P4 Memory Allocator

Due Apr 30 by 11:59pm **Points** 50 **Submitting** a file upload
Available Apr 19 at 12am - May 10 at 8am 21 days

This assignment was locked May 10 at 8am.

Project 4: Memory Allocator

Re-submission Hard Deadline: 8am May 10, Monday

Notes from Mike

Difficulty: This project focuses on one of the major themes of Machine Organization Programming classes taught for many years at Universities all over the world and has gained a reputation for being a challenging project. I found two parts challenging when I first did this project:

1) Understanding how memory allocators work in general. First, watch the lecture videos and read section 9.9 – 9.9.6 of the book. It would be best if you learned this material in preparation for the final anyway. This project is designed to reinforce that learning.

We're doing something a little different with the headers, but the spirit of the implementation matches the implicit free list idea. Invest the time reading the provided code to see what data the header stores and how those data are accessed—the header struct stores additional information not stored by the implicit free list model. We would use this extra information to investigate memory utilization if we had more time.

2) Getting the C code that extracts the information I needed from the headers was tricky. I highly recommend writing helper functions so you only need to get this right once and can reusue this code. I included a list of suggested helper functions in the template file. These helper functions are not long. When I wrote them, they each fit on one line (multiple statements per line). Use the code provided in Mem_Dump() to learn to access each piece of data packed into the header.

3) The overall length of the code you need to write to complete this project is not long. My final solutions for both Mem_Alloc() and Mem_Free() were less than 25 lines long each - including comments. I will admit that those line counts include some lines with more than one statement but not unreasonable. For example, one line reads; int padding = 0; while ((size + padding) % 4) padding++; And, during the writing process, I used about 25 printf statements that I removed when my code was working. *Yes, I'm aware you can compute padding with one statement using simple math, but I was feeling really lazy at that moment.

Cheating: This project has been used in many Universities for this class for many years. There are lots of solutions available on the Internet. We all try to change some little thing to make sure you can't just turn in a solution you find on the Internet. Over the summer session, I caught 9 students out of 80 submitting work from the Internet for this project. I'm told my course has a reputation as a "gpa killer". However, at least part of this is that I'm good at catching cheaters. Getting a 0 on a project or exam will definitely hurt your gpa. Fair warning, your work will be compared with a collection of popular solutions from the Internet. It is very difficult to look at someone else's solution and not copy the structure. I would encourage you to avoid searching for other people's work and instead come to office hours if you would like to talk about strategies for completing this project.

Eight day project: I think this is a very reasonable project for the remaining time left in the semester. Please let me know if after you complete it if I'm wrong – so I can provide adequate time for students in the future. I think every aspect included in this project supports the learning goals for this course without extra work (memory allocations, fitting policies, freeing, and coalescing). We're skipping memory utilization analysis, and we're only implementing the first-fit policy.

It's important to me that I'm not overwhelming anyone during the last two weeks of the semester – I know you all are taking other classes with projects and have jobs that also require your attention. Please reach out to me if you have any concerns about finishing the project.

Corrections and Additions

1. None yet.

Learning Goals

The purpose of this project is to help you understand the nuances of building a memory allocator and further increase your C programming skills by working more with pointers.

Specifications

For this assignment, we give you the template for a simple shared library that implements the memory allocation functions malloc() and free(). Everything is present, except for the definitions of those two functions, called Mem_Alloc() and Mem_Free() in this library. Just replace the "Your code should go in here" in mem.c with your code. You may not use malloc() or include the stdlib library!

Files

In this project, we will be building a library file. This is C code that has been compiled into an object file (.o) or shared object file (.so), which can be included in another C program – the same way we have been #including <stdio.h> in all of our other programs. There are two included Makefiles with this project. The first Makefile takes care of the details required to compile mem.c into this library. The library files, mem.c, mem.h, and the first Makefile need to go in project directory.

Make a directory within the project directory called tests to store all the test files. Each test is a C program that #includes "mem.h", our mem library. This directory also comes with 1) a Makefile that will compile all .c files, 2) a run-tests.sh script that will run all of the tests, and 3) a testlist.txt file with a description of all the included tests.

