each process.
 - Here are two processes for example
 - **Let's focus on just one process and zoom in a little bit on the memory in that single process**



Address (in Hex)	Value
0x20000000 + M	Process B
...	
...	
0x20000000	
...	
...	
...	
...	
0x10000000 + N	Process A
...	
...	
0x10000000	

(Quick Review)

Segments of a Running Process

Address (in Hex)	Value
$0x10000000 + N$	Process A
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
$0x10000000$	

A Single Processes Memory Segments (1/2)

- A process (i.e. a program running on your program) is organized into a few segments of memory

Address (in Hex)	Value
0x10000000 + N	Process A
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
0x10000000	

A Single Processes Memory Segments (2/2)

- A process (i.e. a program running on your program) is organized into a few segments of memory
 - (Read from bottom to top)
 - code (or .text) is where our code is
 - data (initialized and uninitialized data stored in an object file)
 - heap for dynamically allocated memory
 - stack for our 'temporary' memory
 - (Aside: There may be many other segments as well -- use otool or objdump to see other sections like debug, exception table, etc.)

Address (in Hex)	Value
0x10000000 + N	Stack
...	
...	
...	
...	
...	
...	Heap
...	
...	Data
...	
0x10000000	Code

(Aside) From a Code Standpoint

- A process on an operating system is just a 'struct'
 - Observer there's a pointer to the stack
 - Each process also gets a 'page table' (page definition useful later) for heap allocated memory

```
65 // Per-process state
66 struct proc {
67     uint sz;                // Size of process memory (bytes)
68     pde_t* pgdir;           // Page table
69     char *kstack;           // Bottom of kernel stack for this process
70     enum procstate state;    // Process state
71     volatile int pid;        // Process ID
72     struct proc *parent;     // Parent process
73     struct trapframe *tf;    // Trap frame for current syscall
74     struct context *context; // swtch() here to run process
75     void *chan;              // If non-zero, sleeping on chan
76     int killed;              // If non-zero, have been killed
77     struct file *ofile[NOFILE]; // Open files
78     struct inode *cwd;        // Current directory
79     struct shared *shared;    // Shared memory record (0 -> none)
80     char name[16];           // Process name (debugging)
81 };
```

<https://github.com/jeffallen/xv6/blob/master/proc.h>

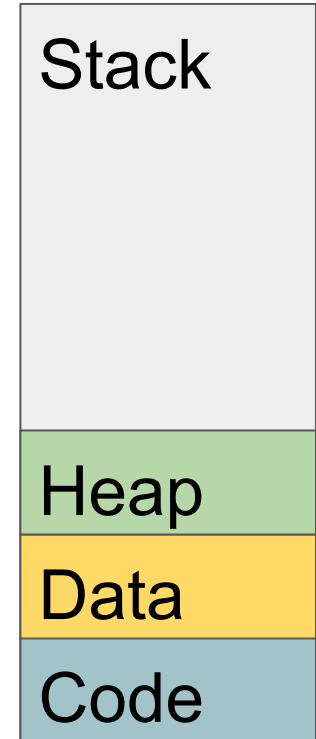
Stack

Heap

Data

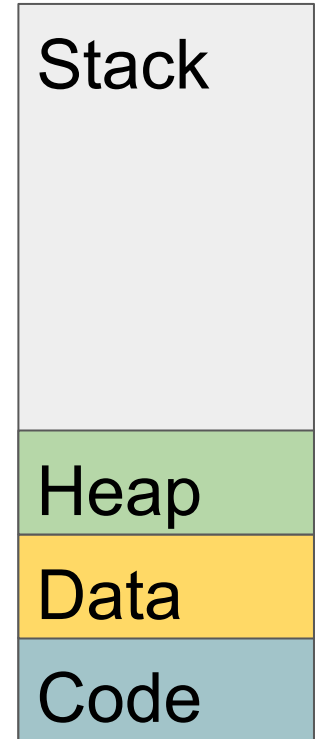
Code

Memory Allocation



Memory Allocation

- In order to understand the different 'segments' of memory, let's take a look at 'stack' and 'heap' memory.



A Process and its stack memory (1/5)

```
1 // g++ -g -std=c++20 stackmemory.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     std::cout << &x << '\n';
7
8     return 0;
9 }
```

Address (in Hex)	Value
0x16d5ff988	Stack
...	
...	
...	
...	
...	
...	Heap
...	
...	
...	Data
...	
0x10000000	Code

A Process and its stack memory (2/5)

```
1 // g++ -g -std=c++20 stackmemory.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     std::cout << &x << '\n';
7
8     return 0;
9 }
```

- Executing this line, we'll allocate on the 'stack' space for x.
 - x stores the value '42'

Address (in Hex)	Value
0x16d5ff988	Stack
...	
...	
...	
...	
...	
...	
...	
...	Heap
...	
...	Data
...	
0x10000000	Code

A Process and its stack memory (3/5)

```
1 // g++ -g -std=c++20 stackmemory.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     std::cout << &x << '\n';
7
8     return 0;
9 }
```

- Executing this line, we'll allocate on the 'stack' space for x.
 - x stores the value '42'

Address (in Hex)	Value
0x16d5ff988	42
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	Heap
...	
...	Data
...	
0x10000000	Code

A Process and its stack memory (4/5)

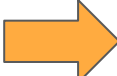
```
1 // g++ -g -std=c++20 stackmemory.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     std::cout << &x << '\n';
7
8     return 0;
9 }
```

- And we can use '&' operator to retrieve that actual stack address!

Address (in Hex)	Value
0x16d5ff988	42
...	
...	
...	
...	
...	
...	
...	
...	
...	Heap
...	
...	Data
...	
0x10000000	Code

A Process and its stack memory (5/5)

```
1 // g++ -g -std=c++20 stackmemory.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     std::cout << &x << '\n';
7
8     return 0;
9 }
```



- And we can use ‘&’ operator to retrieve that actual stack address!

```
mike@Michaels-MacBook-Air 2022_corecpp % ./prog
0x16d5ff988
```

Address (in Hex)	Value
0x16d5ff988	42
...	
...	
...	
...	
...	
...	
...	
...	
...	Heap
...	
...	Data
...	
0x10000000	Code

Stack Allocation With Multiple Variables (1/4)

```
1 // g++ -g -std=c++20 stackmemory2.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     int y= 42;
7     std::cout << &x << '\n';
8     std::cout << &y << '\n';
9
10    return 0;
11 }
```

- Here's another example showing what happens when you allocate multiple variables.
 - The stack grows downward in the order of the allocations.

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	Data
...	
0x10000000	Code

Stack Allocation With Multiple Variables (2/4)

```
1 // g++ -g -std=c++20 stackmemory2.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     int y= 42;
7     std::cout << &x << '\n';
8     std::cout << &y << '\n';
9
10    return 0;
11 }
```

Address (in Hex)	Value
0x16d7f38d8	(x) 42
0x16d7f38d4	Stack
...	
...	
...	
...	
...	
...	
...	Heap
...	
...	
...	Data
...	
0x10000000	Code

- Here's another example showing what happens when you allocate multiple variables.
 - The stack grows downward in the order of the allocations.

Stack Allocation With Multiple Variables (3/4)

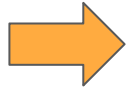
```
1 // g++ -g -std=c++20 stackmemory2.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     int y= 42;
7     std::cout << &x << '\n';
8     std::cout << &y << '\n';
9
10    return 0;
11 }
```

Address (in Hex)	Value
0x16d7f38d8	(x) 42
0x16d7f38d4	(y) 42
...	
...	
...	
...	
...	
...	
...	
0x10000000	
	Heap
	Data
	Code

- Here's another example showing what happens when you allocate multiple variables.
 - The stack grows downward in the order of the allocations.

Stack Allocation With Multiple Variables (4/4)

```
1 // g++ -g -std=c++20 stackmemory2.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     int y= 42;
7     std::cout << &x << '\n';
8     std::cout << &y << '\n';
9
10    return 0;
11 }
```



- Observe the memory addresses growing downward

```
mike@Michaels-MacBook-Air 2022_corecpp % ./prog
0x16d7f38d8
0x16d7f38d4
```

Address (in Hex)	Value
0x16d7f38d8	(x) 42
0x16d7f38d4	(y) 42
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
0x10000000	

Diagram illustrating memory layout and stack growth:

- The top section (blue) represents the stack, where memory addresses grow downward. It contains variables (x) and (y) both holding the value 42.
- The middle section (green) is labeled "Heap".
- The bottom section (yellow) is labeled "Data".
- The bottom-most section (light blue) is labeled "Code" and contains the address 0x10000000.
- A red arrow points downward from the stack area, indicating the direction of stack growth.

Automatic Memory Management -- Local Variables Popped off stack (1/5)

```
1 // g++ -g -std=c++20 stackmemory2.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     int y= 42;
7     std::cout << &x << '\n';
8     std::cout << &y << '\n';
9
10    return 0;
11 }
```



Now when we reach the end of our current scope

- Anything allocated on the stack within the 'block scope' (i.e. current stack frame) will be 'popped' off the stack
- This is effectively done by moving a 'stack pointer' to the next available location to overwrite.

Address (in Hex)	Value
0x16d7f38d8	(x) 42
0x16d7f38d4	(y) 42
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	Heap
...	
...	Data
...	
0x10000000	Code

Automatic Memory Management -- Local Variables Popped off stack (2/5)

```
1 // g++ -g -std=c++20 stackmemory2.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     int y= 42;
7     std::cout << &x << '\n';
8     std::cout << &y << '\n';
9
10    return 0;
11 }
```

Address (in Hex)	Value
0x16d7f38d8	(x) 42
0x16d7f38d4	(y) 42
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
0x10000000	

Stack Pointer

Heap

Data

Code

Now when we reach the end of our current scope

- Anything allocated on the stack within the 'block scope' (i.e. current stack frame) will be 'popped' off the stack
- This is effectively done by moving a 'stack pointer' to the next available location to overwrite.


```
1 // g++ -g -std=c++20 stackmemory2.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     int y= 42;
7     std::cout << &x << '\n';
8     std::cout << &y << '\n';
9
10    return 0;
11 }
```

Address (in Hex)	Value
0x16d7f38d8	(x) 42
	???
...	
...	
...	
...	
...	
...	
...	
...	
...	
0x10000000	

Stack Pointer

Heap

Data

Code

Automatic Memory Management -- Local Variables Popped off stack (4/5)

```
1 // g++ -g -std=c++20 stackmemory2.cpp -o prog
2 #include <iostream>
3 int main(){
4
5     int x= 42;
6     int y= 42;
7     std::cout << &x << '\n';
8     std::cout << &y << '\n';
9
10    return 0;
11 }
```

Address (in Hex)

Value

Stack Pointer

0x16d7f38d4

???

???

...

...

...

...

...

...

...

...

0x10000000

Heap

Data

Code

Now when we reach the end of our current scope

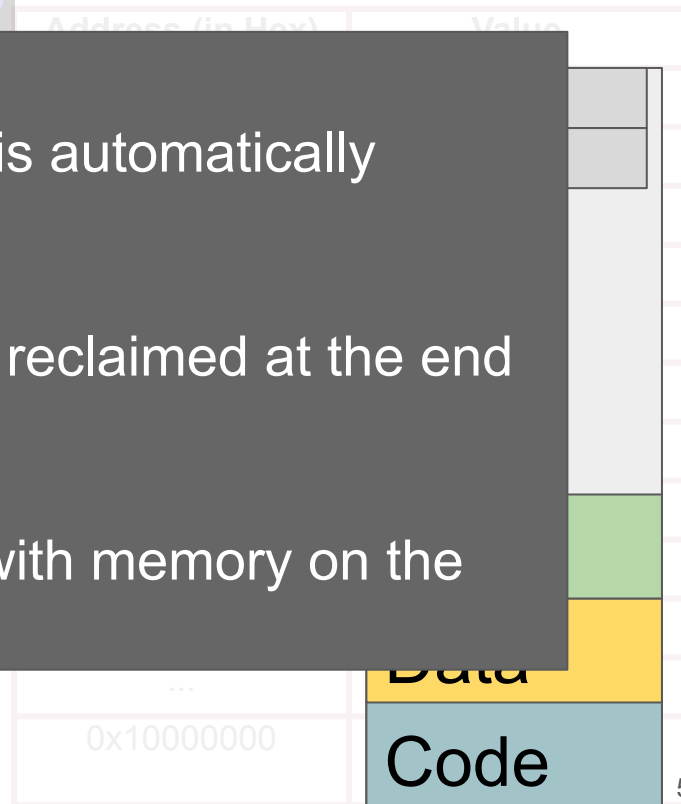
- Anything allocated on the stack within the 'block scope' (i.e. current stack frame) will be 'popped' off the stack
- This is effectively done by moving a 'stack pointer' to the next available location to overwrite.

Automatic Memory Management -- Local Variables Popped off stack (5/5)

```
1 // g++ -g -std=c++20 stackmemory2.cpp -o prog
```

So at this point:

- We understand that stack memory is automatically managed for us
 - Memory is placed on the stack
 - Stack allocated memory will be reclaimed at the end of its block scope
- Pretty simple model!
 - (And for performance working with memory on the stack is fast!)



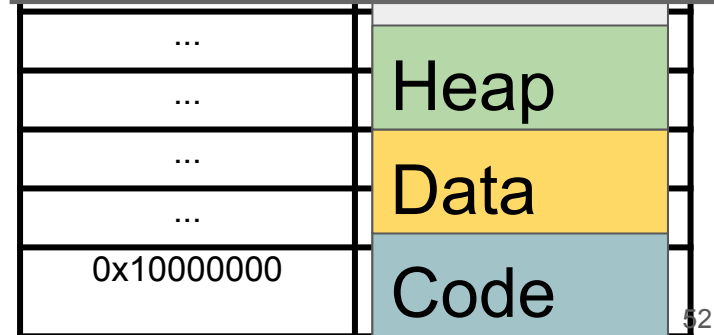
```

1 // @file alloca_example.c
2 #include <stdio.h> // printf
3 #include <stdlib.h> // alloca
4 #include <string.h> // strcpy
5 #include <ctype.h> // toupper
6
7 void PrintUpperCase(char* nullTerminatedCharArray){
8
9     // Allocate on the stack for a new mutated string
10    // Add +1 for the null terminated character
11    char* printName = (char*) alloca(strlen(nullTerminatedCharArray)+1);
12    strcpy(printName, nullTerminatedCharArray);
13
14    // Set the individual characters to uppercase
15    size_t i=0;
16    while(printName[i] != '\0'){
17        printName[i] = toupper(nullTerminatedCharArray[i]);
18        ++i;
19    }
20
21    // Print all at once the uppercase string with an endlime
22    printf("%s\n", printName);
23 }
24
25 // Entry point of program
26 int main(){
27
28    // String literal allocated in static storage with \0 terminator
29    PrintUpperCase("some string");
30
31    return 0;
32 }

```

Quick Note: We can use 'alloca' to also explicitly allocate on the stack as well.

'alloca' would be our first example a memory allocator -- this one with a special purpose to allocate memory on the stack



Question for Audience: (1/2)

How do folks feel about this huge memory allocation on the stack?

Hint which way does stack grow (in our example)?

```
1 // @file alloca_big.c
2 #include <stdlib.h> // alloc
3
4
5 // Entry point of program
6 int main(){
7
8     char* buffer = (char*)alloca(100000000);
9
10    return 0;
11 }
```

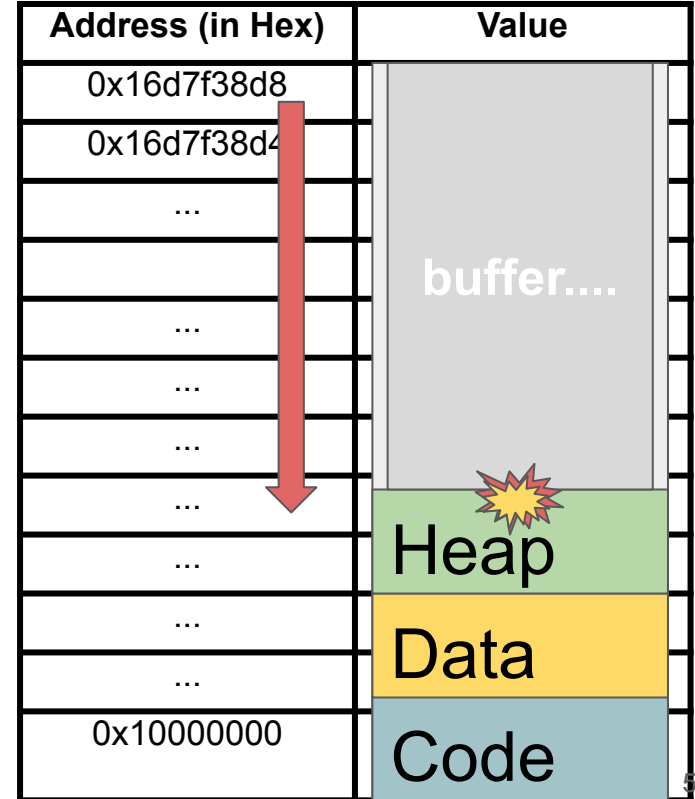
Address (in Hex)	Value
0x16d7f38d8	buffer....
0x16d7f38d4	Stack
...	
...	
...	
...	
...	
...	Heap
...	Data
...	
0x10000000	Code

Question for Audience: (2/2)

Answer: Our stack is going to overflow -- overwriting other segments of memory -- this is bad.

Our stack is a 'fixed size'

```
1 // @file alloca_big.c
2 #include <stdlib.h> // alloca
3
4
5 // Entry point of program
6 int main(){
7
8     char* buffer = (char*)alloca(100000000);
9
10    return 0;
11 }
```



Audience Poll:
How much data
do you see here?

- a.) A lot (More than 7MB)
- b.) Very little



Nanite | Inside Unreal

<https://youtu.be/TMorJX3Nj6U?t=5255>

Audience Poll:
How much data
do you see here?

