

Project 2

Due Nov 7, 2022 by 11:59pm

Points 16


CMPSC 473 Project 2: Memory Allocator

(due Nov 7 11:59PM thru Github - NO EXTENSIONS WILL BE GIVEN!)

This page will continue to be updated. Please check back periodically for updates and announcements.

Please direct all your project-related questions/clarifications to the TAs through Canvas Email or Discussions.

Academic Integrity Statement:

The University defines academic integrity as the pursuit of scholarly activity in an open, honest and responsible manner. All students should act with personal integrity, respect other students' dignity, rights and property, and help create and maintain an environment in which all can succeed through the fruits of their efforts (refer to Senate Policy 49-20. Dishonesty of any kind will not be tolerated in this course. Dishonesty includes, but is not limited to, cheating, plagiarizing, fabricating information or citations, facilitating acts of academic dishonesty by others, having unauthorized possession of examinations, submitting work of another person or work previously used without informing the instructor, or tampering with the academic work of other students. Students who are found to be dishonest will receive academic sanctions and will be reported to the University's Office of Student Conduct for possible further disciplinary sanctions (refer to Senate Policy G-9). The Academic Integrity policy for the Department of Computer Science and Engineering for programming projects can be explicitly found at [EECS Academic Integrity Policy](https://www.eecs.psu.edu/students/resources/EECS-CSE-Academic-Integrity.aspx) .

(<https://www.eecs.psu.edu/students/resources/EECS-CSE-Academic-Integrity.aspx>). Please go through it carefully and do ask the instructor for any clarifications if something is not clear.

No exchange of code is permitted between teams. Both giver(s) and taker(s) will be punished harshly with a zero in the Project Grade, apart from being reported to the office of academic integrity for any further sanctions. Hence, make sure that your code is not visible to anyone outside your team!

Summary

In this project, you will learn about Memory Allocation. This project requires you to implement two memory allocation/de-allocation schemes. One of the jobs of the memory management subsystem is to service memory requests by matching the size of the request with a large enough hole, from which to satisfy the request.

Deadline

November 7, 2022, 11:59 PM

Goals

- Learn how to manage the memory (heap)
- Understand different allocation schemes

Overview

You will implement a memory allocator which supports **Buddy Allocation** and **Slab Allocation** schemes. The functioning/algorithm of each of these schemes has been discussed in class. Your memory allocator will support *my_malloc()* and *my_free()*, which is analogous to the C library's *malloc()* and *free()*. To test your memory allocator, the program will call *my_malloc()* and *my_free()*, based on the input file. The input file has a list of memory allocation requests and memory free requests. The generated output file has the result of your allocations. Your memory allocator will allocate memory only within the given starting address and size provided in *my_setup()*.

Hardware Model

This machine is a single CPU system. Only one thread will make requests to your memory allocator.

Allocation Policies

The size parameter is given only in *my_malloc* and not in *my_free*. However, it is necessary to know the size of what is being freed. This should be handled by using the header of the allocated segment to contain the size. However, programs would assume that all the space, starting from the first byte of the returned pointer, is available to be used. Hence, you should ensure that the pointer that *my_malloc()* returns points to the byte immediately after these HEADER_SIZE bytes (header) containing the size.

Buddy Allocator:

- The minimum chunk of memory that the Buddy System can allocate is MIN_MEM_CHUNK_SIZE bytes defined at *interface.h*.
- Maintain the free (holes) list sorted by increasing addresses individually for different sizes (of

powers of 2) as discussed in class.

- Whenever there is a choice of multiple holes that a scheme allows, always pick the hole with the lowest starting address.
- The HEADER_SIZE byte header is used to maintain the size of the allocated chunk within the chunk itself.

Slab Allocator:

- The number of objects per slab is fixed at N_OBJS_PER_SLAB defined at *interface.h*. Note that each slab will only contain the objects of the appropriate type/size. However the size of the allocated object itself should be accounted to include its HEADER_SIZE byte header. Hence, when using this scheme for allocating an object, there will be a HEADER_SIZE byte header for the object, and additionally a HEADER_SIZE byte header in the slab itself (where this object resides) which is allocated using Buddy.
- The Slab Descriptor Table itself is maintained as a separate data structure (you can use the c library malloc() for allocating it). Please use the data structure explained in class for the Slab Descriptor Table.
- Note that whenever you need to create a new slab, you need to call your Buddy Allocator from above to get the required slab (as discussed in class).