- **Makefile** (https://canvas.wisc.edu/courses/230411/files/19715192/download?download_frd=1) : used for easy compiling
- **mem.c** (https://canvas.wisc.edu/courses/230411/files/19715188/download?download_frd=1) download: where you will write your code
- **mem.h** (https://canvas.wisc.edu/courses/230411/files/19715189/download?download_frd=1) download: header file for mem.c
- **tests** : the directory that holds the test files

Compiling and running

```
make              // use in the project directory to compile the memory allocator library
cd tests          // go to the tests directory
make              // compile all .c test files
./run-tests.sh    // run all the tests (change the permissions if necessary)
```

You can also run each test individually (take a look at run-tests.sh to see how it works).

Linking

The linking step needs to use the created library, libmem.so. So, you need to tell the linker where to find this file. Before you run any of the created dynamically linked executables, you will need to set the environment variable, LD_LIBRARY_PATH, so that the system can find your library at run time. Assuming you always run a testing executable from (your copy of) this same **tests/** directory, and the dynamically linked library (libmem.so) is one level up, that directory (to a Linux shell) is '../', so you can use the command (inside the tests directory):

```
export LD_LIBRARY_PATH=../
```

Or, if you use a *csh shell:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:../
```

If the setenv command returns an error "LD_LIBRARY_PATH: Undefined variable", do not panic. The error implies that your shell has not defined the environment variable. In this case, run:

