Lab 06 - Memory Allocation

Instructor: Lorenzo De Carli, University of Calgary (*lorenzo.decarli@ucalgary.ca*)

Slides by Lorenzo De Carli, based on material by Robert Walls (WPI)

Introduction

- Have you ever wondered what happens when you call **`malloc()`**? In this lab you will figure it out (sorta)
- You will have **two weeks** to complete this lab. As such, you may find it amore challenging than the previous one
- You are encouraged to **read chapter 17 of the book**. We won’t cover this material in class, but it will provide a lot of **useful context** for the lab

What is the goal?

…to implement a memory allocator!

- You must implement a memory allocation library in two files (**`myalloc.h`** and **`myalloc.c`**)
- The library implements four functions:
  - **`void myinit(size_t size);`** // initialize the memory allocator
  - **`void* myalloc(size_t size);`** // request allocation of new memory
  - **`void myfree(void* ptr);`** // free memory allocated with 461alloc
  - **`void mydestroy();`** // release all resources used by the allocator

Warning: you are entering a `malloc()`-free zone

At no point in this assignment you are allowed to use `malloc()`

Function prototypes

- The four functions are prototyped in the header file provided in the code template. The header is called **`myalloc.h`**
- **Do not modify this file**, otherwise you will get **0 points** for the lab
- Your code should go into **`myalloc.c`**
- To avoid **compiler errors**, please make sure your implementation of **`myalloc.c`** includes the following:

```
#include <stddef.h>

#include "myalloc.h" // This include should appear last
```

Test cases

- The assignment is divided in **5 conceptual parts**
- We will provide **test cases for each part** in the code template
- You can run part-specific test cases by going into the appropriate folder and run **`test-partX.sh`**
- There is a **`test_all.sh`** script you can use to run the test cases for all parts
- For obvious reasons, we suggest that you make sure that your code **passes all test cases**

Part 1

Initialization and Destruction

- **`myinit()`** must get a large memory area (called an **arena** in the following) from the OS, which will be used to allocate the chunks of memory requested by **`myalloc()`**
- How do I request such a contiguous area? The easiest is to use **`mmap()`**. Your code should look like the lines below:

```
_arena_start = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
```

Example `myinit()` output

```
Initializing arena:
...requested size 1 bytes
...pagesize is 4096 bytes
...adjusting size with page boundaries
...adjusted size is 4096 bytes
...mapping arena with mmap()
...arena starts at 0x7f8e5f4f9000
...arena ends at 0x7f8e5f4fa000
```

What is this `mmap()` business?

…just the most confusing syscall ever

- If you are confused about **`mmap()`**, then you are not alone
- The canonical use of **`mmap()`** (and probably its original purpose) is to **map a file to a memory region**
  - Reads/writes to **memory** are automatically transferred to the underlying **file**
- However, it also allows to **allocate a memory region without mapping it to any file** (that's how you are using it here)
- There are reasons to use it which we won't get into - but it can be **more efficient** than **`malloc()`** for large allocations

What about destruction?

- **`mydestroy()`** should de-allocate resources allocated by **`myinit()`**
- Primarily, it should use **`munmap()`** to de-allocated the memory requested by **`mmap()`**
- **`mydestroy()`** should also reset any state variable that your code keep, restoring them to their original value
- "**`man mmap`**" and "**`man munmap`**" are your friends here! 🙂

Example `mydestroy()` output

```
Destroying Arena:
...unmapping arena with munmap()
```

Part 2

…the actual allocation

- The **`myalloc()`** function can be called by program code to request a memory allocation (exactly like **`malloc()`**)
- The allocated memory must reside inside the allocated arena
- Once allocated, the program should be able to read/write to that memory until it is freed using **`myfree()`**
- When **`myalloc()`** is called, your code will allocate a memory chunk. Let’s see what is inside…

Allocation chunk

- A **chunk** of allocated memory should consist of **two parts**:
  - A **header** containing metadata about the chunk, of type **`node_t`** (already defined in **`myalloc.h`**)
  - The actual area of memory **allocated to the application**
  - Notes:
    - The size of the chunk should account for the **space taken by the header**
    - Make sure you **do not return the header** to the program!