- a.) A lot (More than 7mb)
- b.) Very little



Nanite | Inside Unreal

<https://youtu.be/TMorJX3Nj6U?t=5255>

Humor me here -- there's 'a lot' of data on cpu and gpu (and that is probably an understatement!)


```
mike@Michaels-MacBook-Air 2022_Handmade_Seattle % limit
cputime      unlimited
filesize     unlimited
datasize     unlimited
stacksize    7MB
coredumpsize 0kB
addressspace unlimited
memorylocked unlimited
maxproc      2666
descriptors  2560
```

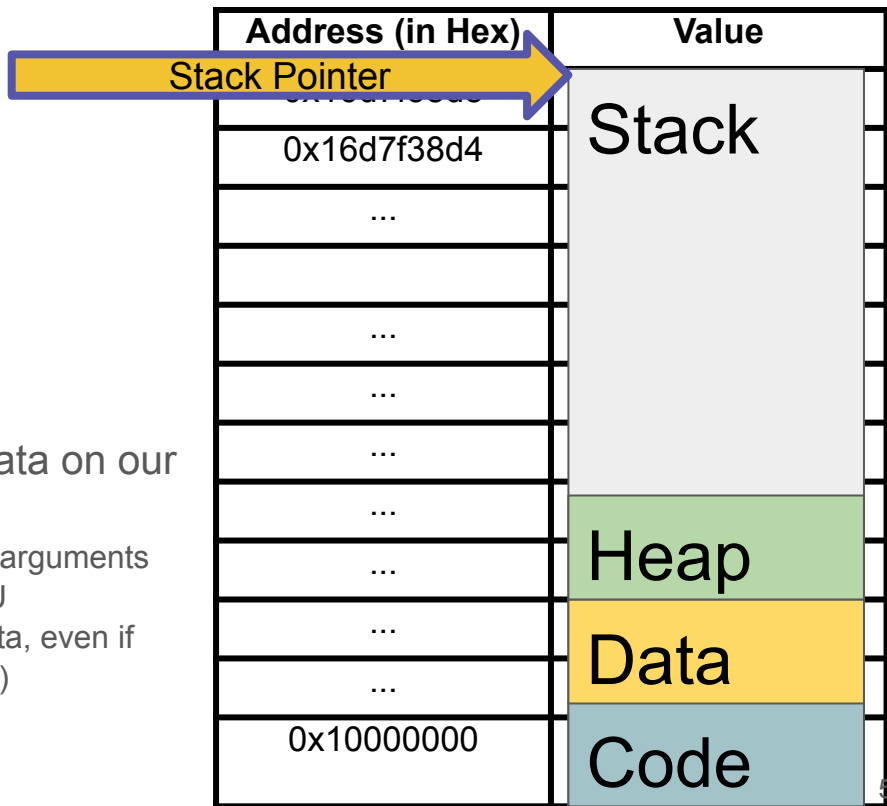
Nanite | In
<https://youtu.be/3Nj6U?t=5255>

My stack on my machine is only 7 mb

Can we store all this data on the stack? (1/2)

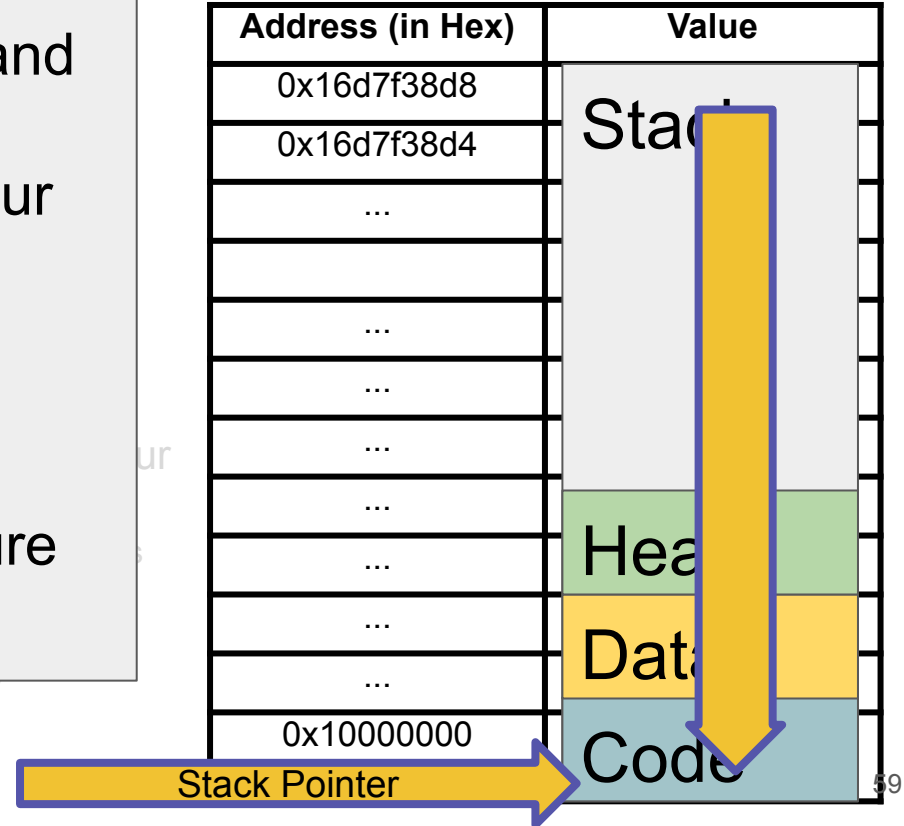


- So my question is, can we store all of this data on our 'stack'?
 - Let's assume this is several gigabytes of data for arguments sake, and that a good chunk of that is on the CPU
 - (Dear experts :) -- just assume there is a lot of data, even if this is a more CPU/GPU memory bound example)



Can we store all this data on the stack? (2/2)

- The answer is **no**.
- Our stack grows downward, and would overflow and overwrite other sensitive segments of our running processes memory
 - So we have another mechanism for 'large allocations' or otherwise allocations we cannot figure out at compile-time



Heap Memory

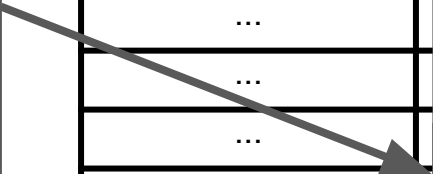
For 'large' and/or 'long lived' Memory
Allocations

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	
...	
...	
...	
	Heap
	Data
0x10000000	Code

Heap Memory

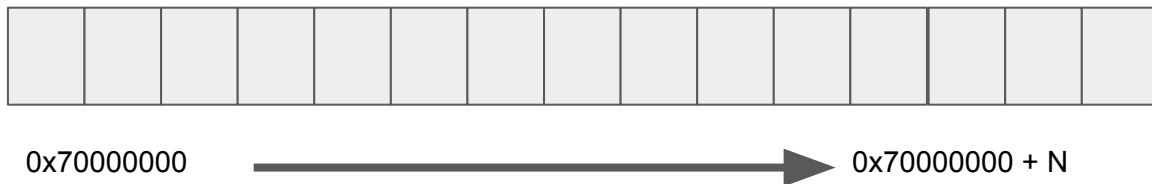
- Heap memory is memory that we allocate at 'run-time'
- 'The heap' is some data structure that stores our successful requests for memory.
 - In order to use that memory, we need a 'pointer' which stores that address of memory
 - **We'll also need some bookkeeping mechanism which we'll talk about later**

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	
...	
...	Data
...	
0x10000000	Code



Heap Visualization (1/9)

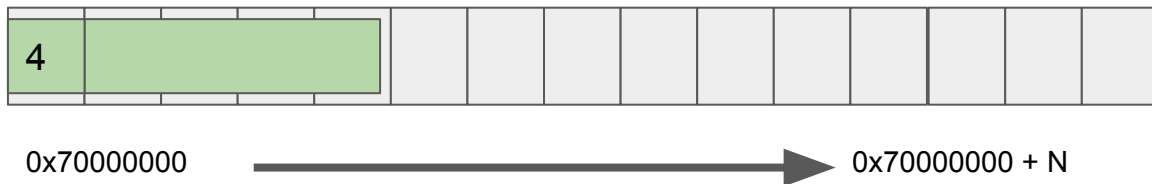
- The heap data structure might look something like this
- A 'large collection of bytes'



Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	Data
...	Code
0x10000000	

Heap Visualization (2/9)

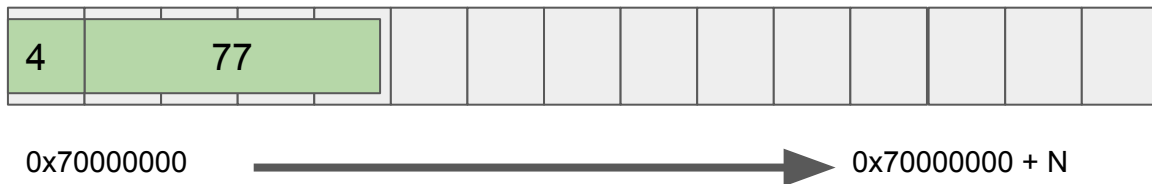
- We 'manually' allocate memory using **new**
 - e.g. `int* data = new int;`



Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	Data
...	
0x10000000	Code

Heap Visualization (3/9)

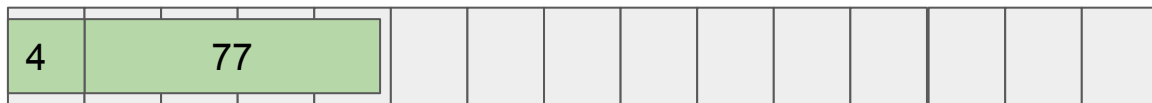
- We 'manually' allocate memory using **new**
 - e.g. `int* data = new int;`
 - And assign with: `*data = 77;`



Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	Data
...	
0x10000000	Code

Heap Visualization (4/9)

- We 'manually' allocate memory using **new**
 - e.g. `int* data = new int;`
 - And assign with: `*data = 77;`



0x70000000  0x70000000 + N

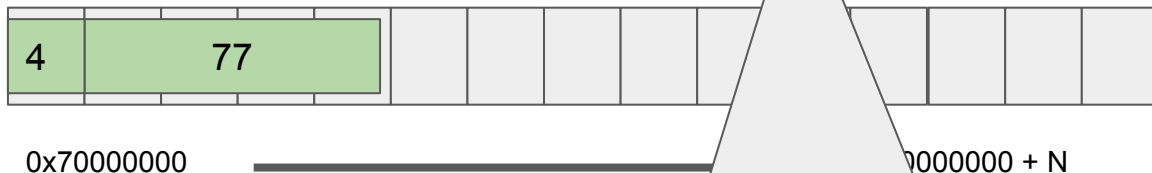
(Aside)

- The '4' at the start does some bookkeeping in our heap structure to tell us how big the allocation was.
- The actual data we write an integer to (again we need 4 bytes) could come after the little 'header' labeled 4 that does bookkeeping for us.

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	
...	Data
...	Code
0x10000000	

Heap Visualization (5/9)

- We ‘manually’ allocate memory using **new**
 - e.g. `int* data = new int;`
 - And assign with: `*data = 77;`

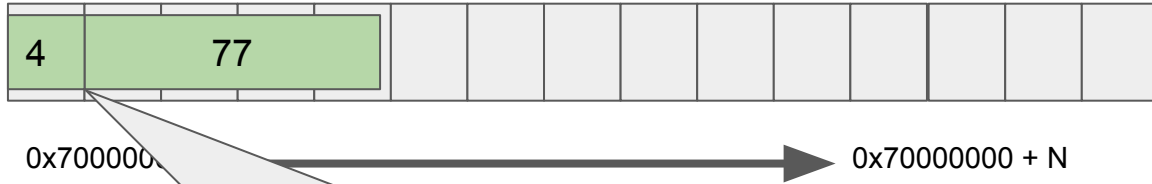


- So ‘new int’ returned an address to some new memory.
- In order to store an address, we need a special data type, known as a ‘pointer’ (i.e. `int*`)

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	Data
...	
0x10000000	Code

Heap Visualization (6/9)

- We 'manually' allocate memory using **new**
 - e.g. `int* data = new int;`
 - And assign with: `*data = 77;`

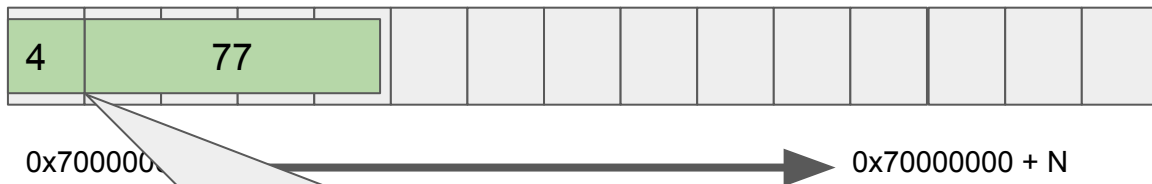


- Now before our program terminates, we're going to need to **delete** the memory 'manually' that we have allocated.

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	Data
...	
0x10000000	Code

Heap Visualization (7/9)

- Heap memory is usually meant to be long lived
 - (It gets its own section for that reason)

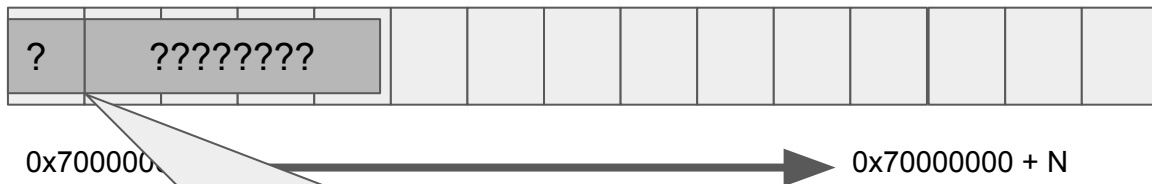


- So we use **delete data**; to reclaim the memory in our process.

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	
...	
...	Data
...	Code
0x10000000	

Heap Visualization (8/9)

- Heap memory is usually meant to be long lived
 - (It gets its own section for that reason)

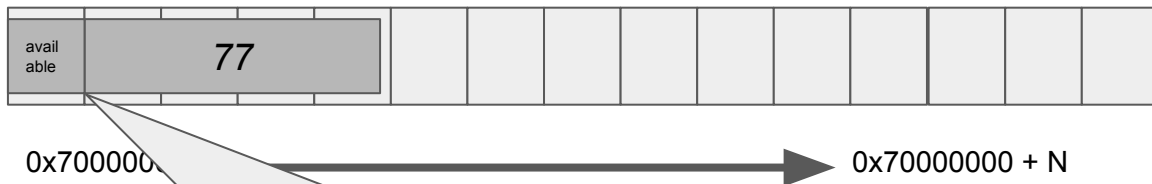


- So we'll use **delete data**; to reclaim the memory in our process.
 - Now our previous block of memory in the heap can be repurposed.

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	
...	
...	Data
...	
0x10000000	Code

Heap Visualization (9/9)

- Heap memory is usually meant to be long lived
 - (It gets its own section for that reason)



- In practice our memory is marked free, and likely holds the previous contents

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	
...	
...	Heap
...	
...	
...	Data
...	
0x10000000	Code

(Quick Recap of our Quick Review)

Stack and Heap

Stack and Heap (1/2)

- **Stack**
 - Fast allocations for local variables
 - memory automatically reclaimed
 - stack size is fixed, so smaller amount of space
- **Heap**
 - Used for larger allocations
 - Used for long-lived memory
 - And we have to manage the heap in an explicit allocator (or otherwise rely on infrastructure like a garbage collector to reclaim memory)
 - And dynamic memory allocation is slow.

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	Heap
...	
...	
...	
...	Data
...	
0x10000000	Code

Stack and Heap (2/2)

- Stack
 - Fast allocations for local variables
 - memory automatically reclaimed
 - stack size is fixed, so smaller amount of space
- Heap
 - Used for larger allocations
 - Used for long-lived memory
 - And we have to manage the heap in an explicit allocator (or otherwise rely on infrastructure like a garbage collector to reclaim memory)
 - **And dynamic memory allocation is slow.**
 - (next slide)

Address (in Hex)	Value
0x16d7f38d8	Stack
0x16d7f38d4	
...	
...	Heap
...	
...	
...	
...	Data
...	
0x10000000	Code

Question to Audience: Wait, why is 'dynamic memory allocation' (i.e. new or malloc) slow -- or at least slow relative to the stack?

Why is Dynamic Memory Allocation Considered **Slow**?

Specifically with Global General Purpose Heap Allocators?