```
setenv LD_LIBRARY_PATH ../
```

After you set the environment variable, you can run the tests as normal or run all of them using the run-tests.sh script.

Memory Allocation Background

Memory allocators have two distinct tasks. First, the memory allocator asks the operating system to expand the heap portion of the process's address space. In lecture and in the textbook, we covered one way to do this using sbrk(). In this project we will use mmap() - done for you and described below. Second, the memory allocator doles out this memory to the calling process. This involves managing a free list of memory and finding a contiguous chunk of memory that is large enough for the user's request; when the user later frees memory, it is added back to this list.

This memory allocator is usually provided as part of a standard library, and it is not part of the OS. To be clear, the memory allocator operates entirely within the virtual address space of a single process and knows nothing about which physical pages have been allocated to this process or the mapping from logical addresses to physical addresses as described in the upcoming OS lectures.

Every memory address that we see when our C program executes is virtual. i.e., the variable addresses that we see in our programs are mapped to actual physical addresses where those variables are stored in the main memory are different. e.g., If variable x is at address 0x3004, this doesn't mean that this variable x is at address 0x3004 in the main memory (physical RAM hardware). Instead, this means that the variable x is placed at the address 0x3004 within the virtual address space of this program (i.e., the addresses in this program starting at address zero) but mapped to an actual physical memory address that is different. We will cover more about virtual memory in the OS part during the last week of the course.

Classic malloc() and free() are defined as follows:

- **void *malloc(size_t size)**: malloc() allocates size bytes and returns a pointer to the allocated memory. **The memory is not cleared.**
- **void free(void *ptr)**: free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc() (or calloc() or realloc()). If free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.

Understand the template code

Mem_Init() and Mem_Dump() are provided for you. There are significant clues as to how to work with the headers within these functions. One of the first things you should do is study these functions to see how they access the data.

mem.c has working code for two functions: **Mem_Init(int sizeOfBlock, int allocate)** and **Mem_Dump()**. Look at them, and understand what they do, as well as how they accomplish their task. There is no need to change these functions. The implementation is slightly different from that described in the lecture and textbook, but it is clearly described in the comments.

Also, note the global variables global block header pointer 1) **first_header** is the head of our free linked list of memory chunks. 2) a BLOCK_HEADER struct to store the data for each header. Read the header

comments for the block header structure provided **very carefully** to understand the convention used. We're using a structure similar to the implicit free list described in the book and lecture, but we also include extra information, so this struct uses 8 bytes instead of just 4. Please read the comments in mem.c for the exact specification. 3) a global variable of type POLICY called 'policy'. This will store the type of fitting policy used to find the next free block to allocate. This semester we will only be implementing the first-fit policy.

Mem_Init(int sizeOfBlock, POLICY FIRST_FIT):

This sets up and initializes the heap space that the module manages. sizeOfBlock is the number of bytes that are requested to be initialized on the heap. The second argument, allocate, determines the allocation policy you would use. **Use the enum defined variables FIRST_FIT, BEST_FIT, WORST_FIT or NEXT_FIT to set the policy for your memory allocator. This semester we will only be implementing FIRST_FIT.**

This function should be called once at the start of any program before calling any of the other three functions. When testing your code you should call this function first to initialize enough space so that subsequent calls to allocate space via Mem_Alloc() can be served successfully. The test files we provide (as mentioned below) do the same.

When a process asks memory for the heap from the operating system, the operating system allocates memory in terms of pages. A page is the smallest unit of data for memory management in a virtual memory operating system. It is a fixed-length contiguous block of virtual memory. Note that Mem_Init() rounds up the amount of memory requested in units of this page size. Because of rounding up, the amount of memory initialized may be more than sizeOfBlock. You may use all this initialized space for allocating memory to the user. This is a more modern approach than used by sbrk and allows us to take advantage of hardware upgrades made available since the beginning of C.

Once Mem_Init() has been successfully called, first_header will be initialized as the first and header in the free list contains points to a single free chunk of memory. You will use this list to allocate space to the user via Mem_Alloc() calls. A second header is placed to mark the end of the heap.

Mem_Init() uses the mmap() system call to initialize space on the heap. If you are interested, read the man pages to see how that works.

Mem_Dump():

This is used for debugging; it prints a list of all the memory blocks (both free and allocated). It will be incredibly useful when you are trying to determine if your code works properly. As a future programming note: take notice of this function. When you are working on implementing a complex program, a function like this that produces lots of useful information about a data structure can be well worth the time you might spend implementing it.

Invest some time learning how the Mem_Dump() function access data using the block headers. This is extremely useful when you write helper functions.

Implement Mem_Alloc() and Mem_Free()

Note: Do *not* change the interface. Do *not* change anything within file mem.h. Do *not* change any part of functions Mem_Init() or Mem_Dump().

Write the code to implement Mem_Alloc() and Mem_Free(). Use the **first fit** algorithm when allocating blocks with Mem_Alloc(). When freeing memory, always **coalesce** both adjacent memory blocks if they are free. First_header is the free list structure as defined and described in mem.c. The concept **is based on the model described in your textbook in section 9.9.6; however, our implementation packs the allocate bit with a pointer to the next header, and stores the exact size requested by the user.** Here are definitions for the functions:

void *Mem_Alloc(int size):