What about freeing?

- `myfree(void* ptr)` should free the memory allocated by myalloc()
- Note that `ptr` will not point to the header - it will point to the part of a chunk which is **reserved to the application**!
- If the program passes an address `a` to `myfree()`, to get the address of the chunk header, you will need to do something like:

  ```
  ((void *)a) – sizeof(node t)
  ```

Putting it all together

- Chunk headers of type **`node_t`** must be organized in a **doubly-linked list**
- In practice this means:
    - The chunk headers are allocated **within each chunk**
    - Each chunk header has pointers to the **next** and the **previous chunk**
- You must keep chunk headers both for **allocated** and **free** chunks of memory

Warning: you are in a `malloc()`-free zone

At no point in this assignment you are allowed to use `malloc()`

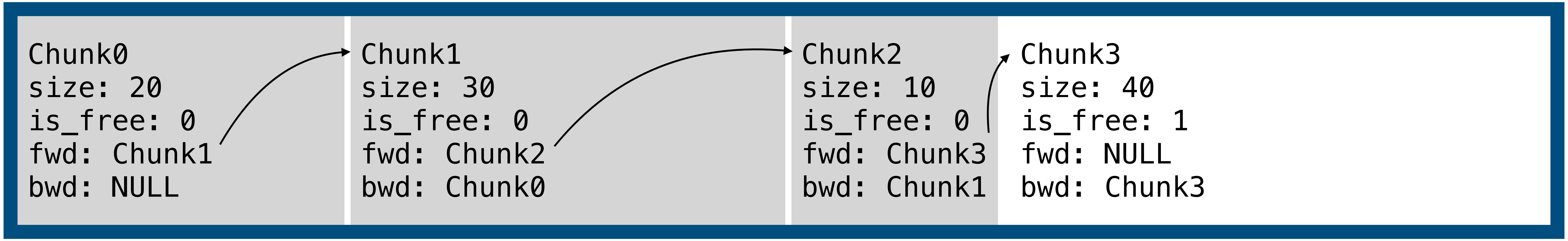
Some graphical examples

- Here is how an arena of size 100 looks after initialization

```
Chunk 0
size: 100
is_free: 1
fwd: NULL
bwd: NULL
```

Chunk header Chunk memory

- Here is what happens after **`myalloc(20); myalloc(30); myalloc(10)`**



- Here is what happens after **`myfree(chunk1)`**

`Chunk0`
`size: 20`
`is_free: 0`
`fwd: Chunk1`
`bwd: NULL`
	`Chunk1`
	`size: 30`
	`is_free: 1`
	`fwd: Chunk2`
	`bwd: Chunk0`
		`Chunk2`
		`size: 10`
		`is_free: 0`
		`fwd: Chunk3`
		`bwd: Chunk1`
			`Chunk3`
			`size: 40`
			`is_free: 1`
			`fwd: NULL`
			`bwd: Chunk3`

Note: if the program requests more memory than available, **`myalloc()`** should return null and set **`statusno`** to **`ERR_OUT_OF_MEMORY`**

Example output of myalloc():

```
Allocating memory:
...looking for free chunk of >= 4064 bytes
...found free chunk of 4064 bytes with header at 0x7f9655b92000
...free chunk->fwd currently points to (nil)
...free chunk->bwd currently points to (nil)
...checking if splitting is required
...splitting not required
...updating chunk header at 0x7f9655b92000
...being careful with my pointer arthimetic and void pointer casting
...allocation starts at 0x7f9655b92020
```

Example output of myfree():

```
Freeing allocated memory:
...supplied pointer 0x7fd515872120:
...being careful with my pointer arthimetic and void pointer casting
...accessing chunk header at 0x7fd515872100
...chunk of size 3808
...checking if coalescing is needed
...coalescing not needed.
```