Reason # 1 -- malloc/new are general purpose allocators

- 1.) The allocator is designed to handle and **book keep allocations of any size.**
 - a. General purpose memory allocators are designed to do 'well enough' for most applications.
 - b. That means allocating 1 byte, 72 bytes, 1mb, or 2.1gb should be supported
 - c. Handling such a variety of scenarios can add a lot of overhead per call to 'malloc' or 'new' for 'finding memory'

```
111 void* operator new(std::size_t) _GLIBCXX_THROW (std::bad_alloc)
112   __attribute__((__externally_visible__));
113 void* operator new[](std::size_t) _GLIBCXX_THROW (std::bad_alloc)
114   __attribute__((__externally_visible__));
115 void operator delete(void*) _GLIBCXX_USE_NOEXCEPT
116   __attribute__((__externally_visible__));
117 void operator delete[](void*) _GLIBCXX_USE_NOEXCEPT
118   __attribute__((__externally_visible__));
119 void* operator new(std::size_t, const std::nothrow_t&) _GLIBCXX_USE_NOEXCEPT
120   __attribute__((__externally_visible__));
121 void* operator new[](std::size_t, const std::nothrow_t&) _GLIBCXX_USE_NOEXCEPT
122   __attribute__((__externally_visible__));
123 void operator delete(void*, const std::nothrow_t&) _GLIBCXX_USE_NOEXCEPT
124   __attribute__((__externally_visible__));
125 void operator delete[](void*, const std::nothrow_t&) _GLIBCXX_USE_NOEXCEPT
126   __attribute__((__externally_visible__));
127
128 // Default placement versions of operator new.
129 inline void* operator new(std::size_t, void* __p) _GLIBCXX_USE_NOEXCEPT
130 { return __p; }
131 inline void* operator new[](std::size_t, void* __p) _GLIBCXX_USE_NOEXCEPT
132 { return __p; }
133
134 // Default placement versions of operator delete.
135 inline void operator delete (void*, void*) _GLIBCXX_USE_NOEXCEPT { }
136 inline void operator delete[](void*, void*) _GLIBCXX_USE_NOEXCEPT { }
137 //@}
138 } // extern "C++"
139
```

The 'new' header file in gcc

Reason# 2 - Context Switching and OS finding resources (1/7)

2.) When you allocate memory,
a context switch takes place

- a. This means the kernel (which is running alongside your program in the operating system) takes over.
 - i. It then grants you the required memory (if you have have enough) and then you return to executing your program.
 - ii. Then you context switch back

Reason# 2 - Context Switching and OS finding resources (2/7)

2.) When you allocate memory, a context switch takes place

- a. This means the kernel (which is running alongside your program in the operating system) takes over.
 - i. It then grants you the required memory (if you have have enough) and then you return to executing your program.
 - ii. Then you context switch back

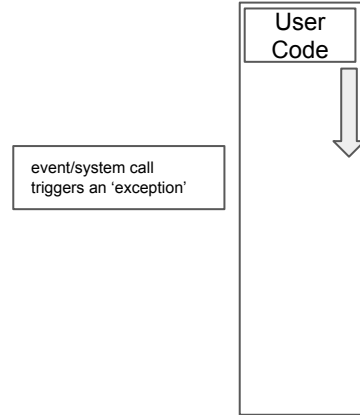


Our process is executing

Reason# 2 - Context Switching and OS finding resources (3/7)

2.) When you allocate memory, a context switch takes place

- a. This means the kernel (which is running alongside your program in the operating system) takes over.
 - i. It then grants you the required memory (if you have have enough) and then you return to executing your program.
 - ii. Then you context switch back

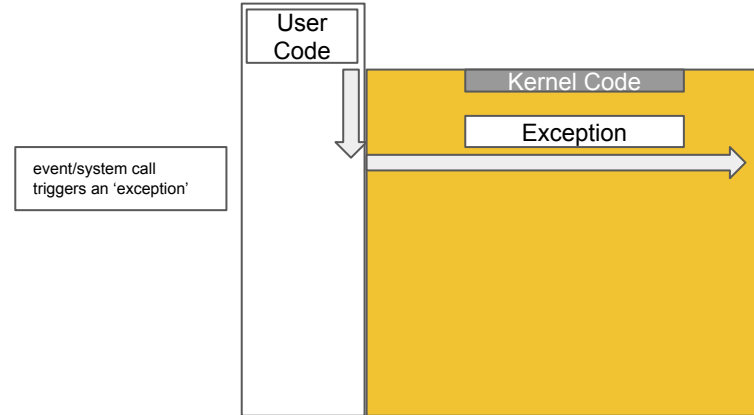


Then the user calls
'malloc'

Reason# 2 - Context Switching and OS finding resources (4/7)

2.) When you allocate memory, a context switch takes place

- a. This means the kernel (which is running alongside your program in the operating system) takes over.
 - i. It then grants you the required memory (if you have have enough) and then you return to executing your program.
 - ii. Then you context switch back



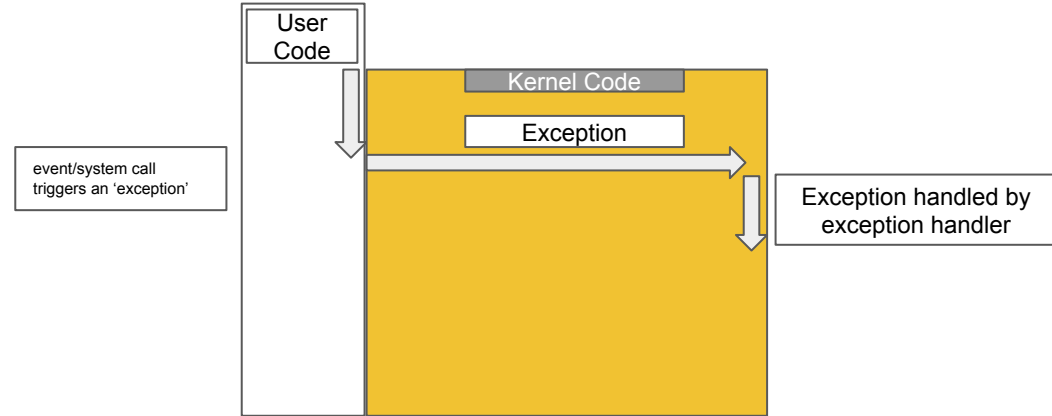
An exception is called (OS level exception on a system call) that transfers control to the kernel.

Registers saved in user process, perhaps other processes also run in-between.

Reason# 2 - Context Switching and OS finding resources (5/7)

2.) When you allocate memory, a context switch takes place

- a. This means the kernel (which is running alongside your program in the operating system) takes over.
 - i. It then grants you the required memory (if you have have enough) and then you return to executing your program.
 - ii. Then you context switch back

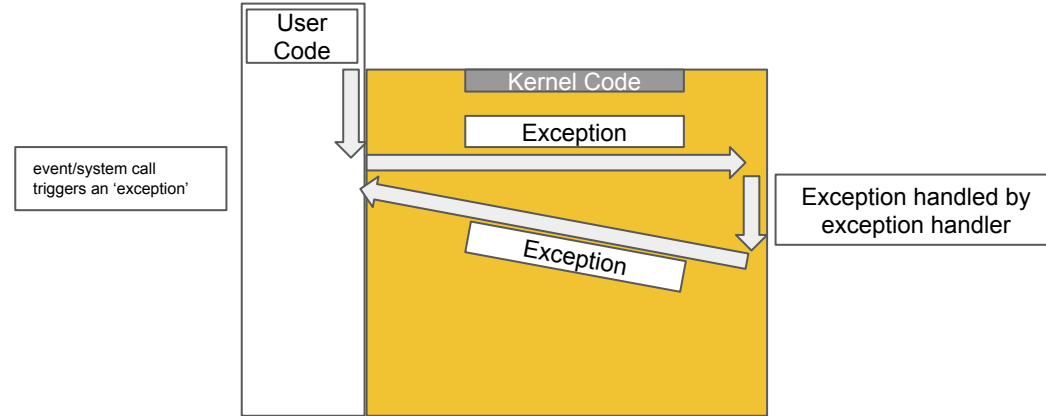


Our Operating System finds some memory that it can return (in increments of a 'page size')

Reason# 2 - Context Switching and OS finding resources (6/7)

2.) When you allocate memory, a context switch takes place

- a. This means the kernel (which is running alongside your program in the operating system) takes over.
 - i. It then grants you the required memory (if you have have enough) and then you return to executing your program.
 - ii. Then you context switch back

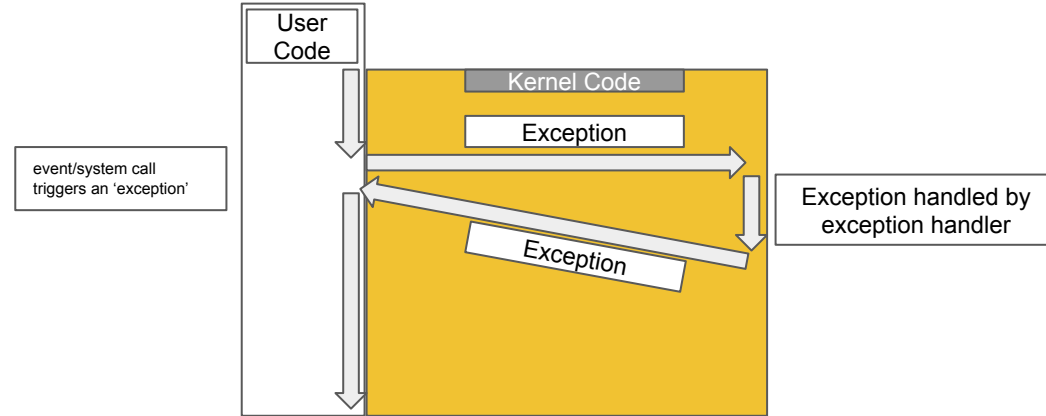


Our Operating System returns control to the process, restores registers to our code.

Reason# 2 - Context Switching and OS finding resources (7/7)

2.) When you allocate memory, a context switch takes place

- a. This means the kernel (which is running alongside your program in the operating system) takes over.
 - i. It then grants you the required memory (if you have have enough) and then you return to executing your program.
 - ii. Then you context switch back



Our program can now proceed.

(Aside) Context Switch Cost

- Note: Anecdotally context switches take the order of 100s-1000s of cycles vs regular instructions taking 1-100 cycles.
 - (You can try using `rdtsc()` to try to measure clock ticks after an allocation)

A Simple Heap Allocator

To understand some of the challenges of
memory allocation

Let's build an **Explicit allocator** (rather than Implicit allocators)

- Our choice of language often defaults us into an allocation strategy
- **Explicit Memory Allocator**
 - The application (i.e. you the programmer) is responsible for allocating and freeing memory
 - This is what we do in C with `malloc` and `free`
 - (or equivalently in C++ with `new` and `delete/delete[]`)
- **Implicit Memory Allocator**
 - The application(program) allocates, but does not free memory
 - A [garbage collector](#) instead frees the memory for us.
 - e.g. The Java programming language has different garbage collectors to help us
- **Note:** More language these days may offer a combination
 - (e.g. [DLang](#) is implicit by default, but allows you to mark code `@nogc` and perform manual memory allocation)
- **Note:** And of course, you could roll your own garbage collector for C or C++ if you wanted, it's just not the default



Implementation

Building a Memory Allocator

mymalloc.c (1/2)

- Given in this example is “mymalloc.c”
 - You will notice, some of it is filled in.
- We are essentially going to ‘override’ the malloc call with our own ‘mymalloc’, ‘myfree’, ‘mycalloc’, etc..

```
#include <stdio.h> // Any other headers we need here
#include <malloc.h> // We bring in the old malloc function
// NOTE: You should remove malloc.h, and not include <stdlib.h> in your final implementation.

void* mymalloc(size_t s){

    void* p = (void*)malloc(s);    // In your solution no calls to malloc should be made!
                                   // Determine how you will request memory :)

    if(!p){
        // We we are out of memory
        // if we get NULL back from malloc
    }
    printf("malloc %zu bytes\n",s);

    return p;
}

void* mycalloc(size_t nmemb, size_t s){

    void* p = (void*)calloc(nmemb,s);    // In your solution no calls to calloc should be made!
                                           // Determine how you will request memory :)

    if(!p){
        // We we are out of memory
        // if we get NULL back from malloc
    }
    printf("calloc %zu bytes\n",s);

    return p;
}

void myfree(void *ptr){
    printf("Freed some memory\n");
    free(ptr);
}
```

mymalloc.c (2/2)

- Here's the blocks of code where you can replace malloc.
- **Note:** I'm calling the original malloc/calloc/free -- but we'll want to replace that with something else!
- **Thus, we need a tool for extending the heap.**

```
#include <stdio.h> // Any other headers we need here
#include <malloc.h> // We bring in the old malloc function
// NOTE: You should remove malloc.h, and not include <stdlib.h> in your final implementation.
```

```
void* mymalloc(size_t s){
```

```
    void* p = (void*)malloc(s);    // In your solution no calls to malloc should be made!
                                    // Determine how you will request memory :)

    if(!p){
        // We we are out of memory
        // if we get NULL back from malloc
    }

    printf("malloc %zu bytes\n",s);

    return p;
```

```
}
```

```
void* mycalloc(size_t nmemb, size_t s){
```

```
    void* p = (void*)calloc(nmemb,s);    // In your solution no calls to calloc should be made!
                                            // Determine how you will request memory :)

    if(!p){
        // We we are out of memory
        // if we get NULL back from malloc
    }

    printf("calloc %zu bytes\n",s);

    return p;
```

```
}
```

```
void myfree(void *ptr){
```

```
    printf("Freed some memory\n");
    free(ptr);
```

```
}
```


(Aside) mymalloc.h - interpositioning

- You can either use your allocation functions, or try to use some compile-time or link-time interpositioning technique to replace malloc.
- malloc.h shows how we are defining 'malloc' to actually mean
 - “Hey compiler, you know that code we wrote with 'malloc'?”
 - “Please replace all malloc's with mymalloc, so I can test other programs that used the malloc.h allocator.”

```
// This header is redefining every call to 'malloc' to our implementation
// 'mymalloc'
// Again, this is a simple textual replacement of the code by the preprocessor
#define malloc(size) mymalloc(size)
#define calloc(nmemb, size) mycalloc(nmemb, size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void *mycalloc(size_t nmemb, size_t size);
void myfree(void *ptr);
```

Ways to request memory for our allocator (on linux) (1/2)

System Calls

- `sbrk`
 - Use internally by allocators to grow or shrink the heap
 - This will be handy for implementing our own memory allocator!
- `mmap`
 - Creates a new mapping in virtual address space (in page size increments) of calling process.

Ways to request memory for our allocator (on linux) (2/2)

System Calls



We'll be using sbrk for our first allocator -- it's good for our first allocator

- **sbrk**
 - Use internally by allocators to grow or shrink the heap
 - This will be handy for implementing our own memory allocator!
- **mmap**
 - Creates a new mapping in virtual address space (in page size increments) of calling process.

sbrk system call

- The 'sbrk' command is the system call for changing the size of the heap segment.
 - (i.e. how we can extend the heap)
- Malloc is built on top of system commands like sbrk (and [mmap](#))

```
BRK(2)                                Linux Programmer's Manual                                BRK(2)

NAME
    brk, sbrk - change data segment size

SYNOPSIS
    #include <unistd.h>

    int brk(void *addr);

    void *sbrk(intptr_t increment);
```

sbrk example (1/4)

- Here is an example of extending the heap 4 bytes
 - Note:** sbrk(0) - returns the address of the top of the heap

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(){
5
6     // Print out the top of the heap for a process.
7     // This is the first byte we can 'grab' to store something in.
8     // We pass in '0' as an argument into sbrk.
9     void* top = sbrk(0);
10    printf("top of heap: %p\n",top);
11
12    // Extend heap 4 bytes
13    char* bytes = sbrk(4);
14    printf("new top of heap: %p\n",sbrk(0));
15
16    // Set our bytes to some data
17    bytes[0] = 'h';
18    bytes[1] = 'i';
19    bytes[2] = '\n';
20    bytes[3] = '\0';
21    printf("Stored in heap %s",bytes);
22
23
24    return 0;
25 }
```

sbrk example (2/4)

- Here is an example of extending the heap 4 bytes
 - Note:** sbrk(0) - returns the address of the top of the heap

Question to Audience: Should we track our allocations and why? Or not (and why?)

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(){
5
6     // Print out the top of the heap for a process.
7     // This is the first byte we can 'grab' to store something in.
8     // We pass in '0' as an argument into sbrk.
9     void* top = sbrk(0);
10    printf("top of heap: %p\n",top);
11
12    // Extend heap 4 bytes
13    char* bytes = sbrk(4);
14    printf("new top of heap: %p\n",sbrk(0));
15
16    // Set our bytes to some data
17    bytes[0] = 'h';
18    bytes[1] = 'i';
19    bytes[2] = '\n';
20    bytes[3] = '\0';
21    printf("Stored in heap %s",bytes);
22
23
24    return 0;
25 }
```

sbrk example (3/4)

- Here is an example of extending the heap 4 bytes
 - Note:** sbrk(0) - returns the address of the top of the heap

Question to Audience: Should we track our allocations and why? Or not (and why?)

For folks answering **yes**

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(){
5
6     // Print out the top of the heap for a process.
7     // This is the first byte we can 'grab' to store something in.
8     // We pass in '0' as an argument into sbrk.
9     void* top = sbrk(0);
10    printf("top of heap: %p\n",top);
11
12    // Extend heap 4 bytes
13    char* bytes = sbrk(4);
14    printf("new top of heap: %p\n",sbrk(0));
15
16    // Set our bytes to some data
17    bytes[0] = 'h';
18    bytes[1] = 'i';
19    bytes[2] = '\n';
20    bytes[3] = '\0';
21    printf("Stored in heap %s",bytes);
22
23
24    return 0;
25 }
```

We need to keep track of this '4' byte allocation so we can free our memory later, and know to mark '4' bytes as free.

sbrk example (4/4)

- Here is an example of extending the heap 4 bytes
 - Note:** sbrk(0) - returns the address of the top of the heap

Question to Audience: Should we track our allocations and why? Or not (and why?)

For folks answering **no**

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(){
5
6     // Print out the top of the heap for a process.
7     // This is the first byte we can 'grab' to store something in.
8     // We pass in '0' as an argument into sbrk.
9     void* top = sbrk(0);
10    printf("top of heap: %p\n",top);
11
12    // Extend heap 4 bytes
13    char* bytes = sbrk(4);
14    printf("new top of heap: %p\n",sbrk(0));
15
16    // Set our bytes to some data
17    bytes[0] = 'h';
18    bytes[1] = 'i';
19    bytes[2] = '\n';
20    bytes[3] = '\0';
21    printf("Stored in heap %s",bytes);
22
23
24    return 0;
25 }
```

This is a **'monotonic allocator'**. We never pay any cost to 'free' memory

Allocators so far (1/2)

1. **malloc/free (C), and new/delete (C++)**

- a. General purpose global allocators that we can allocate and free memory from

2. **alloca**

- a. Special purpose allocator that allows us to allocate memory on the stack
 - i. (Memory reclaimed when we leave scope)

3. **(heap-based) monotonic allocator**

- a. An allocator that allocates memory on the heap (perhaps using sbrk).
- b. We just allocate -- we never or rarely recycle or reclaim memory to use again, even when we're done with it.
 - i. See: https://en.cppreference.com/w/cpp/memory/monotonic_buffer_resource (C++17)
 - ii. Search 'bump allocator'
 - 1. Again idea is to never free, or maybe free all of the memory at once.