APIs to Implement

In interface.h, the header size, the minimum memory chunk size, and the number of objects per slab (for slab allocator) is defined. You should implement the following three interfaces in a file called interface.c, whose semantics are described below.

```
#define MEMORY_SIZE 8 * 1024 * 1024
#define HEADER_SIZE 8
#define MIN_MEM_CHUNK_SIZE 512
#define N_OBJS_PER_SLAB 64
```

```
// Allocation type
enum malloc_type {
    MALLOC_BUDDY = 0, // Buddy allocator
    MALLOC_SLAB = 1, // Slab allocator
};
```

```
void my_setup(enum malloc_type type, int mem_size, void *start_of_memory);
```

The purpose of setup() is to perform any initialization of variables that you may need, specify and give you the pointer (start_of_memory), the total amount of memory at your disposal, and specify the type of memory allocator.

The below functions may be called multiple times, depending on the input.

```
void* my_malloc(int size);
```

This function should allocate *size* bytes of memory from the *mem_size* chunk of memory that is available for you (from the *start_of_memory* pointer) using the specified allocation algorithm. On successful allocation, it returns a pointer to the start of allocated memory. (The pointer returned should: *start_of_memory* ≤ return ptr ≤ *start_of_memory*+*mem_size*.) If a request cannot be accommodated by the scheme, this function should return -1.

Note that the user programs expect that all the *size* bytes, starting from the first byte pointed to by the returned pointer, is available to it.

```
void my_free(void* ptr);
```

This function deallocates the memory segment being passed by the pointer. Note that when freeing, the resulting free segment should be merged with relevant neighboring holes (as per the algorithm/scheme) whenever possible to create holes of larger size. No *size* argument is being explicitly passed in this call.

You can implement additional functions (helper functions) and have additional variables and structures in *my_memory.c/h*, but note that the *main.c* will call only the above three functions. Also, note that there is no *name* argument in *my_malloc()* and *my_free()*. The memory chunk [*name*] of the input file is only for the program user to interact. Your internal memory allocator only works with the *size* and *ptr*, similar to the C library's *malloc()* and *free()*.

Input File Format

An input to the program is a text file which has the following format:

```
[name] [NumOps|Index] [Type] [Size]
[name] [NumOps|Index] [Type] [Size]
[name] [NumOps|Index] [Type] [Size]
...
```

where,

- **Name:** Name (specified as a single alphabetic character) of the chunk(s) of memory that is to be allocated
- **NumOPS/Index:**

In case of allocation request (type "M"), **NumOPS** indicates the number of allocation requests for specified **Size**.

In case of free request (type "F"), it indicates the **Index** of the corresponding **Name** to be freed.

- **Type:** Indicates the type of operation which can be either "M" for memory allocation or "F" for memory free.

- **Size:**

If the operation type is "M", it indicates the size of request to be allocated.

If the operation type is "F", this field is not used (ignored).

Sample Input and Output

① Basic Case

```
Z 5 M 1234
A 1 M 4321
C 3 M 19
Z 1 F 0
Z 3 F 0
```

can be interpreted as:

- Five memory allocation (M) requests, each requesting 1234 bytes of memory. The corresponding name and index of these 1234 byte chunks will be referred as Z1, Z2, ..., Z5.
- One memory allocation (M) request of 4321 bytes of memory. The corresponding name and index of this chunk will be A1.
- Three memory allocation (M) requests, each requesting 19 bytes of memory. The corresponding name and index of these 19 byte chunks will be referred as C1, C2, and C3.
- Free (F) request of the chunk named Z1.
- Free (F) request of the chunk named Z3.

Following is the MALLOC_BUDDY output of the above sample input (assuming HEADER_SIZE=8, MIN_MEM_CHUNK_SIZE=512):

MALLOC_BUDDY
Start of first Chunk Z is: 8 Start of first Chunk Z is: 2056 Start of first Chunk Z is: 4104 Start of first Chunk Z is: 6152 Start of first Chunk Z is: 8200 Start of Chunk A is: 16392 Start of Chunk C is: 10248 Start of Chunk C is: 10760 Start of Chunk C is: 11272 freed object Z at 8

