CS3841 Lab 8: A Dynamic Memory Manager

Dr. Walter Schilling

Due: Friday, November 9, 2012 17:00 CDT

1. Lab Objectives

1. Understand the operation of dynamic memory management by implementing a dynamic memory manager.
2. Practice C development in a UNIX environment.
3. Construct software which uses C structures and pointer references.
4. Analyze the performance of a system using Dynamic memory allocation
5. Port C code from one platform to another platform.

2. Introduction

The purpose of this lab is to develop a dynamic memory manager. This memory manager will replace the functionality of the malloc and free routines built into the operating systems. In doing this, you will be exposed to linked lists in C, performance analysis, and other relevant topics. Additionally, this lab will demonstrate how file allocation system might function for a contiguous allocation scheme.

There are numerous algorithms that have been used for dynamic memory allocation, with each algorithm and implementation tailoring itself to a given system. The algorithm you will be implementing in this lab is similar to that provided in Kernighan and Ritchie's The C Programming Language (Englewood Cliffs, N.J.: Prentice-Hall, 1988). However, it has been slightly simplified to suite your abilities.

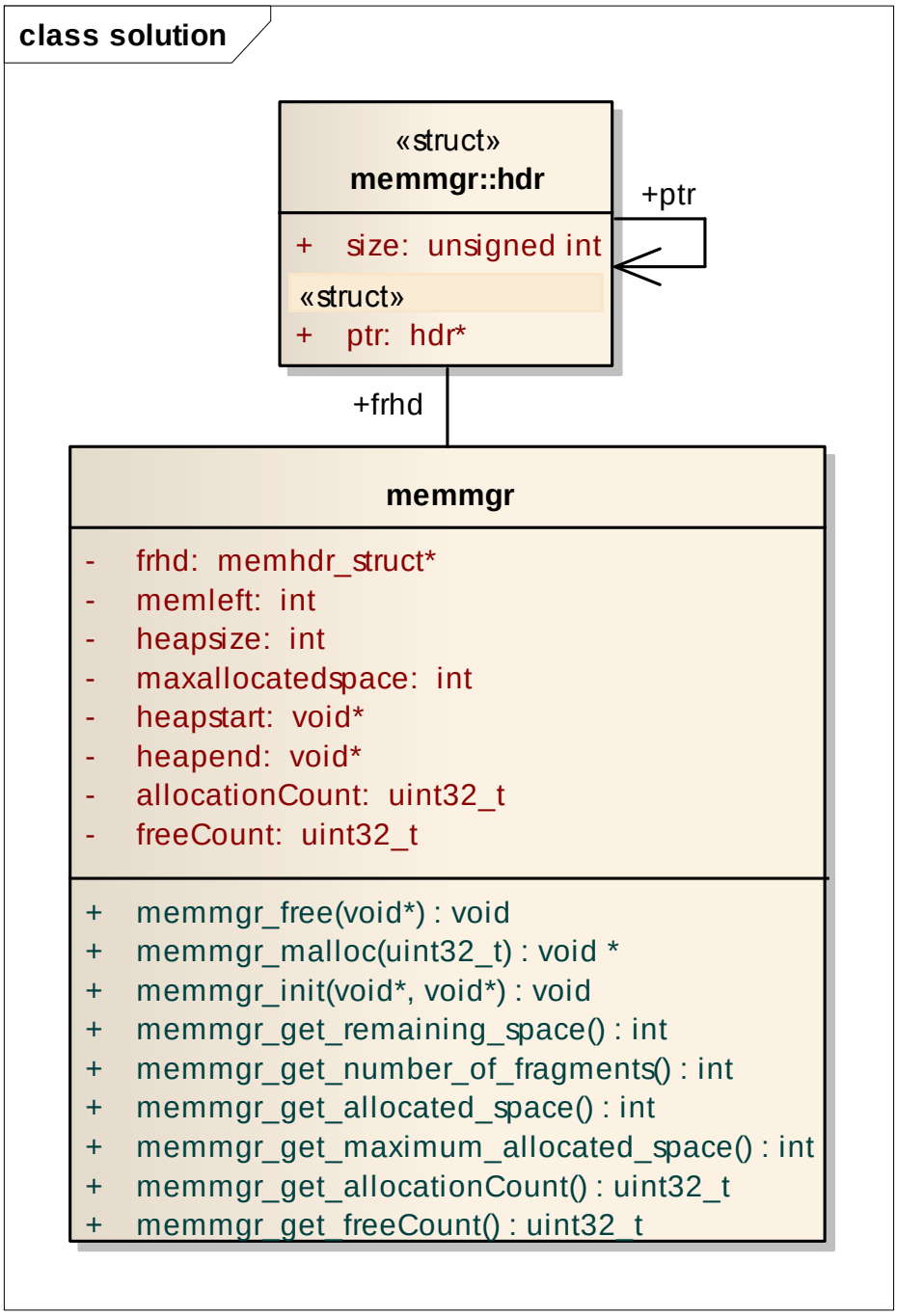
In general, a dynamic memory manager must meet numerous requirements. These requirements, in no specific order, include

1. All internal structures and linkage must be hidden from the application. The application only should be required to request memory from the memory manager.
2. The order of calls to malloc() and free() should be independent of each other. The manager should not make any assumptions that calls to free will be the opposite of calls to malloc.
3. The free method prevents fragmentation of freed memory. If multiple memory blocks which are contiguous are freed, they shall be combined to form one larger block.
4. Memory manager overhead and execution time shall be minimized.
5. The memory manager shall return an error condition if it is not possible to allocate memory.