Allocators so far (2/2)



1. malloc/free (C), and new/delete (C++)

- a. General purpose global allocators that we can allocate

2. alloca

- a. Special purpose allocator that allows us to allocate memory
 - i. (Memory reclaimed when we leave scope)

3. (heap-based) **monotonic allocator**

- a. An allocator that allocates memory on the heap (permanently)
- b. We just allocate -- we never or rarely recycle or reclaim memory we're done with it.
 - i. See: https://en.cppreference.com/w/cpp/memory/monotonic_buffer_resource (C++17)
 - ii. Search 'bump allocator'
 - 1. Again idea is to never free, or maybe free all of the memory at once.

For some of you, this might be the right strategy! The talk is effectively done for you :)

For the rest of us, this is a useful strategy to know about, that we might combine with others.

Let's learn about bookkeeping now :)

Tracking Memory (i.e. Bookkeeping)

Allocating with `sbrk` was easy -- how do we keep track and free the right amount of memory?

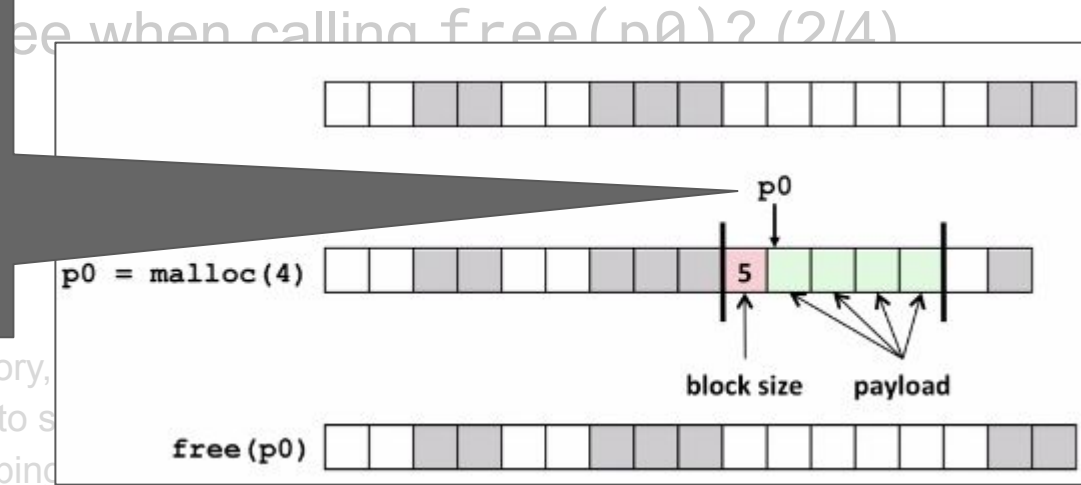
How much memory do we free when calling `free(p0)`? (1/4)

- In order to answer this, we need to do some bookkeeping
- That means creating a data structure to track how much memory we are using.
 - Note that when we allocate memory, we are actually going to allocate a **'block' + the size of the actual data** we want to store (and maybe more metadata as well).
 - The 'block' handles the book keeping

```
struct block{
    size_t size; // How many bytes beyond this block have been allocated in the heap
    block* next; // Where is the next block in your linked list
    int free; // Is this memory free?
    int debug; // (optional) Perhaps you can embed other information--remember, you are the boss!
};
```

Key idea: Malloc'ing 4 means '4'+sizeof(block)
Sometimes we call this the 'header' (See end of slide deck for extra slides)

- Note that when we allocate memory, **size of the actual data** we want to store is 4
- The 'block' handles the book keeping

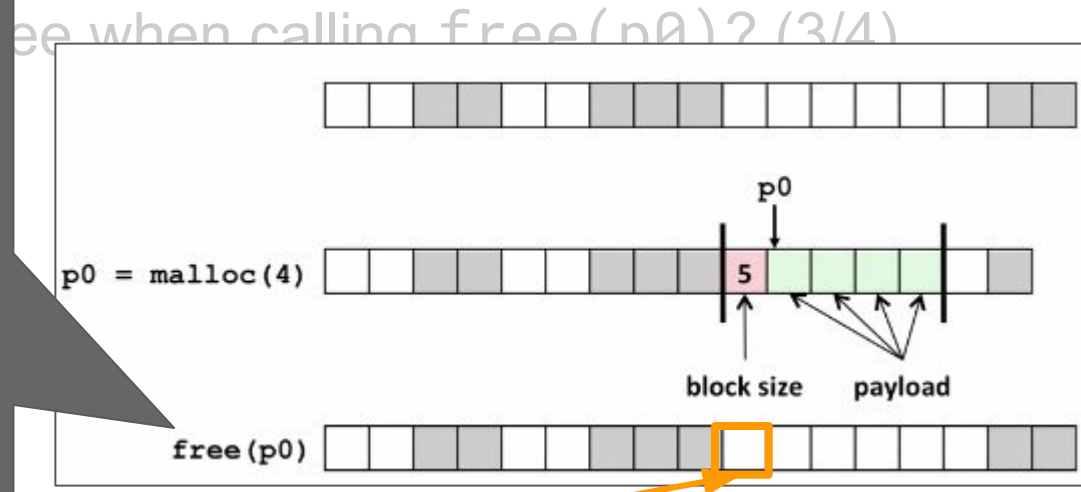


```
struct block{
    size_t size; // How many bytes beyond this block have been allocated in the heap
    block* next; // Where is the next block in your linked list
    int free; // Is this memory free?
    int debug; // (optional) Perhaps you can embed other information--remember, you are the boss!
};
```

So when we free memory, we look for the block in a data structure that holds all of the blocks--and flip a bit to free.

Now what this could mean, is traversing through an entire list everytime we want to free something!

What strategies do we otherwise have?

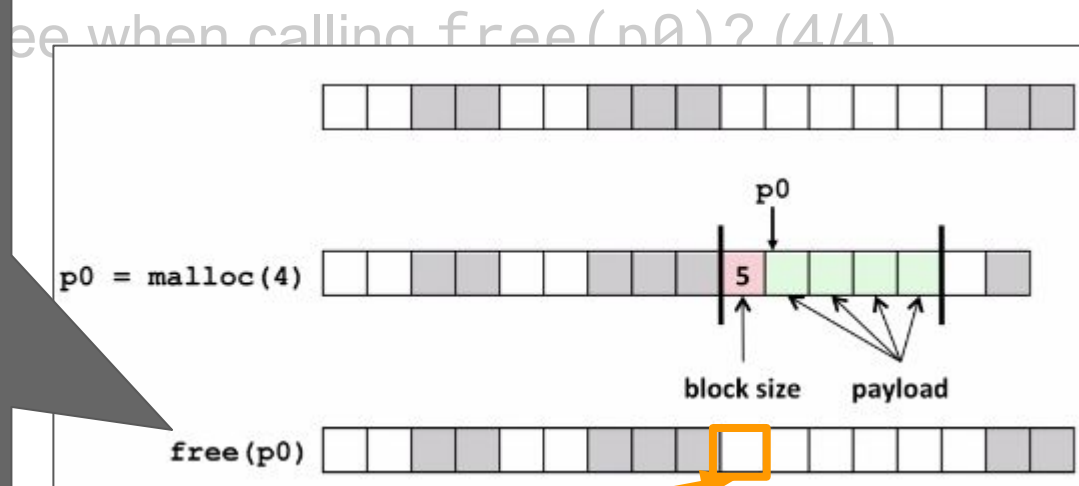


```
struct block{  
    size_t size; // How many bytes beyond this block have been allocated in the heap  
    block* next; // Where is the next block in your linked list  
    int free; // Is this memory free?  
    int debug; // (optional) Perhaps you can embed other information--remember, you are the boss!  
};
```

(Quick note)

'block size' in the visualization is actually bigger than the 1-byte box represented.

- (For convenience, it is represented as one box in some of the figures)



```
struct block{
    size_t size; // How many bytes beyond this block have been allocated in the heap
    block* next; // Where is the next block in your linked list
    int free; // Is this memory free?
    int debug; // (optional) Perhaps you can embed other information--remember, you are the boss!
};
```

Keeping Track of Memory

Data Structures and strategies

~~Keeping track of memory~~ | Strategy 1 - **monotonic allocator** (1/6)

- Simply allocate memory as you need
 - No need for a block structure even

~~Keeping track of memory~~ | Strategy 1 - monotonic allocator (2/6)

- Simply allocate memory as you need
 - No need for a block structure even
 - `char* A = malloc(sizeof(char));`



'A'

~~Keeping track of memory~~ | Strategy 1 - monotonic allocator (3/6)

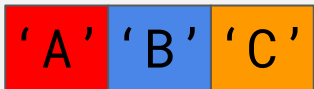
- Simply allocate memory as you need
 - No need for a block structure even
 - `char* A = malloc(sizeof(char));`
 - `char* B = malloc(sizeof(char));`

A diagram illustrating memory allocation. It shows a large, light gray rectangular area representing memory. In the top-left corner of this area, there are two adjacent colored squares: a red square on the left and a blue square on the right. The red square contains the character 'A' in black, and the blue square contains the character 'B' in black. This represents two separate memory blocks allocated sequentially.

'A' 'B'

~~Keeping track of memory~~ | Strategy 1 - monotonic allocator (4/6)

- Simply allocate memory as you need
 - No need for a block structure even
 - `char* A = malloc(sizeof(char));`
 - `char* B = malloc(sizeof(char));`
 - `char* C = malloc(sizeof(char));`



'A' 'B' 'C'

~~Keeping track of memory~~ | Strategy 1 - monotonic allocator (5/6)

- Simply allocate memory as you need

- No need for a block structure even

- `char* A = malloc(sizeof(char));`

- `char* B = malloc(sizeof(char));`

- `char* C = malloc(sizeof(char));`

- `short* s = malloc(sizeof(short));`

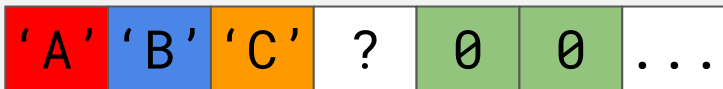
Note: we need to take into consideration fragmentation (coming up) and alignment, thus the gap.

[\[See wiki\]](#)



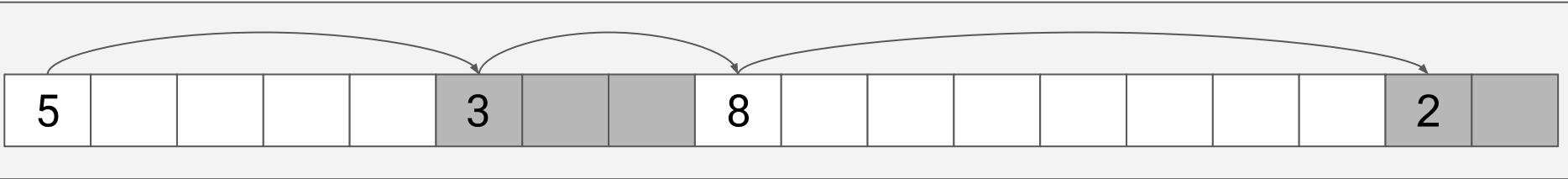
~~Keeping track of memory~~ | Strategy 1 - monotonic allocator (6/6)

- Simply allocate memory as you need
 - No need for a block structure even
 - `char* A = malloc(sizeof(char));`
 - `char* B = malloc(sizeof(char));`
 - `char* C = malloc(sizeof(char));`
 - `short* s = malloc(sizeof(short));`
- So just to be clear, we are not keeping track of anything here -- our strategy is keep track of nothing.



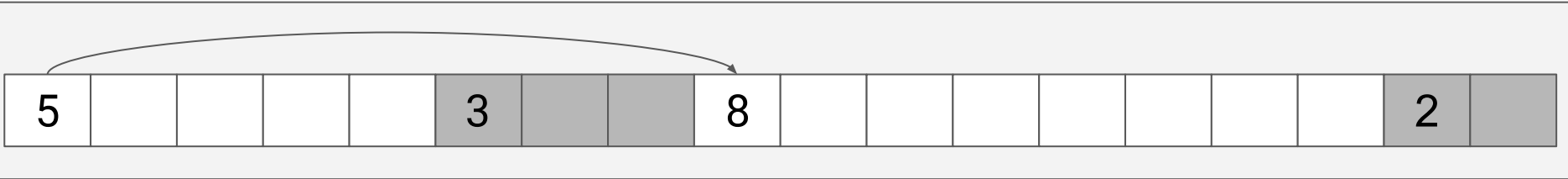
Keeping track of memory | Strategy 2 - **Implicit List**

- An **implicit list** keeps track of all of the blocks using length of allocation in the block structure
 - We have this 'emergent data structure' that forms a singly linked list for all of the memory that we have handed out.
 - Prior to doing a call to `sbrk` to extend the heap (or `mmap`), we can then check if a 'free' block (marked free in the *struct block*) and reuse that block.
 - When we 'free' memory, we simply retrieve the memory address of the data, subtract the block size, and mark in that block as free.



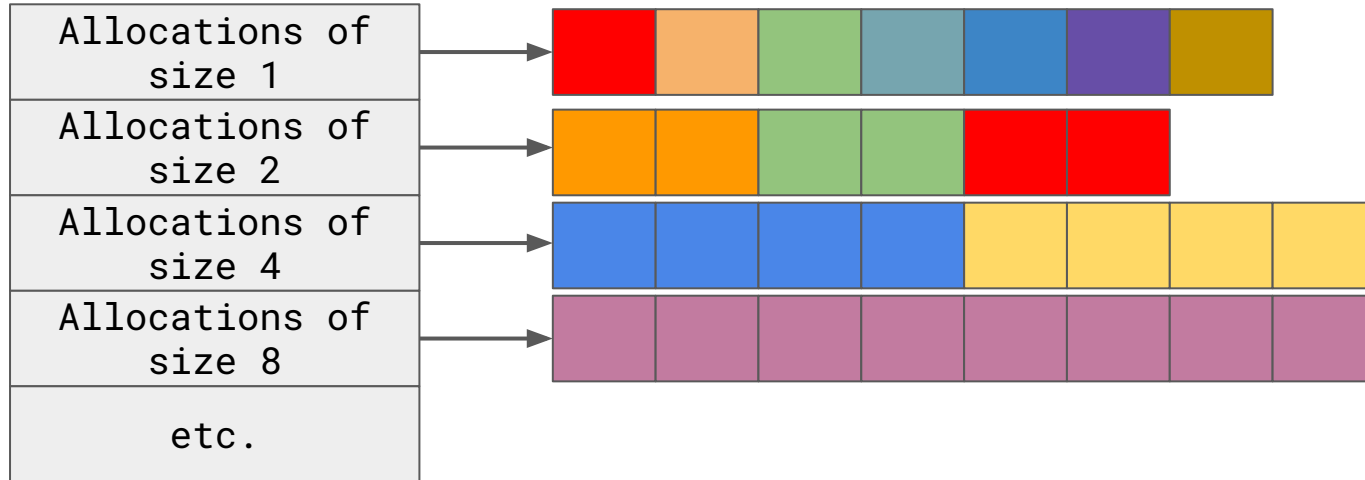
Keeping track of memory | Strategy 3 - **Explicit List**

- An **explicit list** maintains a list that only points to free blocks
 - What's the trade-off?
 - This could make allocation faster--now we iterate through our 'explicit list' that points to blocks of memory that are free of a certain size.



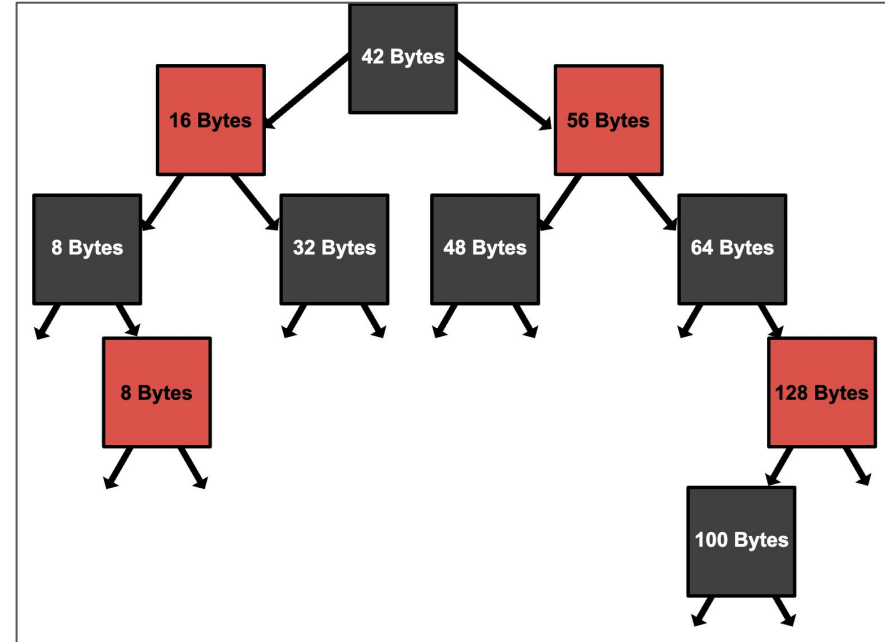
Keeping track of memory | Strategy 4 - **Pool**

- Maintain a separate free list for different size classes
 - (i.e. have either multiple linked lists, either explicit or implicit, keeping track of memory)
 - Perhaps size your pools to reasonable values -- all powers of 2
 - **Or even better** -- sizes of objects in your game/application



Keeping track of memory | Strategy 5 - Tree

- Maintain a structure that is sorted by size
 - Some heap, or balanced tree structure
 - (Red-black tree shown and link to the bottom-right)



<https://www.gingerbill.org/article/2021/11/30/memory-allocation-strategies-005/#red-black-tree-approach>

Okay, we have a general idea of our storage mechanism (i.e. data structure) to bookkeep memory and also find free memory.

What further strategies do we have to allocate/free memory during run-time?