Mem_Alloc() is similar to the library function malloc(). Mem_Alloc() takes as an input parameter the size in bytes of the memory space to be allocated, and it returns a pointer to the start of that memory space (i.e, this means, a pointer to the start of the first useful byte, **after the header**). The function returns NULL if there is not enough contiguous free space within sizeOfBlock allocated by Mem_Init() to satisfy this request. Mem_Alloc() is required to return **4-byte aligned chunks of memory**. For example, if a user requests 1 byte of memory, the Mem_Alloc() implementation should return 4 bytes of memory so that the next free block will also be 4-byte aligned. To debug whether you return 4-byte aligned pointers, you could print the pointer this way:

- printf("%08x", ptr)
- The last digit should be a multiple of 4 (that is, 0, 4, 8, or C). For example, 0xb7b2c04c is okay, and 0xb7b2c043 is *not*

Once the appropriate free block is located, we could use the entire block for the allocation. The disadvantage is that it causes internal fragmentation and wastes space. So, we will split the block into two. The first part becomes the allocated block, and the remainder becomes a new free block. Before splitting the block there should be enough space left over for a new free block. i.e., the header and its minimum payload of 4 bytes, otherwise do not split the block.

The size of a block does NOT include the size of the header. We store the exact size requested by the user. Note that this is different than the allocator blocks we saw in the lecture which includes the size of the header and padding in the block size.

int Mem_Free(void *ptr):

Mem_Free() frees the memory object that ptr points to. Just like with the standard free(), if ptr is NULL, then no operation is performed. The function returns 0 on success and -1 if the ptr was not allocated by Mem_Alloc(). If ptr is NULL, also return -1. For the block being freed, always **coalesce** with its adjacent blocks if either or both of them are free.

Test the Code

It is time to test if your Mem_Alloc() and Mem_Free() implementations work. You will need to write a separate program that links in your shared library and makes calls to the functions within this shared library. We've already written a bunch of small programs that do this, to help you get started, located in the tests folder. Copy all the files within this directory into a new directory within the one containing your shared library. Name your new directory **tests**.

In this directory, file testlist.txt contains a list of the tests we are giving you to help you start testing your code. The tests are ordered by difficulty. **Please note that these tests are not comprehensive for testing your code**. Though they cover a wide range of test cases, there will be additional test cases that your code will be tested against. You will need to modify these files by adding additional calls to Mem_Dump(). You will have to read and understand this output to verify your code works. We cannot provide exact output files because Mem_Init() may reserve a different part of the address space for your heap. (Note from Mike – I'm giving you all the test files I've written up to this point. You should modify these and write additional test files to test all aspects of your memory allocator.)

Design a New Test

Create a new C program that tests whether simple Mem_Alloc() calls work. The test should determine if a single allocation, followed by a call to Mem_Free() does the right thing. After you have debugged enough to know that it works correctly, add to this same C program a test case to make sure that Mem_Free() does the right thing if passed a bad pointer. A bad pointer is one with the NULL value or an address of memory space *not* allocated by Mem_Alloc(). Name this testing program free-tests.c. The main purpose of this part is to help you get started with writing your own tests for testing your memory allocator.

Hints

- Invest the time to learn how the BLOCK_HEADER struct stores the data. It contains
  - The address of the next header
  - An alloc bit
  - The exact size requested by the user
  - Note the address and alloc bit are packed into the same void*
- Write helper functions: isBusy, isFree, getter and setter functions for all data contained in the header.
- Take the time to study the first_header initialization in Mem_Init().
- Take the time to study how the data is accessed in Mem_Dump() using the block headers.
- Be very careful of pointer arithmetic (int*)+1 changes the address by 4, (void*)+1 or (char*)+1 changes it by 1. What does (block_head*)+1 change it by?
- Bitwise operations can make it easy to extract data from the void* packed pointer data.
- Follow the format of the provided C test files to write your own.

- Verify return values are correct for all function calls made.

Turn in

- Upload your mem.c file to Canvas.

Style

1. Use meaningful variable names. Either use underscores for multi-word variables or CamelCase. Be consistent.
2. Be consistent with capitalization. For example, capitalize function names, use all caps for structs, enums, and #defines, and use lower case for variables.
3. Indent to match scope. Use spaces instead of tabs.
4. Organize your code as follows:
    1. #include <>
    2. #include ""
    3. #defines
    4. Data Types (e.g., structures and typedef)
    5. Global variables – (use of global variables only when necessary)
    6. Function Prototypes
    7. Code - main() should be either first or last
5. Comments
    1. Describe what each block of code is trying to do.
    2. Good variable names may make this description easier.
    3. For functions, describe the purpose of the function and indicate the purpose of input parameters, output parameters, and the return value.
6. Multi-statement lines – should include statements that logically go together.
    1. For example: if (ptr == NULL) { printf(“error message\n”); exit(1); }

Project 4

Criteria	Ratings		Pts
1. Compilation	**8 pts** **Code complies**	**0 pts** **Code does not compile**	8 pts
2. Execution test: alloc8.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts
3. Execution test: alloc1.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts
4. Execution test: alloc4080.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts
5. Execution test: alloc4078.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts
6. Execution test: alloc_nospace.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts
7. Execution test: writeable.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts
8. Execution test: align.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts
9. Execution test: mem_free.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts
10. Execution test: coalesce.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts
11. Execution test: firstfit.c	**3 pts** **Correct answer**	**0 pts** **Incorect answer**	3 pts

Criteria	Ratings		Pts
12. Execution test: free2.c	**2 pts** **Correct answer**	**0 pts** **Incorect answer**	2 pts
13. Execution test: coalesce4.c	**2 pts** **Correct answer**	**0 pts** **Incorect answer**	2 pts
14. Execution test: coalesce6.c	**2 pts** **Correct answer**	**0 pts** **Incorect answer**	2 pts
15. Execution test: full.c	**2 pts** **Correct answer**	**0 pts** **Incorect answer**	2 pts
16. Execution test: bigtest.c	**2 pts** **Correct answer**	**0 pts** **Incorect answer**	2 pts
17. Execution test: sanity.c	**2 pts** **Correct answer**	**0 pts** **Incorect answer**	2 pts
			Total Points: 50