freed object Z at 4104

- Since we have five allocation requests for name Z, we see five lines of output for Z. Since $1024 < \mathbf{1234} < 2048$, each chunk is allocated on the $0 + 8$ (HEADER_SIZE), $2048 + 8$, $4096 + 8$, $6144 + 8$, ... address.
 - Since we have one allocation request for name A, we see one line of output for A. Since $4096 < \mathbf{4321} < 8192$, the chunk is allocated on the $16384 + 8$ (HEADER_SIZE) = 16392 address. Addresses lower than that can't fulfill 4321 bytes of memory, according to the buddy allocation algorithm.
 - Since we have three allocation requests for name C, we see three lines of output for C. Since $19 < 512$ (MIN_MEM_CHUNK_SIZE), each chunk is allocated on the $10240 + 8$ (HEADER_SIZE), $10752 + 8$, $11264 + 8$ address.
 - Since we have a free request for chunk Z1, your memory allocator should free the chunk at address 8.
 - Since we have a free request for chunk Z3, your memory allocator should free the chunk at address 4104.
- * NOTE: the above explanation/algorithm is NOT how to actually allocate/free memory. The above explanation is for illustrative purposes only. See *Allocation Policies* below for more details.

② Same input, different output

```
A 1 M 250000
B 1 M 511000
C 1 M 150000
D 1 M 2000
```

can be interpreted as:

- One memory allocation (M) request of 250000 bytes of memory. The corresponding name and index of this chunk will be A1.
- One memory allocation (M) request of 511000 bytes of memory. The corresponding name and index of this chunk will be B1.
- One memory allocation (M) request of 150000 bytes of memory. The corresponding name and index of this chunk will be C1.
- One memory allocation (M) request of 150000 bytes of memory. The corresponding name and index of this chunk will be D1.

Following are the outputs of the above sample input (assuming HEADER_SIZE=8, MIN_MEM_CHUNK_SIZE=512):

MALLOC_BUDDY	MALLOC_SLAB
--------------	-------------

Start of first Chunk A is: 8 Start of Chunk B is: 524296 Start of Chunk C is: 262152 Start of Chunk D is: 1048584	Allocation Error A Allocation Error B Allocation Error C Start of first Chunk D is: 16
--	---

- The buddy allocator is straightforward.
 - The slab allocator allocates `N_OBJS_PER_SLAB` objects, which exceeds the `MEMORY_SIZE`. Therefore, it returns an error. (i.e., $250000 * N_OBJS_PER_SLAB > MEMORY_SIZE$; $511000 * N_OBJS_PER_SLAB > MEMORY_SIZE$; ...) The last request is the only request that is possible using the slab allocator, so name D gets allocated. Note that the start address is 16, which is `HEADER_SIZE + HEADER_SIZE`. See *Allocation Policies* below for more details.
- * NOTE: The above explanation/algorithm is NOT how to actually allocate/free memory. The above explanation is for illustrative purposes only. For example, even though the request size $* N_OBJS_PER_SLAB < MEMORY_SIZE$, there may be not enough space due to memory fragmentation.

③ Allocate-Free

L 3 M 1024 L 1 F 0 L 2 F 0 L 3 F 0 N 3 M 1023 N 1 F 0 N 2 F 0 N 3 F 0 E 4 M 1024
--

can be interpreted as:

- Three memory allocation requests for name L, each requesting 1024 bytes of memory. The corresponding name and index of these 1024 byte chunks will be referred as L1, L2, and L3.
- Free (F) request of the chunk named L1.
- Free (F) request of the chunk named L2.
- Free (F) request of the chunk named L3.
- ...

Following are the outputs of the above sample input (assuming `HEADER_SIZE=8`, `MIN_MEM_CHUNK_SIZE=512`):

MALLOC_BUDDY	MALLOC_SLAB
--------------	-------------

Start of first Chunk L is: 8 Start of first Chunk L is: 2056 Start of first Chunk L is: 4104 freed object L at 8 freed object L at 2056 freed object L at 4104 Start of Chunk N is: 8 Start of Chunk N is: 2056 Start of Chunk N is: 4104 freed object N at 8 freed object N at 2056 freed object N at 4104 Start of Chunk E is: 8 Start of Chunk E is: 2056 Start of Chunk E is: 4104 Start of Chunk E is: 6152	Start of first Chunk L is: 16 Start of first Chunk L is: 1048 Start of first Chunk L is: 2080 freed object L at 16 freed object L at 1048 freed object L at 2080 Start of Chunk N is: 16 Start of Chunk N is: 1047 Start of Chunk N is: 2078 freed object N at 16 freed object N at 1047 freed object N at 2078 Start of Chunk E is: 16 Start of Chunk E is: 1048 Start of Chunk E is: 2080 Start of Chunk E is: 3112
---	--

- For both buddy and slab allocator, you can see that the memory locations may be reused if freed. Therefore, it is important to free properly (according to the buddy/slab algorithm) so the memory can be re-allocated at a later time. You should follow the buddy/slab allocation/free policy precisely.

