Assignment 5

CS/ECE 354 - Spring 2016

DUE: April 22nd (Friday) at 9 am

1. Collaboration Policy

For this assignment, you may work in pairs (2 people). All students (whether working in a pair or not) must individually turn in the assignment. Therefore, 2 copies of the assignment will be turned in for those working as a pair. The grader will choose to grade only one copy of a pair's work, and assign the same score to both students.

If one member of a pair does something unexpected with respect to turning in the assignment (for example: bad copy of the source code, late, or perhaps not turning in portions of the assignment), then *both* of the pair get the same penalty and the same score.

If working in a pair, the names and sections of *both* students must appear as comments at the top of all files of the turned in assignment.

In addition, students who work in a pair must turn in an extra file that identifies the pair. Details are given in the section on Handing In the Assignment.

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 get familiar with using Makefiles.

3. Specifications

For this assignment, you will be given the structure 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.

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 calling either sbrk() or mmap(). 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.
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()). Otherwise, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.

3.2. Understand the code:

Create a directory for this assignment. The source code files you will need are in the directory:
`/p/course/cs354-common/public/spring16/code/p5`

Copy the files `Makefile`, `mem.c` and `mem.h` to your own directory. In mem.c is fully working code for two functions: **`Mem_Init(int sizeOfRegion)`** and **`Mem_Dump()`**. Look at them, and understand what they do, as well as how they accomplish their task. Also note the global block header pointer **`list_head`** which is the head of our free linked list of memory chunks. Read the header comments for the block header structure provided **very carefully** to understand the convention used.

`Mem_Init(int sizeOfRegion):`
This sets up and initializes the heap space that the module manages. sizeOfRegion is the number of bytes that are requested to be initialized on the heap.
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.
Note that Mem_Init(int sizeOfRegion) rounds up the amount memory requested in units of the page size.
Because of rounding up, the amount of memory initialized may be more than sizeOfRegion. You may use all of this initialized space for allocating memory to the user.
Once Mem_Init(int sizeOfRegion) has been successfully called, list_head will be initialized as the first and only header in the free list which points to a single free chunk of memory. You will use this list to allocate space to the user via `Mem_Alloc()` calls.
Mem_Init(int sizeOfRegion) 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.

3.3. Implement malloc and 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 a **best fit** algorithm when allocating blocks with Mem_Alloc(). When freeing memory, always **coalesce** with the adjacent memory blocks if they are free. list_head is the free list structure as defined and described in mem.c. **It is based on the model described in your textbook in section 9.9.6 (except our implementation has an additional next pointer in the header in order to make it easier to traverse through the free list structure).** 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. The function returns NULL if there is not enough contiguous free space within sizeOfRegion allocated by Mem_Init to satisfy this request. For better performance, Mem_Alloc() is 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* okay.

Once the best fit 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 we don't split the block.

`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.

3.4. Test the Code:

You have a provided Makefile that compiles your code in mem.c and mem.h into a shared library called libmem.so. To do the compilation, the command is

```
make mem
```

With this shared library, it is time to test if your Mem_Alloc() and Mem_Free() implementations work. This implies that 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. Our programs and a second Makefile are in

`/p/course/cs354-common/public/spring16/code/p5/tests`

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.
When you run *make* within the tests directory, it will make executables of all the C programs in this directory.
The linking step needs to use your 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 ../
```

3.4. Design a New Test:

Create a new C program that tests whether simple Mem_Free() 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 freetests.c.

5. Hints

- Always keep in mind that the value of size_status excludes the space for the header block. Make use of sizeof(block_header) to set the appropriate size of requested block.
- It is highly recommended that you write small helper functions(test them first) for common operations and checks such as: isFree, setFree, setAllocated etc.
- Double check your pointer arithmetic. (int*)+1 changes the address by 4, (void*)+1 or (char*)+1 changes it by 1. What does (block_header*)+1 change it by?
- For any tests that you write, make sure you call Mem_init() first to allocate sufficient space.
- Check return values for all function calls to make sure you don't get unexpected behaviour.

6. Handing in the Assignment

Turn in files mem.c and freetests.c. Copy these files into your handin directory. Your handin directory for this project is
`/p/course/cs354-common/public/spring16.handin/<yourloginID>/p5`

<yourloginID> is the username of your CS account.
Your Handin Folder should have the following files:

1. mem.c
2. freetests.c

If you are working as part of a pair, you must turn in an extra file. This file will contain the names and sections of **both** students in the pair. As an example, if Haseeb worked with Urmish on this assignment, the contents of the extra file for both Haseeb and Urmish would be
Haseeb Tariq section 1
Urmish Thakker section 2
The name of this file is specialized to help the 354 automated grading tools identify who worked together. This file name is composed of the CS logins of the partners separated by a period. The file name is of the form <login1>.<login2>. Haseeb's login is haseeb, and Urmish's login is uthakker. The file name that both use will be haseeb.uthakker. Please have both partners use the same file name. It does not matter which partner's name is first within this file name.

7. Requirements

1. Within the comment block at the top of mem.c, add a new line of the form:
   * MODIFIED BY: name, section#, partnername, section#
2. Your program is to continue the style of the code already in the file. Use the same types of comments, and use tabs/spaces as the existing code does.
3. Document your added functions with inline comments!
4. Your programs must compile on the CS Linux lab machines as indicated in this specification *without warnings or errors*.
5. **Do *not* use any stdlib allocate or free functions in this program!** The penalty for using malloc() (or friends) will be no credit for this assignment.

8. About Copying Code and Academic Misconduct

Don't cheat. Read this link carefully.
*Do **not** post your assignment solutions (or drafts) on any publicly accessible web sites. This specifically includes GitHub. It is academic misconduct to post your solution.*
For almost any C program that does something useful, someone has already written this program and further, has posted it for others to use. These programs do not do much that is useful, and are not likely posted anywhere. Still, it is academic misconduct for you to copy or use some or all of a program that has been written by someone else.
The penalty for academic misconduct on this assignment (and all CS/ECE 354 assignments) will be a failing grade in the course. This penalty is significantly more harsh than if you simply do not do the assignment. You will gain much more by doing the assignment than by copying, possibly modifying, and turning in someone else's effort.

*Do **not** post your assignment solution (or drafts) on any publicly accessible web sites. This specifically includes GitHub. It is academic misconduct to post your solution.*