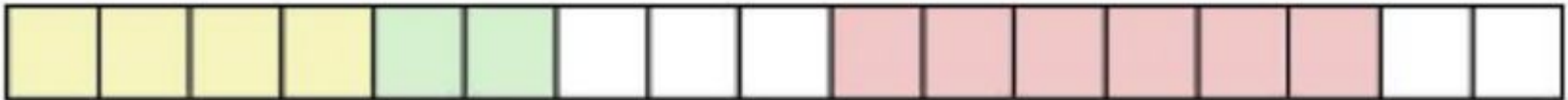
Revisit Naive Strategy - monotonic allocator

- Allocate every time we need memory (monotonic allocator)
 - (Also see term [bump allocator](#))
- **I instead** want to show you how to reuse blocks that have been previously freed after malloc'ing
 - (See next slide!)

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(){
5
6     // Print out the top of the heap for a process.
7     // This is the first byte we can 'grab' to store something in.
8     // We pass in '0' as an argument into sbrk.
9     void* top = sbrk(0);
10    printf("top of heap: %p\n",top);
11
12    // Extend heap 4 bytes
13    char* bytes = sbrk(4);
14    printf("new top of heap: %p\n",sbrk(0));
15
16    // Set our bytes to some data
17    bytes[0] = 'h';
18    bytes[1] = 'i';
19    bytes[2] = '\n';
20    bytes[3] = '\0';
21    printf("Stored in heap %s",bytes);
22
23
24    return 0;
25 }
```

Implicit List: How to find/choose a free block (1/4)

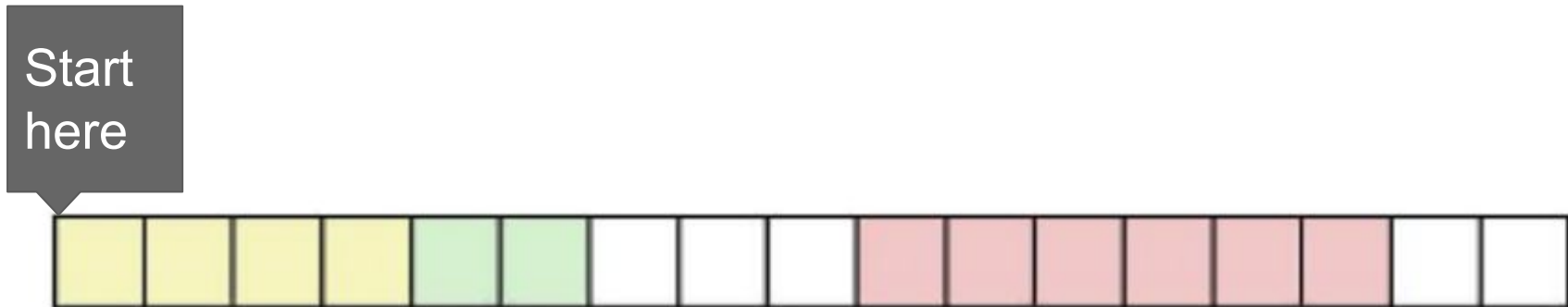
- **First fit strategy** (For a first allocator, I recommend this for learning)
 - Search from beginning of list
 - Choose first free block that fits
 - Takes linear time: $O(\text{number of allocated and freed blocks})$



Implicit List: How to find/choose a free block (2/4)

- **First fit strategy**

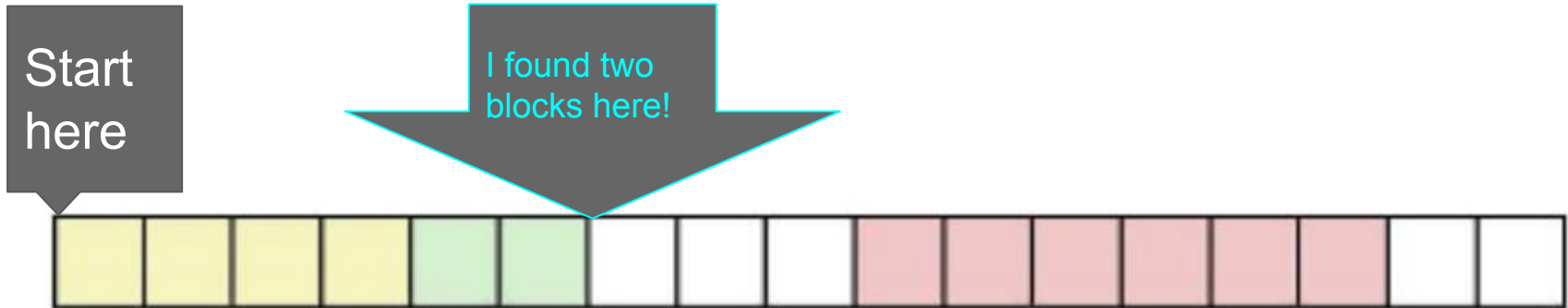
- Search from beginning of list
 - Choose first free block that fits
- Takes linear time: $O(\text{number of allocated and freed blocks})$



Implicit List: How to find/choose a free block (3/4)

- **First fit strategy**

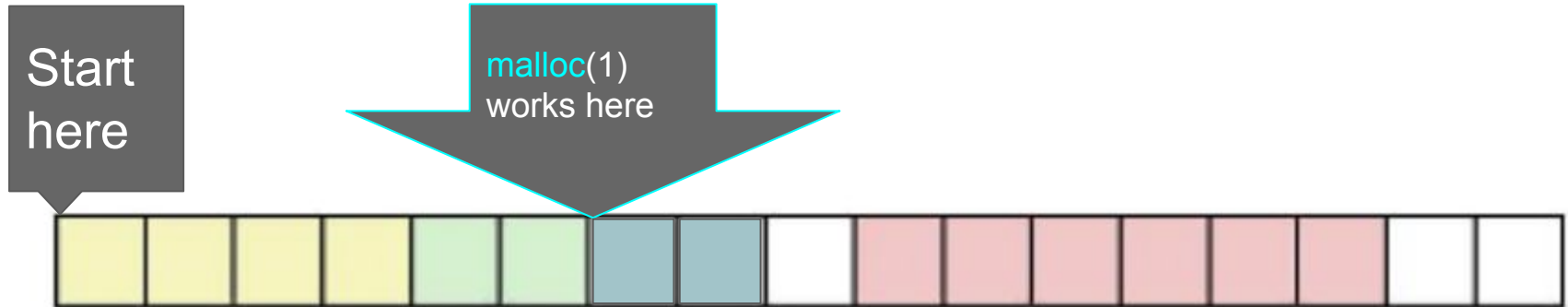
- Search from beginning of list
 - Choose first free block that fits
- Takes linear time: $O(\text{number of allocated and freed blocks})$



Implicit List: How to find/choose a free block (4/4)

- **First fit strategy**

- Search from beginning of list
 - Choose first free block that fits
- Takes linear time: $O(\text{number of allocated and freed blocks})$



Implicit List: How to find/choose a free block (1/4)

- **Next-fit strategy**

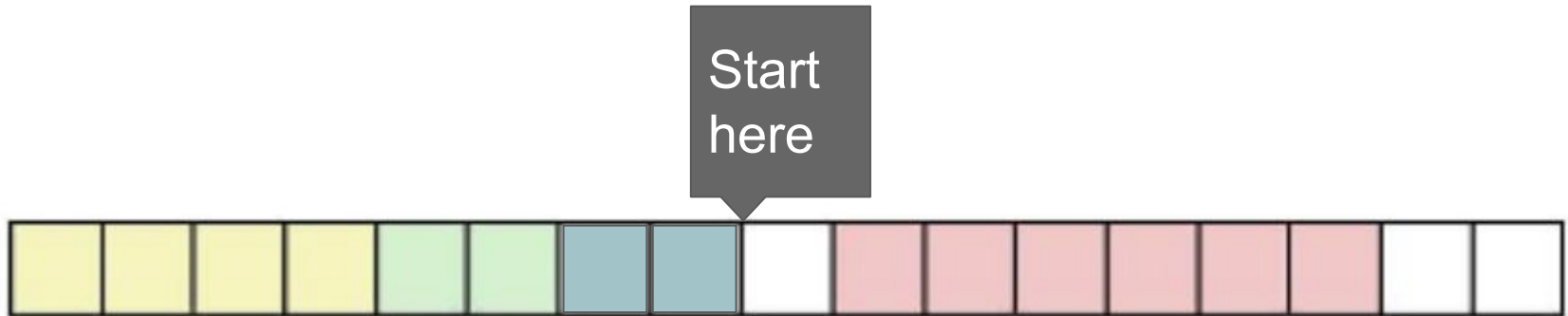
- Search from where you left off from your previous search
 - Choose first free block that fits



Implicit List: How to find/choose a free block (2/4)

- **Next-fit strategy**

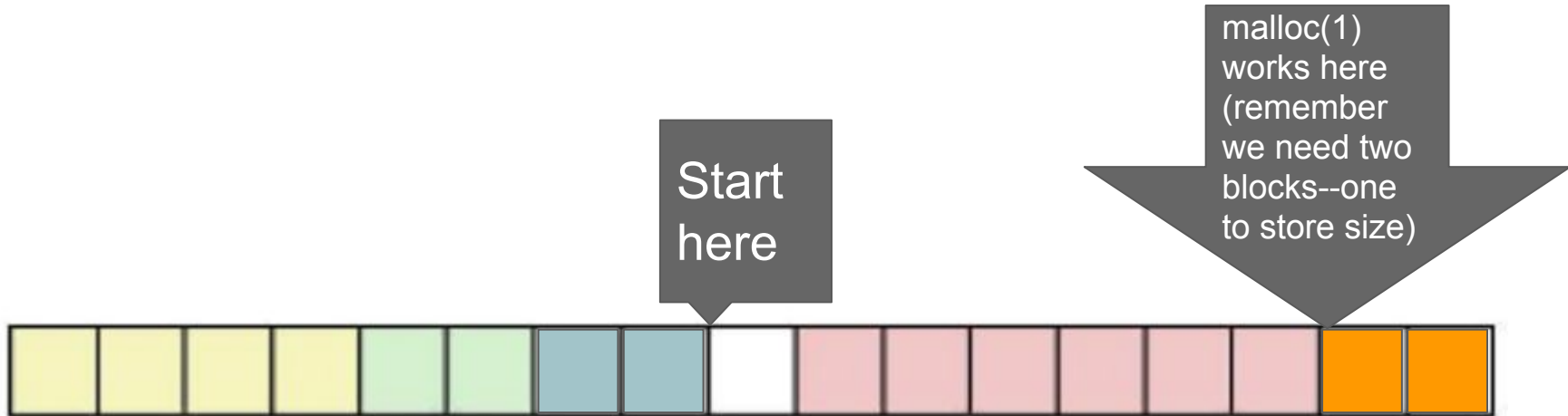
- Search from where you left off from your previous search
 - Choose first free block that fits



Implicit List: How to find/choose a free block (3/4)

- **Next-fit strategy**

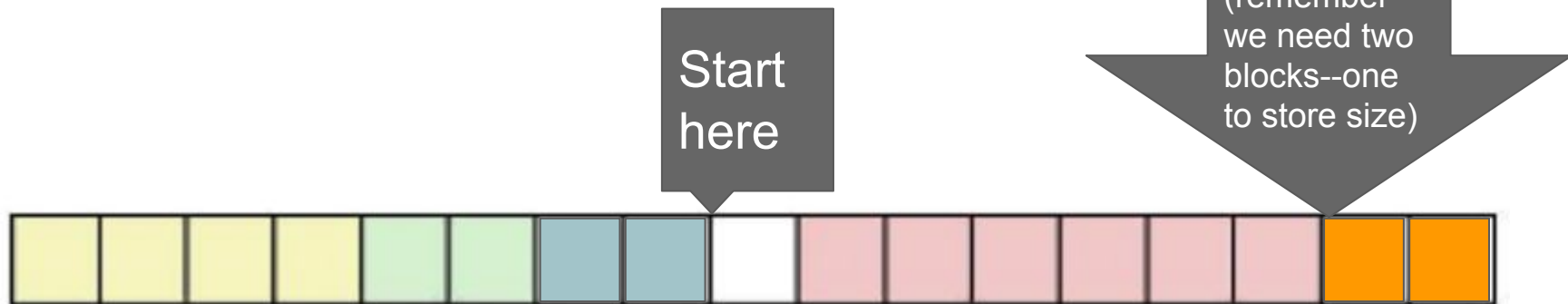
- Search from where you left off from your previous search
 - Choose first free block that fits



Implicit List: How to find/choose a free block (4/4)

- **Next-fit strategy**

- Search from where you left off from your previous search
 - Choose first free block that fits
- Takes linear time: $O(\text{number of allocated and freed blocks})$
 - May be better, avoids re-scanning unhelpful blocks if you are doing many similar allocations
 - Could make fragmentation worse though!

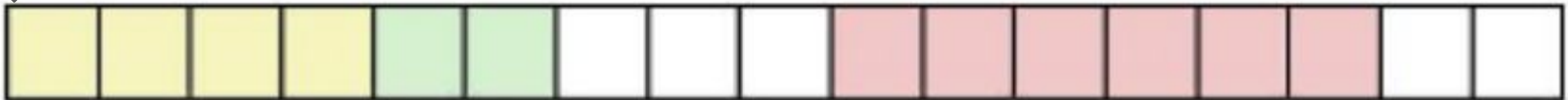


Implicit List: How to find/choose a free block (1/4)

- **Best-fit strategy**

- Scan for the block that fits best
 - i.e. fewest bytes left over
- Keeps fragmentation small and improves memory utilization
- Will typically run slower than first-fit (longer scan for optimal block)

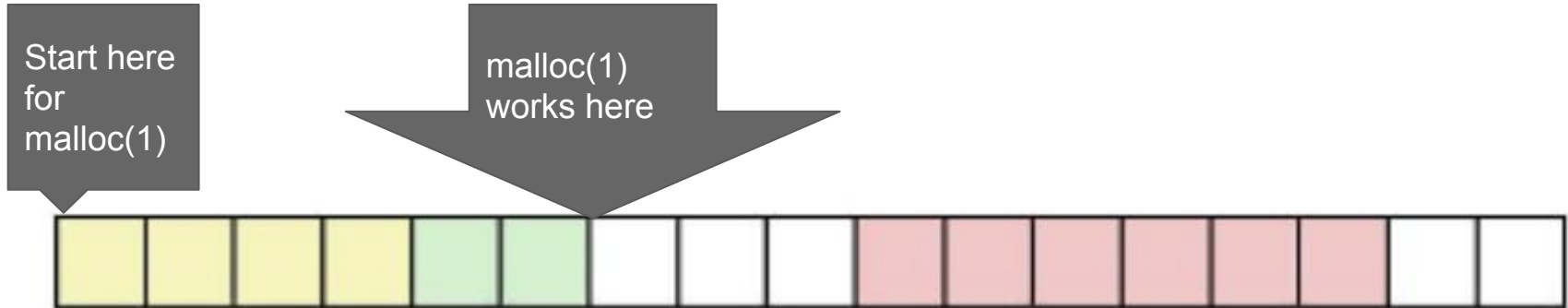
Start here
for
malloc(1)



Implicit List: How to find/choose a free block (2/4)

- **Best-fit strategy**

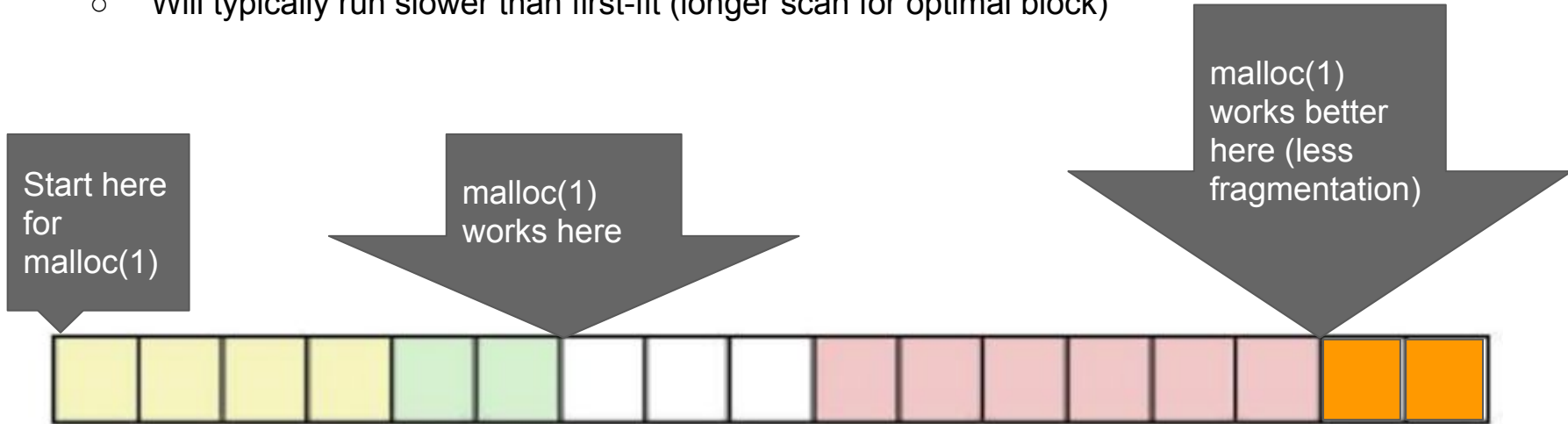
- Scan for the block that fits best
 - i.e. fewest bytes left over
- Keeps fragmentation small and improves memory utilization
- Will typically run slower than first-fit (longer scan for optimal block)



Implicit List: How to find/choose a free block (3/4)

- **Best-fit strategy**

- Scan for the block that fits best
 - i.e. fewest bytes left over
- Keeps fragmentation small and improves memory utilization
- Will typically run slower than first-fit (longer scan for optimal block)

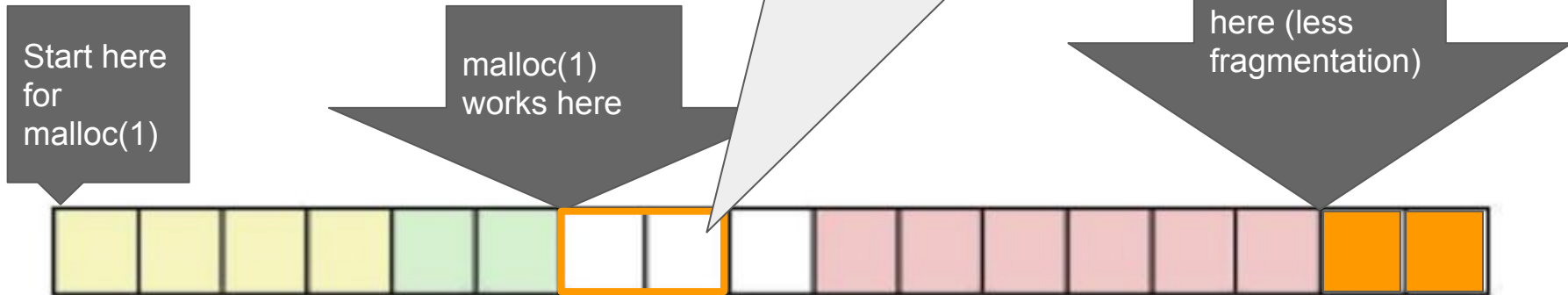


Implicit List: How to find/choose

- Best-fit strategy
 - Scan for the block that fits best
 - i.e. fewest bytes left over
 - Keeps fragmentation small and in check
 - Will typically run slower than first-fit

Observe that if allocated here instead of at the end, we would have '1' wasted block (only perhaps available for the header)

This is known as **fragmentation**, and it's a challenge with memory allocators

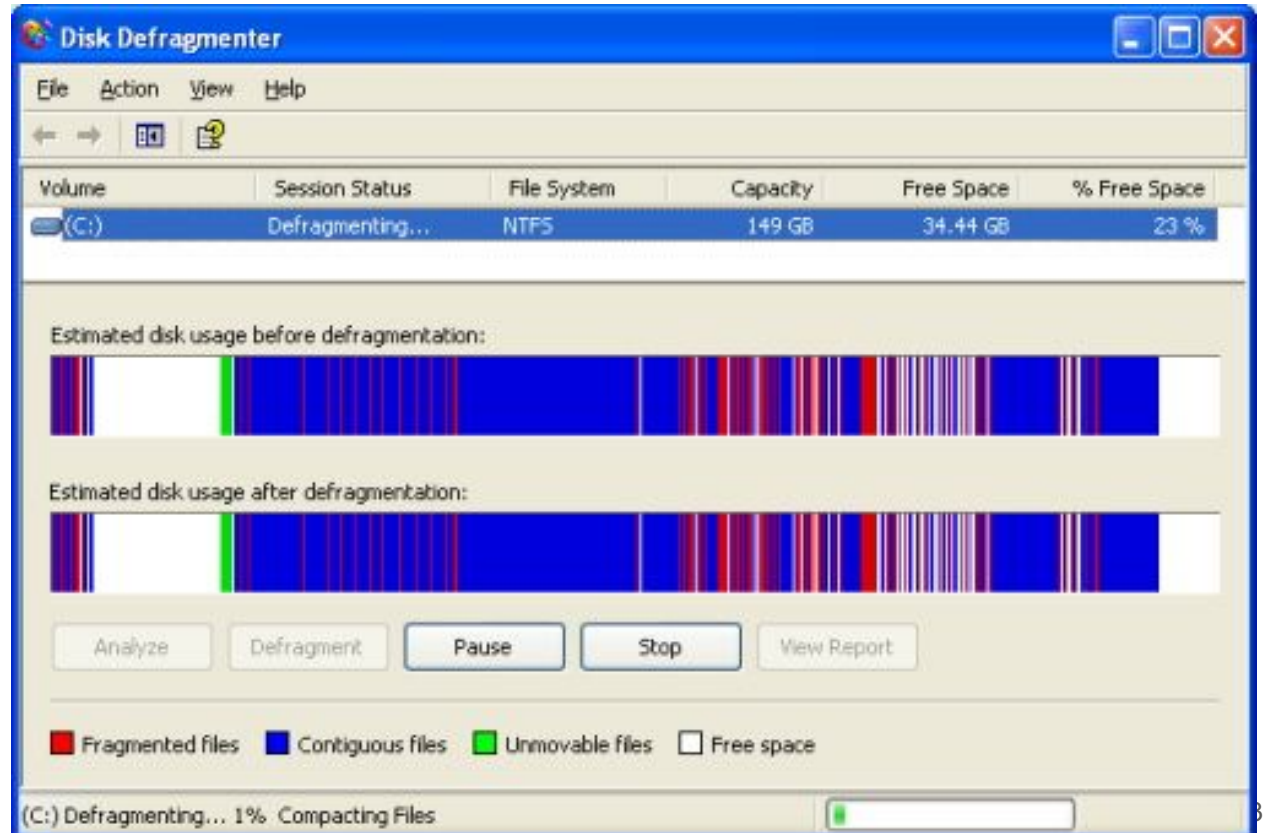


What is Fragmentation?

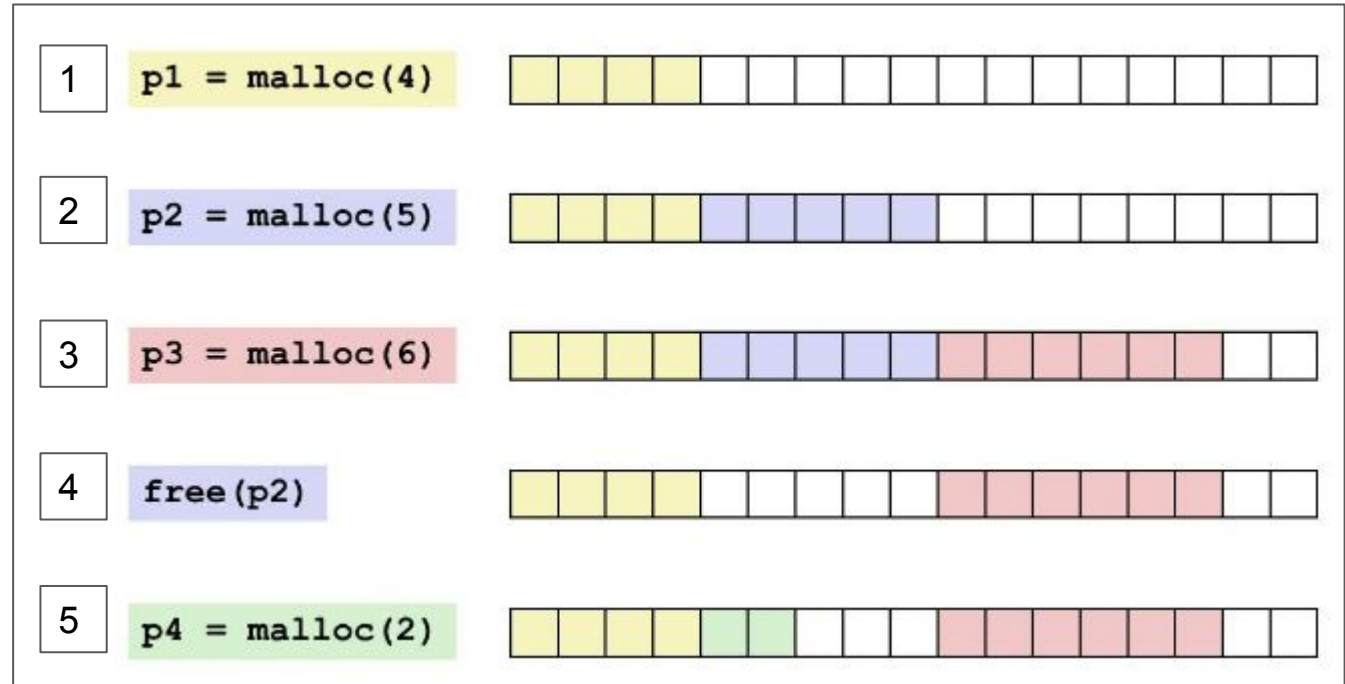
A challenge with memory allocation

Question to Audience: Who has run a disk defragmentor?

- It's good to do relatively often the longer you own your machine and the more files you store.



Fragmentation example visualization (1/2)

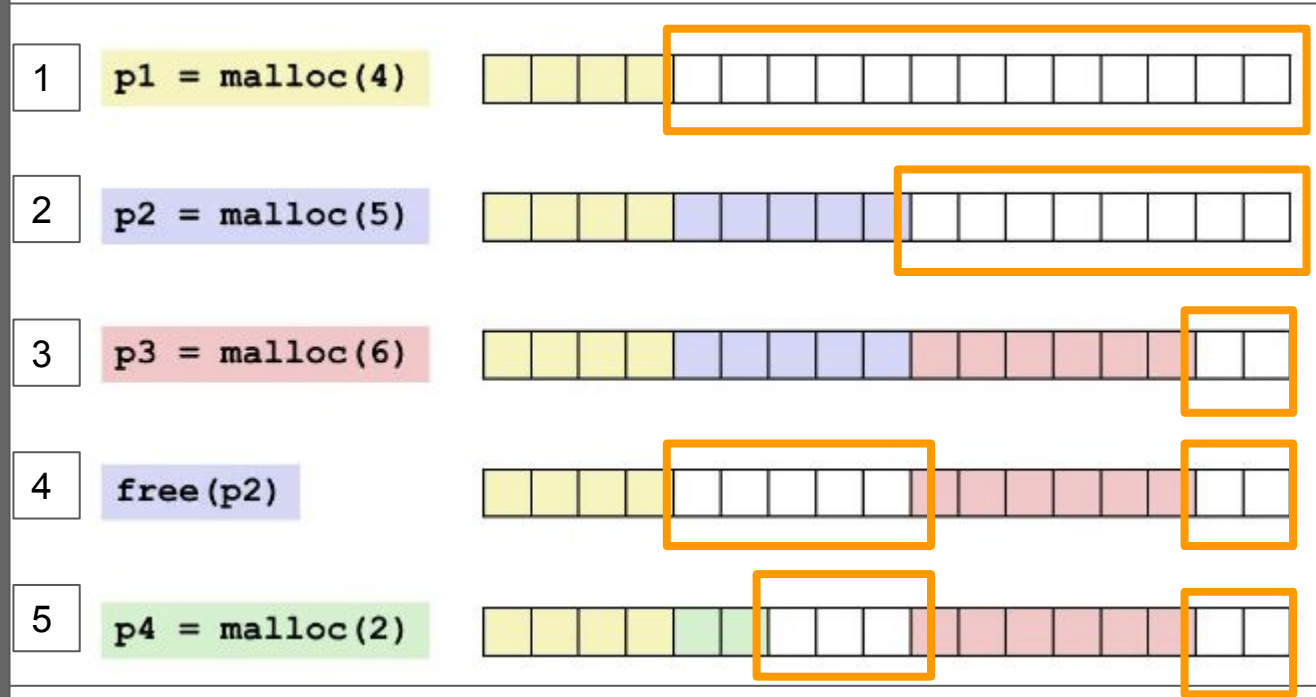


Ideally we have no empty blocks ever*

We only want to use what we need.

These 'gaps' are sometimes a result of what is called fragmentation.

example visualization (2/2)



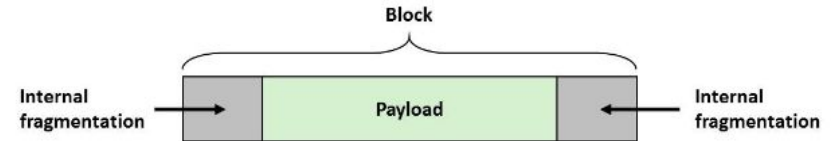
*observer we malloc 17 total bytes at most, so that's our upper bound. At any given time the maximum we have is '15' bytes, but we always have some extra empty blocks.

Excessive Fragmentation = Poor memory utilization (1/2)

- Two types of fragmentation

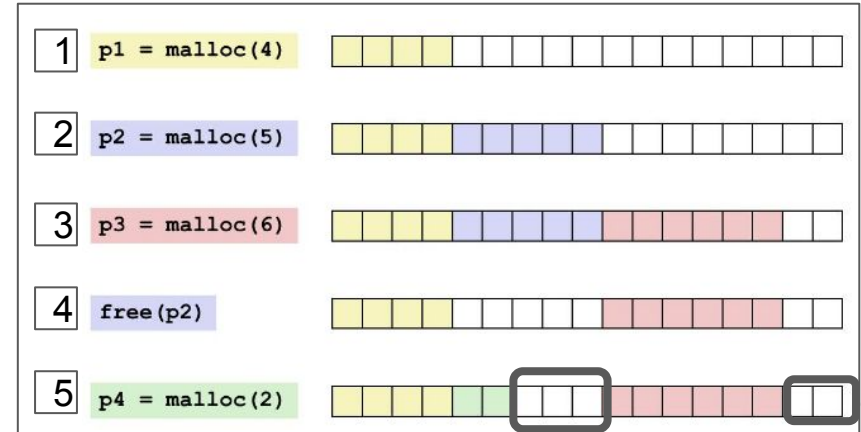
- a. Internal Fragmentation

- e.g. we allocate structures smaller than our block size of X bytes (where X is architecture dependent)
 - (e.g. We allocate 1 character which is one byte, but our blocks are given out 8 bytes at a time)



- b. External Fragmentation

- We have enough blocks (i.e. we don't need to extend the heap), but the allocations are not all contiguous
 - -- see on the right that we cannot malloc(4) for instance.



Excessive Fragmentation = Poor memory utilization (2/2)

- Two types of fragmentation

- a. Internal Fragmentation

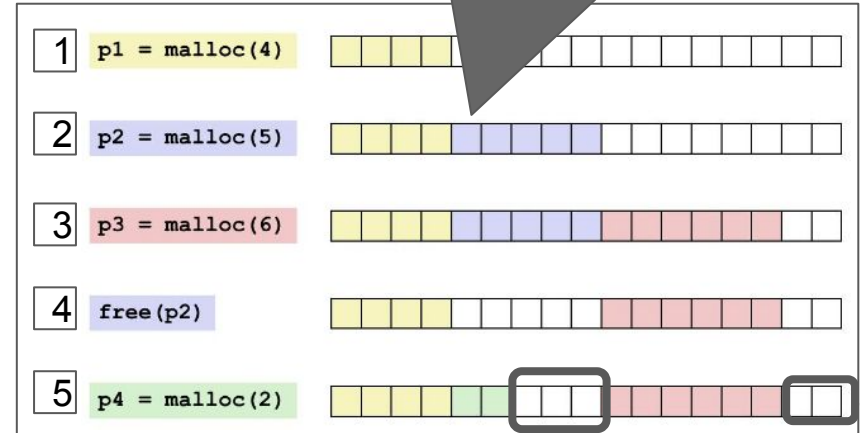
- e.g. we allocate structures smaller than our block size of X bytes (where X is architecture dependent)