④ Header Consideration

Assuming HEADER_SIZE=8, MIN_MEM_CHUNK_SIZE=512:

Input	Output of MALLOC_BUDDY
A 3 M 1016	Start of first Chunk A is: 8 Start of first Chunk A is: 1032 Start of first Chunk A is: 2056
A 3 M 1021	Start of first Chunk A is: 8 Start of first Chunk A is: 2056 Start of first Chunk A is: 4104

You need to consider the header. Since $1016 + \text{HEADER_SIZE} \leq 1024$, they are allocated on the 1024 buddy chunk. However, $1021 + \text{HEADER_SIZE} > 1024$, so they are allocated on the next larger buddy chunk.

* NOTE: The above explanation is for illustrative purposes only. Please follow the buddy algorithm.

⑤ Additional Case 1

A 3 M 1500 A 2 F 0 B 5 M 600

- The interpretation is straightforward now.

Following are the outputs of the above sample input (assuming HEADER_SIZE=8, MIN_MEM_CHUNK_SIZE=512):

MALLOC_BUDDY	MALLOC_SLAB
<div>Start of first Chunk A is: 8 Start of first Chunk A is: 2056 Start of first Chunk A is: 4104 freed object A at 2056 Start of Chunk B is: 2056 Start of Chunk B is: 3080 Start of Chunk B is: 6152 Start of Chunk B is: 7176 Start of Chunk B is: 8200</div>	<div>Start of first Chunk A is: 16 Start of first Chunk A is: 1524 Start of first Chunk A is: 3032 freed object A at 1524 Start of Chunk B is: 131088 Start of Chunk B is: 131696 Start of Chunk B is: 132304 Start of Chunk B is: 132912 Start of Chunk B is: 133520</div>

- When allocating B chunks, the buddy allocator *fills up the hole* due to the memory free of A2.
- The slab allocator doesn't fill up the hole here, because N_OBJS_PER_SLAB number of slab objects are allocated, and the freed space is reserved for a future 1500 byte allocation.

⑥ Additional Case 2

Following is the input and the corresponding output (assuming HEADER_SIZE=8, MIN_MEM_CHUNK_SIZE=512):

Input	Output of MALLOC_BUDDY
<div>Z 3 M 1234 C 2 M 19 Z 1 F 0 Z 2 F 0 Z 3 F 0 U 5 M 1230 V 3 M 123</div>	<div>Start of first Chunk Z is: 8 Start of first Chunk Z is: 2056 Start of first Chunk Z is: 4104 Start of Chunk C is: 6152 Start of Chunk C is: 6664 freed object Z at 8 freed object Z at 2056 freed object Z at 4104 Start of Chunk U is: 4104 Start of Chunk U is: 8 Start of Chunk U is: 2056 Start of Chunk U is: 8200 Start of Chunk U is: 10248 Start of Chunk V is: 7176 Start of Chunk V is: 7688 Start of Chunk V is: 12296</div>

- Look at the allocation of name U. You can see that the 4104 is returned first (instead of 8). This is important! Why is my_malloc() of U not returning in 8, 2056, 4104, ... order? This is because the previous freed Z (the address 0 and 2048 buddy) gets combined into a larger buddy. So, the appropriate 2KB size buddy for the first allocation of U is the address 4096 buddy.

- This is why we put the *NOTE: The above explanation is for illustrative purposes only* reminder. You have to manage the actual buddy/slab state, and the allocations/deallocations are based on the state/algorithm, NOT a simple power of two calculation from the start.

⑦ Additional Case 3

Assuming HEADER_SIZE=8, MIN_MEM_CHUNK_SIZE=512:

Input	Output of MALLOC_SLAB
A 3 M 120 A 2 F 0 A 3 F 0 T 5 M 120 A 1 F 0 S 3 M 120 T 4 F 0 J 2 M 120 Q 3 M 119 R 2 M 120	Start of first Chunk A is: 16 Start of first Chunk A is: 144 Start of first Chunk A is: 272 freed object A at 144 freed object A at 272 Start of Chunk T is: 144 Start of Chunk T is: 272 Start of Chunk T is: 400 Start of Chunk T is: 528 Start of Chunk T is: 656 freed object A at 16 Start of Chunk S is: 16 Start of Chunk S is: 784 Start of Chunk S is: 912 freed object T at 528 Start of Chunk J is: 528 Start of Chunk J is: 1040 Start of Chunk Q is: 16400 Start of Chunk Q is: 16527 Start of Chunk Q is: 16654 Start of Chunk R is: 1168 Start of Chunk R is: 1296

Try to figure out the logic of the slab allocator by checking the input and corresponding output.
Drawing the memory layout of each step may help.

