p3: Dynamic Memory Allocation

1. Logistics

1. This project must be done individually.

   It is academic misconduct to share your work with others in any form including posting it on publicly accessible web sites, such as GitHub.

   It is academic misconduct for you to copy or use some or all of a program that has been written by someone else.

2. All work for this project is to be done on one of the CS department's instructional Linux machines. You're welcome to remotely login using ssh or MobaXterm as was shown in lecture (or any other way that you figure out on your own).
3. All projects in this course are graded on CS department's instructional Linux machines. It is your responsibility to make sure that your code runs as you expect on these machines.

2. Learning Goals

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

3. Specifications

For this assignment, you will be given the structure for a simple shared library that is used in place of the heap memory allocation functions **`malloc()`** and **`free()`**. You'll code two functions in this library, named **`Mem_Alloc()`** and **`Mem_Free()`**.

3.1. 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 by using the `sbrk()` system call, but we'll use `mmap()` to simulate a heap in the memory mapped area. Second, the memory allocator doles out this memory to the calling process that requests heap memory. This involves managing a list of free memory blocks. When the process requests memory the allocator searches that list for a free block that is large enough to satisfy the request and marks it as allocated. When the process later frees memory, the allocator marks that block as free.

This memory allocator is usually provided as part of a standard library rather than being part of the operating system. Thus, 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 virtual addresses to physical addresses.

The C programming language defines its allocator with the functions `malloc()` and `free()` found in "stdlib.h" as follows:

- `void *malloc(size_t s)`: allocates s bytes and returns a pointer to the allocated memory. The memory is not cleared.
- `void free(void *ptr)`: frees the memory pointed to by `ptr` that was previously allocated by `malloc()` (or `calloc()` or `realloc()`). The behaviour is undefined if ptr is a stray pointer or if an attempt is made to free an allocation more than once. If `ptr` is `NULL`, the function does nothing and simply returns.

3.2. Understand the Code

Copy the entire contents from following directory into your working directory for this assignment:

```
/p/course/cs354-skrentny/public/code/p3
```

In this directory you'll find the files named "Makefile", "mem.c" and "mem.h" as well as a directory named "tests". In "mem.c" is the completed code for two functions: `Mem_Init(int sizeOfRegion)` and `Mem_Dump()`. Look at these functions to understand what they do and how they do it. Also note the global block header pointer `first_block` is the head of our list of memory blocks, where each block would be marked as either free or allocated. **Very carefully read** the comments for the provided block tag structure to understand the conventions used in this program. These functions we've completed are described below:

`Mem_Init(int sizeOfRegion)`

This function sets up and initializes the "heap" space that the allocator will manage. `sizeOfRegion` is the number of bytes desired for the heap.

This function should be called **once at the start of any main program** before calling any of the other allocator functions. In your main programs to test your code call this function first to initialize enough space so that subsequent calls to `Mem_Alloc()` function properly. The test main programs we've provided (discussed below) already do this.

Note that for improved performance, `Mem_Init(int sizeOfRegion)` rounds up the amount memory requested to the nearest page so it is possible that more memory might be allocated than originally specified by `sizeOfRegion`. All of this memory is initialized as the first and only block in the free list to be used by your allocator and is accessed via `first_block`, which will be pointing to that block's header. This is the beginning of the implicit free list that your allocator uses to allocate blocks via `Mem_Alloc()` calls. Your allocator will need to divide this block into smaller pieces as memory is requested.

`Mem_Dump()`

This function is used for debugging. It prints the list of all the memory blocks (both free and allocated). **Use this to determine if your code works properly.** Implementing functions like this are very helpful and well worth your time when working on complex programs. It produces lots of useful information about the data structure so take a closer look at what it does. Currently the function only displays information stored in the headers of the blocks. It is recommended that you

extend the implementation to display the information stored in the footers of the free blocks too. No points will be deducted if you chose not to.

3.3. Implement the Allocator

Note: Do not change the interface. Do not change anything within the file "mem.h". Do not change any part of the function `Mem_Init()`.

Write the code to implement `Mem_Alloc()` and `Mem_Free()`. Use **best fit placement policy** when allocating blocks with `Mem_Alloc()` and **splitting the block** if possible. If there is a tie for best fit, then you can select any block. When freeing memory, **use immediate coalescing** with the adjacent memory blocks if they're free. Recall that `first_block` is the beginning of the implicit free list structure as defined and described in the file "mem.c". It uses ideas discussed in the lecture (textbook section 9.9.11) of using headers/footers in order to allocate and free blocks. We'll also use **address ordering** of the blocks in this list. The functions you'll code are described below:

`void *Mem_Alloc(int size):`

This function is similar to the C library function `malloc()`. It returns a pointer to the start of allocated payload of `size` bytes. The function returns `NULL` if there isn't a free block large enough to satisfy the request. Note we won't request more memory from the OS than what was originally allocated by `Mem_Init()`. `Mem_Alloc()` returns single word (4 bytes) aligned chunks of memory, which improves performance. For example, a request for 1 byte of memory uses 4 bytes - 1 byte for the payload and 3 bytes for padding to achieve word alignment. To verify that you're returning 4-byte aligned pointers, you can print the pointer in hexadecimal this way:

```
printf("%08x", ptr)
```

The last digit displayed should be a multiple of 4 (that is, 0, 4, 8, or C). For example, 0xb7b2c04c is okay, and 0xb7b2c043 indicates a problem with your alignment.