 - (e.g. We allocate 1 character which is one byte, but our blocks are given out 8 bytes at a time)

- b. External Fragmentation

- We have enough blocks (i.e. we don't need to extend the heap), but the allocations are not all contiguous

Note: It is **hard** up to this point to know fragmentation will occur based on the order of mallocs and frees.

We cannot always predict the future.

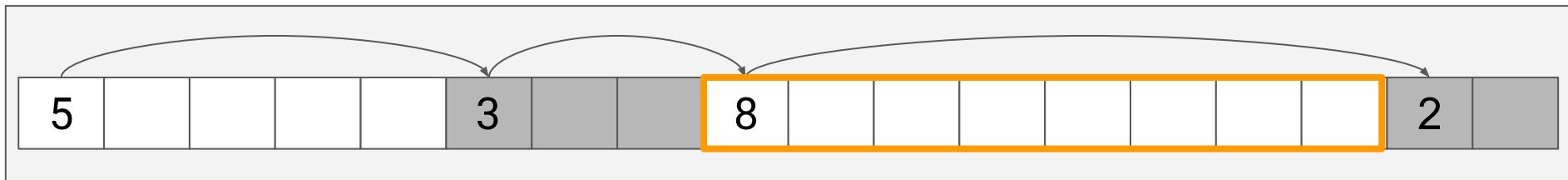


Splitting and Coalescing (i.e. Combining Blocks)

To help avoid fragmentation/use finite resources more efficiently

Implicit List: How to allocate a free block? (**Possible to split**) (1/2)

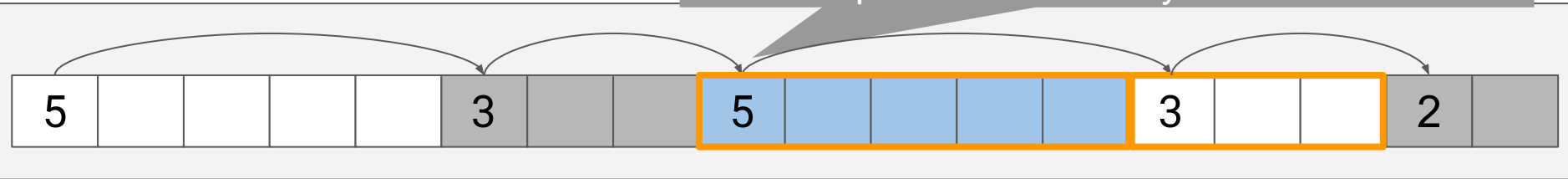
- Depending on our strategy
 - We could allocate to the block found
 - i.e. we set a pointer to that block and size
 - We **may want to also split that block** if there is room to do so and leave some reasonable free space for another allocation (See example below)



Implicit List: How to allocate a free block? (**Possible to split**) (2/2)

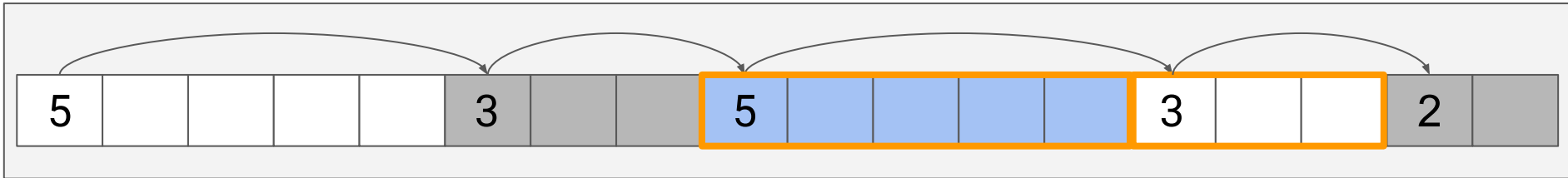
- Depending on our strategy
 - We could allocate to the block found
 - i.e. we set a pointer to that block and size
 - We **may want to also split that block** if there is room to do so and leave some reasonable free space for another allocation (See example below)

allocate: 'p' here that is 4 bytes + header



Implicit List: How to free a block and combine? (1/3)

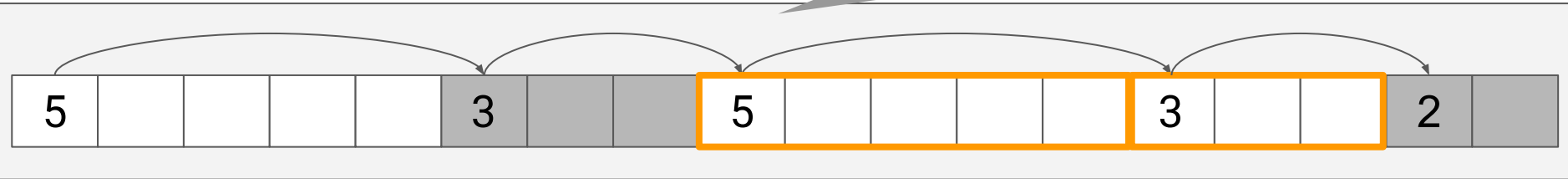
- Simply free a block by setting the free bit in the header
 - This could lead to fragmentation however!



Implicit List: How to free a block and combine? (2/3)

- Simply free a block by setting the free bit in the header
 - This could lead to fragmentation however!

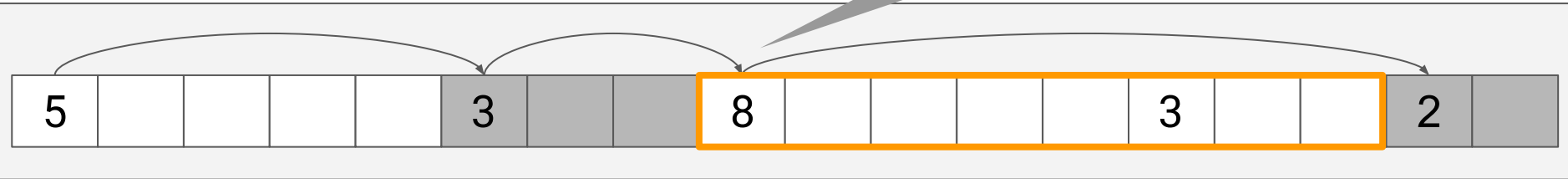
Let's free p now, and now below 'p' is logically free.



Implicit List: How to free a block and combine? (3/3)

- Simply free a block by setting the free bit in the header
 - This could lead to fragmentation however!

Could combine (coalesce) adjacent free blocks into a bigger block of '8'

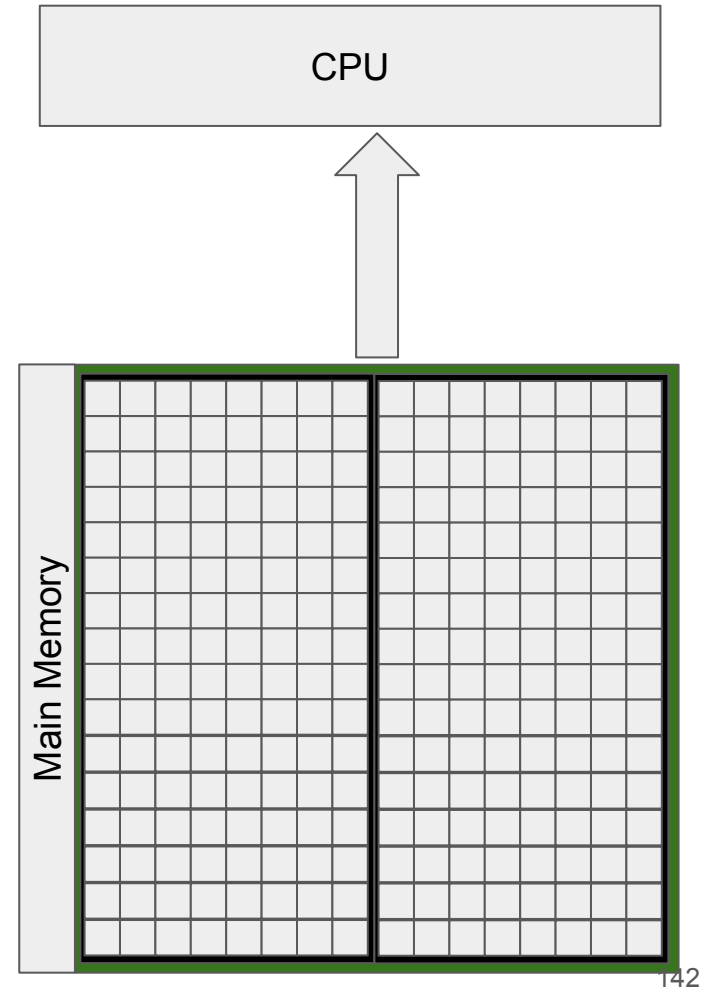


Global and Local Allocators

(Approaching the 'finale' portion of the talk)

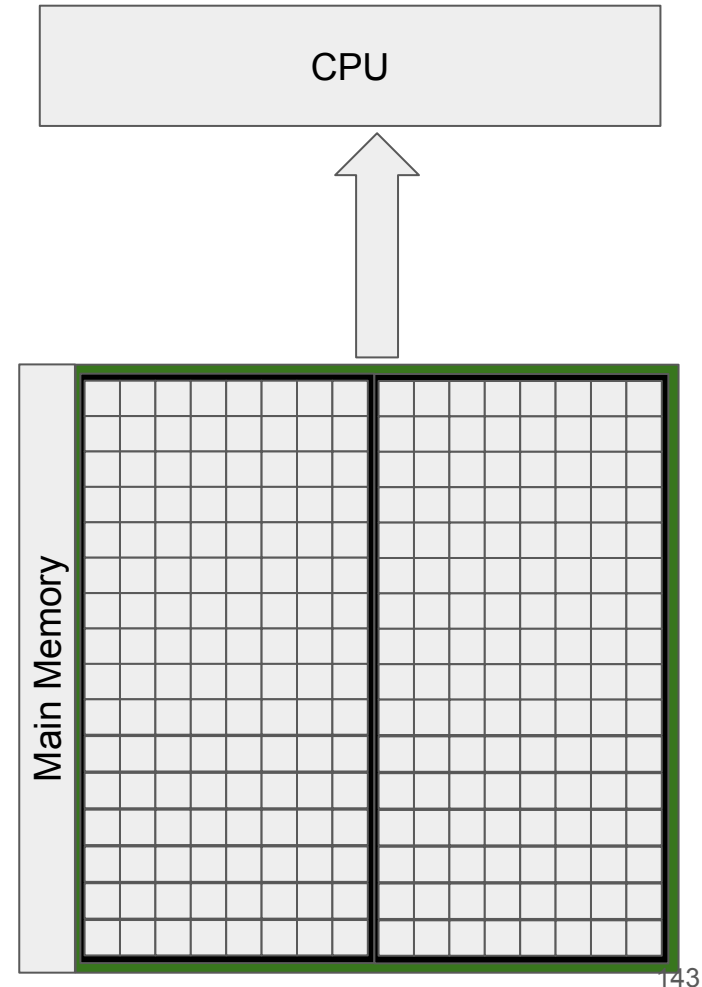
Global versus Local allocators (1/10)

- So we've got a way to grab memory
- We have a way to track memory
- And we have a way to navigate memory allocations to fit in new memory
 - First-fit, next-fit, best-fit
- We've been assuming we get all of the memory as shown on the right in our allocator
- This is how 'malloc' implementations operate



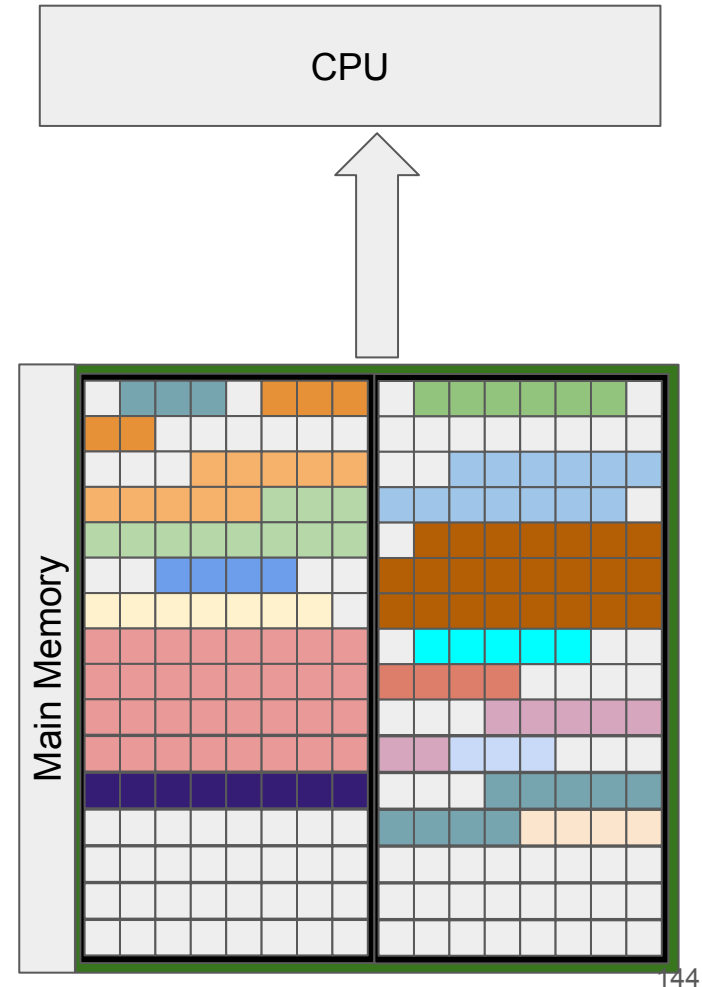
Global versus Local allocators (2/10)

- There are some issues that arise with a global allocator
 - We have a lot of memory to manage -- so we might get fragmentation
 - (and diffusion -- which is lots of allocations left all over memory)



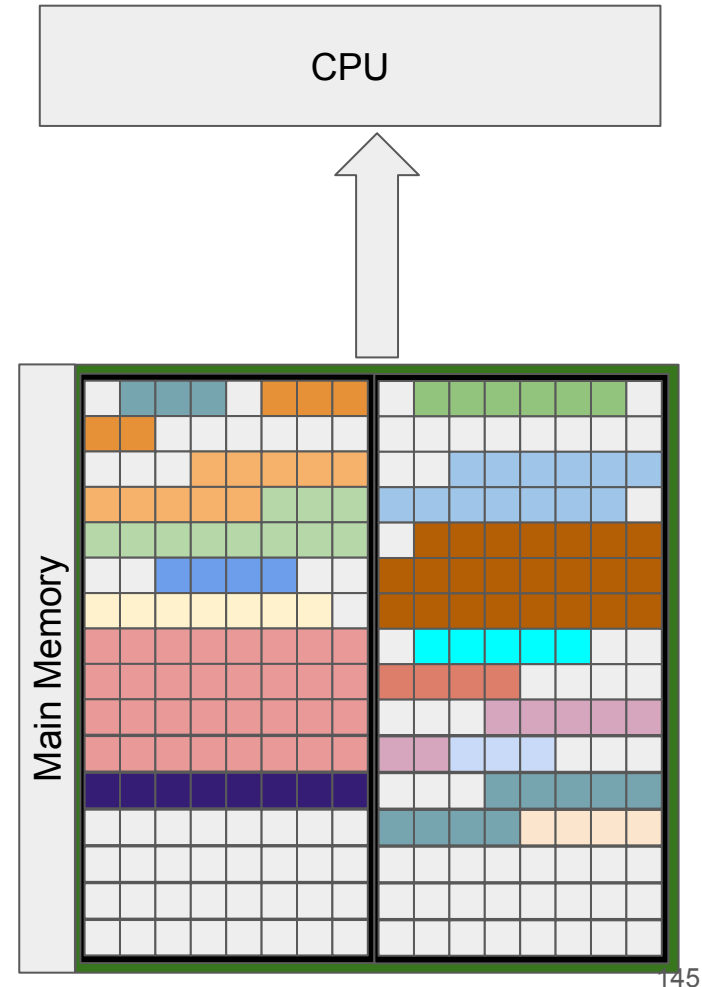
Global versus Local allocators (3/10)

- There are some issues that arise with a global allocator
 - We have a lot of memory to manage -- so we might get fragmentation
 - (and diffusion -- which is lots of allocations left all over memory)



Global versus Local allocators (4/10)

- There are some issues that arise with a global allocator
 - We have a lot of memory to manage -- so we might get fragmentation
 - We might also have issues of 'contention' with multithreaded programs
 - So let's imagine two threads are calling on 'malloc' to allocate
 - Essentially we need a global lock
 - (another reason heap memory allocation slows down)

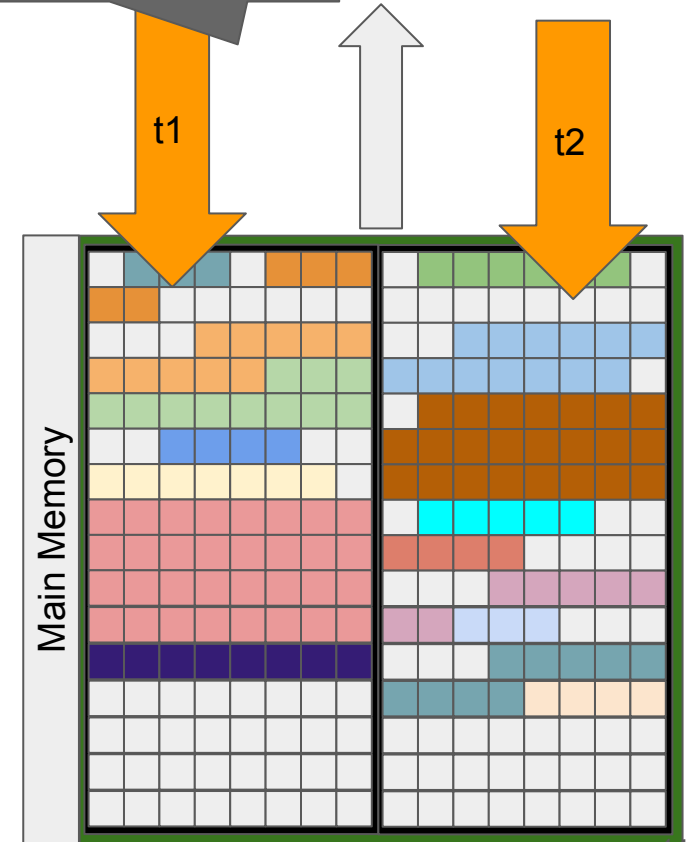


Global versus Local allocators

Threads t1 and t2 searching for open memory need to obtain a lock first

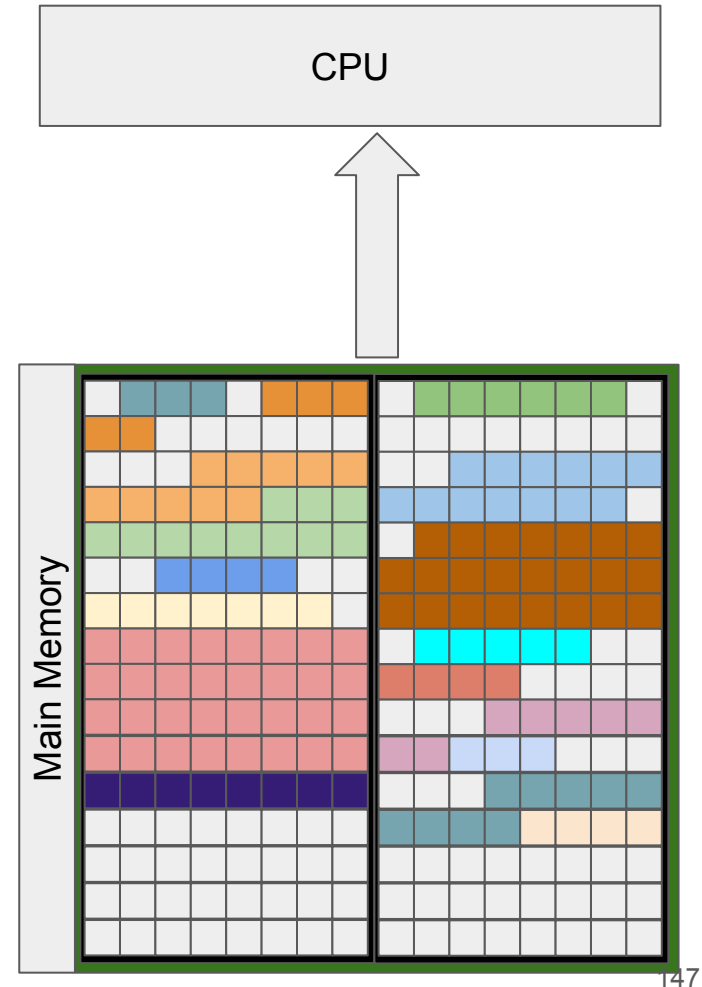
PU

- There are some issues that arise with a global allocator
 - We have a lot of memory to manage -- so we might get fragmentation
 - We might also have issues of 'contention' with multithreaded programs
 - So let's imagine two threads are calling on 'malloc' to allocate
 - Essentially we need a global lock
 - (another reason heap memory allocation slows down)



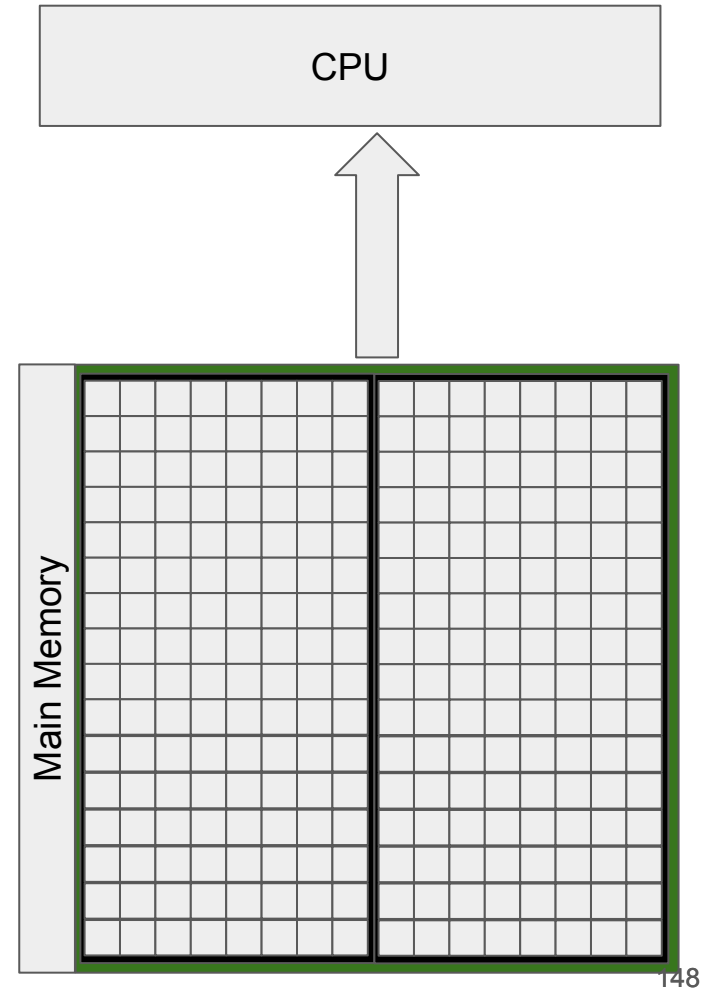
Global versus Local allocators (6/10)

- There are some issues that arise with a global allocator
 - We have a lot of memory to manage -- so we might get fragmentation
 - We might also have issues of 'contention' with multithreaded programs
 - So let's imagine two threads are calling on 'malloc' to allocate
 - Essentially we need a global lock
 - (another reason heap memory allocation slows down)
 - And we haven't touched on locality yet -- but because we've guided our program to look everywhere for memory, this has implications on cache performance.



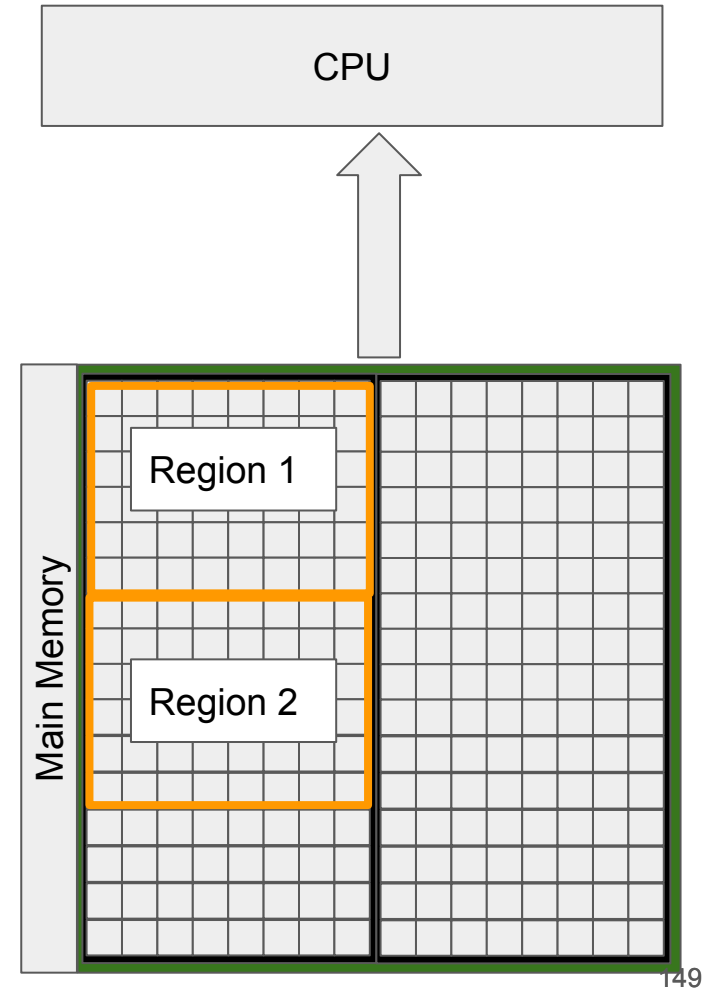
Global versus Local allocators (7/10)

- One strategy is to use 'local allocators' that are given a few pages (or some other fixed allocation size that you choose)



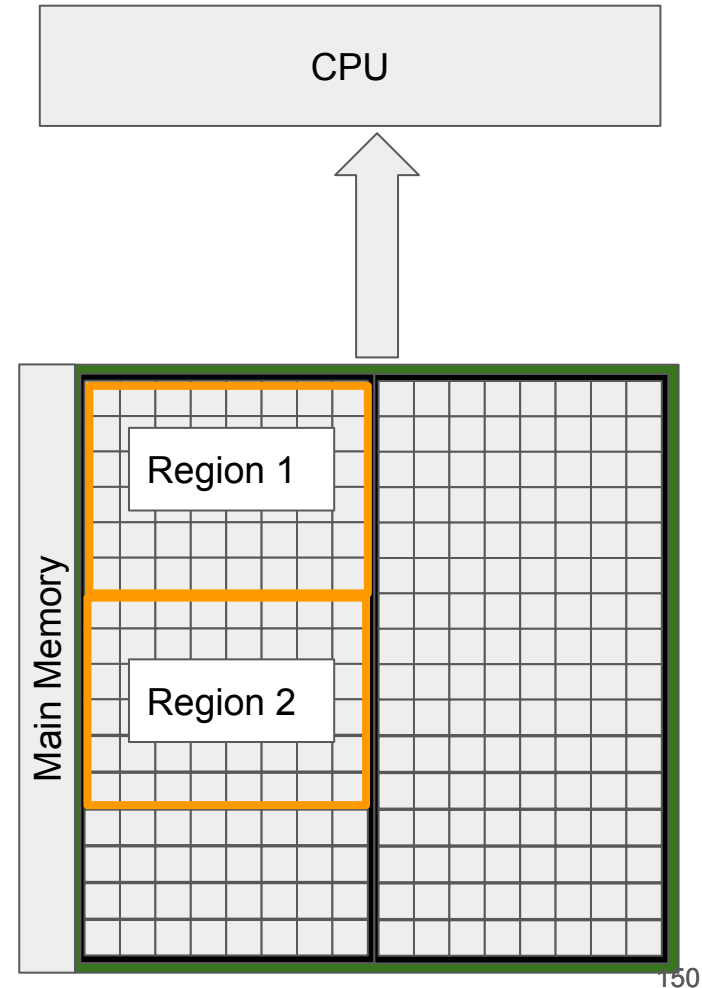
Global versus Local allocators (8/10)

- One strategy is to use 'local allocators' that are given a few pages (or some other fixed allocation size that you choose)



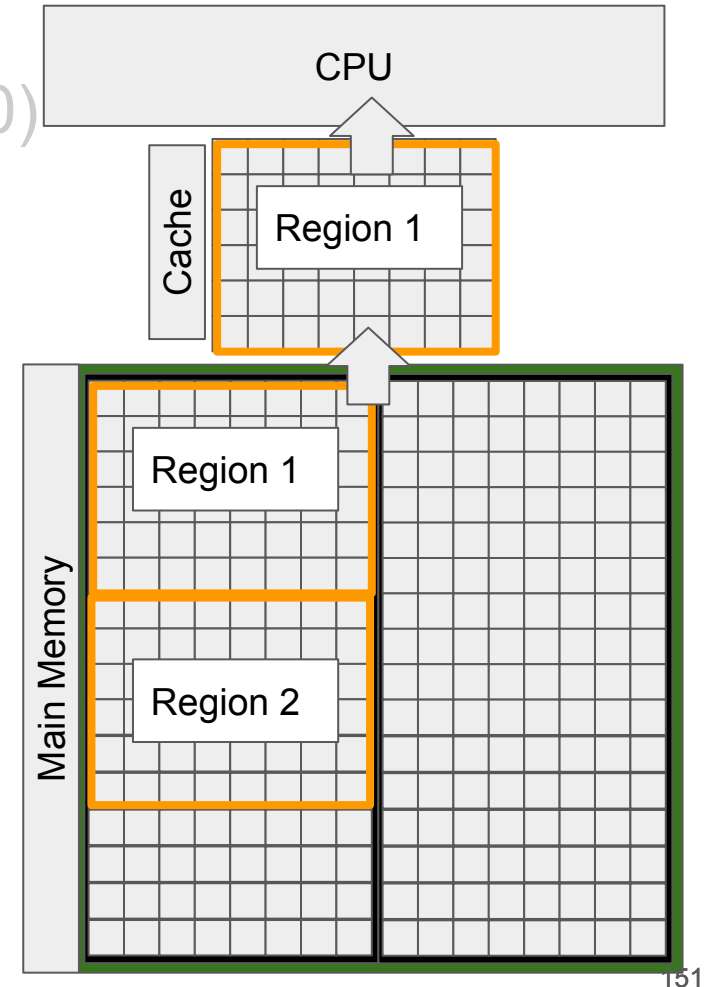
Global versus Local allocators (9/10)

- One strategy is to use 'local allocators' that are given a few pages (or some other fixed allocation size that you choose)
 - Now within each of these regions, you might have different allocation strategies themselves!
 - e.g.
 - Region 1 is a pool allocator with certain fixed sizes
 - Region 2 acts as a general purpose allocator
 - And if Region 2 runs out of memory, it could fall back to 'malloc' for instance



Global versus Local allocators (10/10)

- Kind of neat if we size 'Region 1' to match your L1/L2/L3/L4 cache size for some subsystem that you're utilizing.
- Kind of neat if 'Region 1' and 'Region 2' can be used safely from separate threads
 - i.e. we can avoid some contention
- Kind of neat if you can fit one subsystem (graphics, AI, physics) in a specific region



Here is some
'example' of how you
might structure an
arena allocator

(This is purely
slideware, but
probably a reasonable
interface to build off
of)