Additional Implementation Requirements/Restrictions

- You cannot use additional libraries.
- You cannot modify *main.c* file.
- Write a brief report (pdf of around 2 pages) explaining the design of your memory allocator, any implementation choices/restrictions, any functions not implemented, things learned, distribution of the workload within the team, etc. Commit and push the report with the code into your repo before the deadline.

Teamwork and Submission Instructions

1. You can work in teams (at most 2 members per team - they could be across sections), or

individually, for the project. You are free to choose your partner. But if either of you choose to drop out of your team for any reason at any time, each of you will be individually responsible for implementing and demonstrating the entire project and writing the project report by the designated deadline. (Please notify the TAs if this happens.)

2. If you have difficulty finding a partner, give your name to the TAs who will maintain an online list of students without partners to pair you up. Feel free to change teams from the prior projects. (It does not need to be the same from Project 1.)
3. Even though you will work in pairs, each of you should be fully familiar with the entire code and be prepared to answer a range of questions related to the entire project. It will be a good idea to work together at least during the initial design of the code. You should also ensure that there is a fair division of labor between the members. Remember that you both are owning up to the entire project that is being submitted, and you both are owning up to what works, what doesn't and the integrity of the entire project (not just your piece)!
4. Accept GitHub invitation and git clone your remote repository.
Link: <https://classroom.github.com/a/awy4gGcW> ➞ <https://classroom.github.com/a/awy4gGcW>
5. One person per team is required to accept the invitation link and then add the other team member to the repository. To add your team member to your private repository, go to your repository settings in GitHub. (Check previous P1 document for details.)
6. The code-bases will be tested on the W-135 Linux Lab machines located in Westgate. Regardless of where/how you develop/debug your code (laptop linux, Linux VMs, etc.), you need to ensure you test and run them in W135 machines for being graded!
7. Additional announcements would be sent out in the future to get your team information and the GitHub repository. Stay tuned!

Hints

- Re-read this document. Re-read and fully understand what is required from you.
- Try to go through *main.c* and understand what it does. You can see that `./proj2 <allocation-policy> <input-filename>` is how you run the program.
- Check how and what is `fprintf()`ed to the output file.
- Go through the lecture slides and the textbook again and understand how buddy and slab allocators work.
- Implement the buddy allocator first, verify that it works before moving to the slab allocator.
- Create your own corner-case inputs to test your memory allocator.

Submission Guidelines

- Ensure you commit your code often using the `git commit` and the `git push` commands (similar to how you had done P0~P1).
- The project (your final memory allocator implementation, including a brief report) must be pushed before the posted deadline. The last commit and push that you do prior to the deadline will be taken as your final submission.
- You may make multiple branches, but make sure you push your code into the main branch before the posted deadline. We will only clone from the main branch.
- Double-check and see if your submission is properly pushed by accessing your repository in GitHub.com
- Keep your repository private.
- **NO EXTENSIONS WILL BE ENTERTAINED.** ENSURE YOU PUSH YOUR CODE AND REPORT TO THE GITHUB REPOS BEFORE THE DEADLINE.

How Your Project is Graded

- We will clone the last push you made before the deadline, on the e5-cse-135 server. We will type *make* to compile your program. We will not fix any compile errors.
- We have additional test cases (they may be large, e.g., 500~ requests!) and use them as input to your memory allocator.
- After your memory allocator produces all outputs (MALLOC_BUDDY, MALLOC_SLAB), we will use the *diff* command to see any differences from our output. (i.e., Your output must match our results.)
- We will run the same test cases multiple times and see if you get the same result.
- Read the report and see how your team attempted/implemented the memory allocator.
- Check if *main.c* was changed or not. Do not change the *main.c* file.
- Have a short meeting with your team to demonstrate/ask about the project, the code, the project report, and the functionality.

Questions

For questions, please use the 'Discussions' tab of Canvas or use office hours. Please direct all project-related questions to the TAs. Please do NOT copy/paste code and post it in Discussions.