Once the best fitting free block is located we'll split the block in two to minimize internal fragmentation. The first part becomes the allocated block, and the remainder becomes a new free block. If the remaining free block doesn't meet the minimum size (the header plus its minimum payload of 4 bytes) then don't split the block. Note, you are not to remove the final allocated block from the list of blocks.

`int Mem_Free(void *ptr):`

This function is similar to the C library function `free()`. It frees the block of heap memory containing `ptr`'s payload and returns 0 to indicate success. If `ptr` is `NULL`, not within the range of memory allocated by Mem_Init() or is not 4 byte aligned the function just returns -1. When freeing a block you must coalesce it with its adjacent blocks that are free. Remember, this leaves only one header and one footer in the coalesced free block.

3.4. Test the Code

We have provided a file, named "Makefile", that is used to compile your code in "mem.c" and "mem.h" into a shared library named "libmem.so". To compile, enter the command `make` on the Linux command line while you are working in your project directory.

Once the shared library is created, you can then test if your allocator works properly by writing a test main program. Your test program links this shared library, and makes calls to the functions `Mem_Alloc()` and `Mem_Free()`. We've already written some small programs that do this, and to help you get started in writing your own tests. Our test programs are in the "tests" subdirectory that you copied along with a second `Makefile` used to compile those test using the same `make` command.

You'll also find the file "testlist.txt" that contains a brief description of the tests we provided ordered by increasing difficulty. **Please note that these tests are not exhaustive.** They cover a good range of test cases, but there will be additional tests that we'll use to grade your code. To compile these tests change your working directory to the "tests" subdirectory and enter `make` on the Linux command line. This will make executables for all the test programs in this directory and link them to the shared library that you've compiled beforehand.

4. Notes and Hints

- Keep in mind that the value of `size_status` includes the space for the block tags. Make use of `sizeof(block_tag)` to set the appropriate size of requested block.
- **Double check your pointer arithmetic**. `(int*)+1` changes the address by 4, `(void*)+1` and `(char*)+1` changes it by 1. What does `(block_tag*)+1`change it by?
- For main programs that you write to test your allocator, make sure you call `Mem_Init()` first to allocate the initial space.
- Check return values for all function calls to make sure you don't get unexpected behavior.
- Incrementally build up your program by testing against the tests in the order that they are listed in "testlist.txt". For e.g, focus on making sure your best fit algorithm is correct before you implement coalescing.
- You may want to use '`(size_status & 1) == 0`' to check if a block is busy or not. It is important that you put a parenthesis around '`size_status & 1`' other wise the compiler will interpret the code as '`size_status & (1 == 0)`'

5. Requirements

- **Do *not* use any of C's allocate or free functions in this program!** You will not receive credit for this assignment if you do since that simply avoids the purpose of this program.
- Your program must follow these Style Guidelines.
- Your program must follow these Commenting Guidelines. Keep the function header comments we've put in the skeleton code.
- Your programs must operate exactly as specified.
- We'll compile your programs (using the Makefiles we have provided) on the CS Linux lab machines. It is your responsibility to ensure that your programs compile on these machines, **without warnings or errors**. Points are deducted if compiling your programs on the CS Linux lab machines produces warnings and/or errors.

7. Submitting Your Work

Submit the following **source files** under Project 3 in Assignments on Canvas on or before the deadline:

1. mem.c

It is your responsibility to ensure your submission is complete with the correct file names having the correct contents. The following points will seem obvious to most, but we've found we must explicitly state them otherwise some students will request special treatment for their carelessness:

- **You will only receive credit for the files that you submit.** You will not receive credit for files that you do not submit. Forgetting to submit, not submitting all the listed files, or submitting executable files or other wrong files will result in you losing credit for the assignment.
- **Do not zip, compress or submit your files in a folder.** Submit only the text files as listed.
- **Make sure your file names exactly match those listed.** If you resubmit your work, Canvas will modify the file names by appending a hyphen and a number (e.g., mem-1.c) and these Canvas modified names are accepted for grading.

Repeated Submission: You may resubmit your work repeatedly until the deadline has passed. **We strongly encourage you to use Canvas as a place to store a back up copy of your work.** If you resubmit, you must resubmit all of your work rather than updating just some of the files.