3. Assignment

This is a lab that is to be completed with a lab partner. To start, you will need to read the article "Memory Allocation in C" by Leslie Aldridge, available at http://www.embedded-systems.com/design/opensource/209600563. This article explains how a simple, malloc library functions. It also provides C source code for a simple dynamic memory allocator.

After you have completed this reading, take a look at the UML design in the figure below. In essence, this is a UML representation for this design, and you should use it as a starting point for the development of your memory manager. In essence, your implementation will be based upon the design in the article except that the names will be different, you must provide a slightly different initialization method, and you need to provide a few diagnostic methods which determine how many free blocks there are (an indication of fragmentation), the allocated space, and the maximum allocated space overall.



Unlike the malloc which is built into the C library, our malloc requires initialization. The initialization defines the start and end of the memory space that is to be used for dynamic memory allocation. These are void pointers. In terms of the article, the initialization method will define the start and end of the heap.

4. Lab Activities

Your first responsibility is to implement the dynamic memory manager in compliance with this program. To facilitate this, a set of test routines has been provided on the course website which will exercise your memory manager. As with the Linked List lab, these are written in CppUnit.

Once you have completed implementation of your program, modify your first word counter to use this memory manager. In doing this, print out the maximum memory used by the program as well as the maximum fragmentation measured. You'll also want to print out the number of allocations and the number of frees that occur when the program executes. To do this, you'll need to recompile your linked list as well as possibly modify your main program. Start first with the Gettysburg Address, and then move upwards to a medium file, such as Shakespear's The Tempest (http://shakespeare.mit.edu/tempest/full.html). If you really want a challenge, try to make War and Peace work properly.(Note: This will really be a challenge.☺)

5. Deliverables

1. Tar file including all of your source code, makefile, etc. Your source code must compile without any compiler warnings or errors using a makefile.
2. Lab report, submitted in pdf format, with the following
    (a) Name, date, and lab title
    (b) What things went right and what things went wrong with this lab.
    (c) What did you learn from this lab.
    (d) Conclusions

Deliverables are to be uploaded to the course website.

6. Preliminary Grading Rubric

	Weight Factor	Rubric Score	
Source Code			Code Quality - Code properly commented - Code indicates name, course, and date in comment block - Multiple C files used for application development - Header files protected against multiple inclusion - Static prefix used on file scope variables - Global variables defined in header files - Protection against NULL pointer dereferencing. - Other issues as noted.
			Code Functionality - Memory allocation occurs properly - Memory frees work properly - All tests pass properly - Other errors as noted.
			Run Quality - No Segmentation faults when program runs - Program executes efficiently and quickly
			Build Quality - Code compiles without warning or error when using makefile - Makefile works by issuing simple unix make command - No compiler warnings related to pointer definitions - No compiler warnings related to implicit function definitions - All code included in tar file - All source files and header files included
Report			Format - Name, Date, Lab Title, partners name - File submitted in pdf format - Source code in appendix
			Things gone right and things gone wrong clearly described - Complete sentences - All aspects covered
			What was learned thoroughly discussed - Complete Sentences
			Conclusions - Complete sentences

7. Method Descriptions

7.1. void memmgr free(void *ap);

The memmgr free method shall free an allocated block, returning it to the memory pool. The pointer, ap, is a void pointer, representing the start of the allocated address.

7.2. void* memmgr_malloc(int nbytes);

This method shall allocate a block of ram of the given size, returning a void* to the start of the block. If the block can not be allocated, NULL shall be returned.

7.3. void memmgr_init(void* pheapStart, void* pheapEnd);

This method will initialize dynamic memory, starting at the first location and going to the second location.

7.4. int memmgr_get_remaining_space();

This method will return the remaining free space which can be allocated. It is equal to the total space minus the allocated space minus any additional overhead. (Note: For even an empty memory block, it WILL NOT be equal to the size within the heap.)

7.5. int memmgr get number of fragments();

This method will count the number of memory fragments. This is the length of the free space linked list minus 1. By default, if all free space is in one block, there are no memory fragments. However, if the linked list is greater than 1, then there is some fragmentation within the memory.

7.6. int memmgr get allocated space();

This method will return the current allocated space in bytes.

7.7. int memmgr get maximum allocated space();

This method will return the maximum allocated space since initialization

7.8. uint32_t memmgr_get_allocationCount();

This method will return the total number of successfully allocated memory blocks/

7.9. uint32_t memmgr_get_freeCount();

This method will return the total number of freed memory blocks.