Part 3

Free chunk splitting

- Suppose a requested allocation is **smaller than free chunk size** (e.g., **`myalloc(10)`** but the free chunk has size 25)
- In that case, the allocator must **split the chunk in two:** one allocated chunk of the requested size, and one free chunk of the remaining size
- I slightly lied to you above, because the actual size of the chunk is $\textit{requested space} + \textit{size of the chunk header}$
    - **Make sure you take that into account!**

Part 4

Placement of allocations

- The chunk list must be **logically ordered by address** of each chunk, in **increasing order**
- The **first chunk** in the list will be the chunk with the **lowest address**, and the **last chunk** will be the one with the **highest address**
- The ordering should be **naturally maintained** by properly implementing **`myalloc()`** and **`free()`**; in other words you should not need to implement any sorting function
- Your allocator should satisfy a **`myalloc()`** request by **allocating the first free chunk** which is **large enough** for the request

Part 4 example

- The following sequence should result in **`buff3`** being placed at the previous location of **`buff1`**:

```
buff1 = myalloc(64);
buff2 = myalloc(64);
myfree(buff1);
buff3 = myalloc(48);
```

Part 5

Free chunk coalescing

- Suppose freeing a chunk results in two or more **contiguous free chunks** in the list
- Then, your allocator should **coalesce those chunks** (i.e., merge them) into a **single larger chunk**

Part 5 example

- The following sequence should result in the memory allocator coalescing all three chunks in a single free chunk when the last **`myfree()`** is called:

```
buff1 = myalloc(64);
buff2 = myalloc(64);
buff3 = myalloc(64);
myfree(buff1);
myfree(buff3);
myfree(buff2);
```

Test cases

…find them in the code template!

- We are going to grade your code based on these test cases
- Hardcoding output to pass test cases will result in a **score of 0** and the student(s) being **referred for academic misconduct**

How to run tests?

- First, you may need to give execution permission to all test scripts:

```
chmod a+x test_all.sh
cd test-part1
chmod a+x test_part1.sh
cd..
cd test-part2
chmod a+x test_part2.sh
cd..
```

- …and so on

How to run tests? /2

- After that, you can run individual tests by doing (e.g., for part 1):

  ```
  cd test-part1
  ./test_part1.sh
  ```

- From the output, it will be clear which tests pass and which ones do not
- Look into the **`tests.c`** files in each test directory to understand what specifically is being tested
- To run all tests, run **`./test_all.sh`**
- Note, **`myalloc.c`** should be in the same directory as **`myalloc.h`** (no need to copy it into **`test-part1`** or any other subdirectory)

Grading rubric

- You get **3 points** for submitting your initial work during the lab
- The remaining 7 points are distributed as follow (not passing all test cases for one part will result in a score of 0 for that part; we will not assign partial scores):
  - Passing all part 1 test cases: **1 pt**
  - Passing all part 2 test cases: **2 pts**
  - Passing all part 3 test cases: **1 pt**
  - Passing all part 4 test cases: **1 pt**
  - Passing all part 5 test cases: **2 pts**

Deadline & delivery

- Given the upcoming midterm, you get two weeks to complete this lab
- Submit your **final version** on **the day before the next lab**, which is in **two weeks**
- Note, there is **no lab next week**
- You just need to deliver **`myalloc.c`**. Do not upload **`myalloc.h`**, test cases, or any other file

Advice

- Reach **Chapter 17 of the book** to get some **context**
- Note, the allocator presented here **works slightly differently** from the one on the book
- Familiarize yourself with **void pointers `(void*)`**, **structs**, and **pointer arithmetics**
  - The **test cases** have **useful examples**
- Familiarize yourself with **`size_t`** and remember that it is **unsigned**
- Always check for **`NULL`** pointers!

That is all… good luck!