```
1 // @file arena.hpp
2 class LocalArena{
3     private:
4         // Choose a data structure to manage your data
5         // - ImplicitList* myImplicitList;
6         // - ExplicitList* myExplicitList;
7         // - ListOfLists* myPoolAllocator
8
9         // *Maybe* one std::mutex if your Local Arena is shared.
10
11     public:
12         /* Create a region or 'arena' of memory */
13         LocalArena(void* startOfRegion, void* endOfRegion){ /* .. */ }
14
15         /* Find space in your arena for the # of bytes requested
16         */
17         void* Allocate(std::size_t bytes);
18
19         // Very simple allocation functions as an example for pool allocator
20         // void* Allocate8Bytes();
21         // void* Allocate16Bytes();
22         // void* Allocate32Bytes();
23
24         /* Pass in an address, and mark free in Local Arena
25         Note: Optionally 'zero' out memory or do other bookkeeping.
26         */
27         void DeAllocate(void* address){ }
28
29         /* reclaim all memory in just this arena */
30         void Release(){ }
31 };
```


Some more Ideas and Best Practices

mmap

- Our strategy of using 'sbrk' to allocate works
- But making calls to 'sbrk' constantly is performing many systems calls
- Instead, we typically allocate in 'page size' increments and can use mmap to ask for larger allocations.
 - sbrk again might be used for small heap allocations in a global allocator

