Project 5

Corrections and Additions

1. There is a comment above Mem_Alloc that says to use the best free block. Instead you should use the given fit policy (best, first or worst).

1. Learning Goals

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

2. 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. Just replace the "Your code should go in here" in mem.c with your code.

2.0. Files

- Makefile : used for easy compiling
- mem.c : where you will write your code
- mem.h : header file for mem.c
- tests/ : holds the test files

2.1. 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.

Every memory that we see when our C program is executed is virtual. i.e., the variable addresses that we see in our programs and the actual physical addresses that those variables are placed 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 (RAM). 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 the actual physical memory address of this variable will be different. To understand more about virtual memory, you may read this .
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.

2.2. Understand the code

Create a directory for this assignment. The source code files you will need are listed above.

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 sizeOfBlock, int allocate)** and **Mem_Dump()**. Look at them, and understand what they do, as well as how they accomplish their task. The implementation is slightly different from described in the lecture and textbook, but is clearly described in the comments. 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 sizeOfBlock, int allocate):

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. **If it is 0 follow best fit, if it is 1 follow first fit and if it is 2 follow worst 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. Read more about pages here . 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.

Once Mem_Init 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 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.

2.3. 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/worst/first 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. (i.e, This means, pointer to the start of the first useful byte, after header.)The function returns NULL if there is not enough contiguous free space within sizeOfBlock 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*

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 we don't split the block.

The size of a block does NOT include the size of the header. Note that this is different than the allocator blocks we saw in the lecture which includes the size of the header 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.

2.4. 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, located in the tests folder above.
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 can run the command make within the tests directory, which 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 ../

After you set the environment variable, you can run the tests as normal.

2.5. 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 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.

3. Hints

- Always keep in mind that the value of **size_status** (in the block_head) **excludes** the space for the header block.
- It is highly recommended that you write small helper functions(test them first) for common operations and checks such as: isFree(), setFree(), setAllocated()
- Double check your 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?
- 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 behavior.

4. Requirements

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