```
MMAP(2)                                Linux Programmer's Manual                                MMAP(2)

NAME
    mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
    #include <sys/mman.h>

    void *mmap(void *addr, size_t length, int prot, int flags,
                int fd, off_t offset);
    int munmap(void *addr, size_t length);

    See NOTES for information on feature test macro requirements.

DESCRIPTION
    mmap() creates a new mapping in the virtual address space of the
    calling process. The starting address for the new mapping is spec-
    ified in addr. The length argument specifies the length of the
    mapping (which must be greater than 0).

    If addr is NULL, then the kernel chooses the address at which to
    create the mapping; this is the most portable method of creating a
    new mapping. If addr is not NULL, then the kernel takes it as a
    hint about where to place the mapping; on Linux, the mapping will
    be created at a nearby page boundary. The address of the new map-
    ping is returned as the result of the call.
```

Potentially Ideal 'Heap' Memory Management

- Only one large heap allocation (at the start of the program) in order avoid system calls context switches.
 - i.e. allocate one big chunk (e.g. `malloc(100000000);` or `new data[10000000];`)
 - Note: This is essentially what we do per process for stack memory -- just allocate once.
- Once we have our large block of memory, you can then divide it up as needed and perform bookkeeping
 - Again, take advantage of your custom memory allocator to handle when a programmer asks for memory.

A few more ideas

- Consider trying to avoid dynamic memory allocations altogether!
 - e.g. static strings allocated in static portion of memory
 - e.g. create static containers
 - e.g. small string allocations (small string allocations get stored on stack
 - i.e. think about where you can put memory

Conclusion and Further Resources

Wrapping up what we've learned

Summary on Memory Allocation

- We should have a good understanding of stack and heap
 - And the tradeoffs when allocating in each segment of memory in our process
- We've discussed motivation on why we might want to move away from using a general purpose heap allocator (e.g. malloc)
 - We've looked at some different memory allocator strategies
 - We've looked at some allocator designs
 - alloca
 - monotonic
 - various dynamic memory allocators using 'sbrk'
 - LocalArena allocator

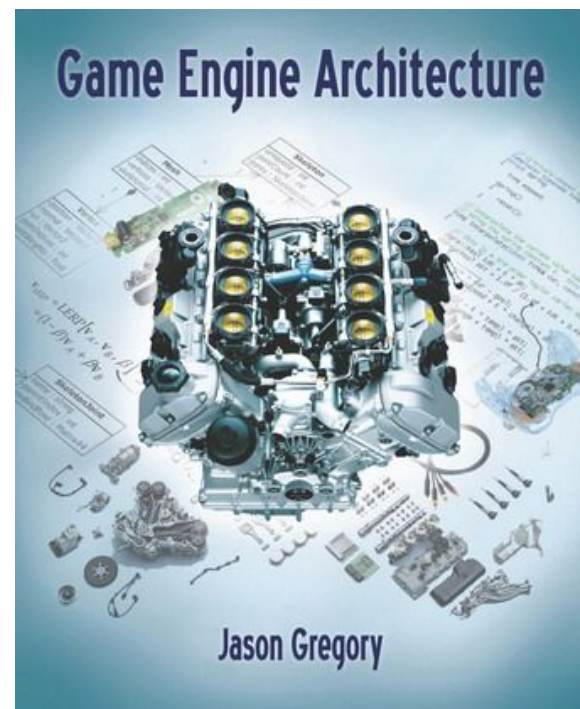
The Next 3 Resources to Learn More (1/3)

(Listed in the order I think they should be read/watched)

1. Computer Systems: A Programmer's Perspective (Wonderful book)
 - a. Book used in Carnegie [Mellon Course on Systems](#)
2. Jason Gregory
 - a. Game Engine Architecture
 - b. <https://www.gameenginebook.com/> (The book is wonderful!)
3. John Lakos Arena allocators talk
 - a. <https://www.youtube.com/watch?v=xUtndRUJHX8&t=1s>

Resources (2/3)

- Jason Gregory's book "Game Engine Architecture" is where I learned a lot of this stuff a few years ago
 - <https://www.gameenginebook.com/>
- (And Jason has presented at an iteration of handmade -- how I originally found out about Handmade Seattle!)



Resources (3/3)

- Ryan Fleury blog post
 - <https://www.rfleury.com/p/untangling-lifetimes-the-arena-allocator> (Thank you for sending this Abner)
- [Emery Bergrers paper on custom memory allocators vs malloc \(2002\)](#)
 - “On reconsidering memory management)
- Emery Berger paper on Hoard
https://en.wikipedia.org/wiki/Hoard_memory_allocator
- More overview on different types of memory allocators
 - <https://www.openmp.org/spec-html/5.0/openmps53.html>
- See other allocation strategies
 - Slab allocator (GNU libc)
 - Buddy system (linux kernel)
- Googles allocator tcmalloc [[github](#)]
- jemalloc [<https://jemalloc.net/>] [[github](#)] [[Video](#)]
- [C++Con 2017: Pablo Halpern “Allocators: The Good Parts”](#)

Small Aside

- Of the two books in my 'Building Game Engines' -- they are from two speakers who've presented in the Handmade community
 - Jason Gregory -- Game Engine Architecture
 - Robert Nystrom -- Game Programming Patterns
- Go buy the books if you are able, they're worth every dollar to support the authors.

Resources

There will be no required textbook for this course. However, these resources are recommended.

- (Strongly Recommended) [Game Engine Architecture](#)
- (Free) [Game Programming Patterns](#)
- (Free) [C++ Tutorial](#)



Thank you!

November 16-18th, 2022

**Independent
Systems Programming
Conference**

Introduction to Memory Allocation: Design and Implementation

Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube: [www.youtube.com/c/MikeShah](#)

17:00-18:00, Wed, 16th November 2022

<https://handmade-seattle.com/>

60 minutes | Introductory/Intermediate Audience

Extras and Notes