CMSC 257: Assignment 3
Dynamic Memory Management

Summary:

In the stack/heap model of memory management, the heap is used for dynamic memory. In C, we use the functions "**malloc**", "**calloc**", "**realloc**", and "**free**" to manage the heap. In this assignment, you will implement your own versions of these functions.

Deliverable:

A g-zipped tarball called "**asgn3_[your_eID].tgz**" (without quotes, replace your_eID with your VCU eID). In this tarball will be the files "**asgn3.c**", "**asgn3_support.c**", "**asgn3_support.h**", and "**Makefile**". The description of these files is as follows:

1) **asgn3.c** – The main file. This file will contain the main function which will test your code.
2) **asgn3_support.c** – The support file. This file will contain the support functions for the main operations.
3) **asgn3_support.h** – The header file. This file will allow you to link the support file with the main file.
4) **Makefile** – A script to compile the project.

Memory Management Tutorial:

For this assignment, we will be building on top of a memory management tutorial created by Dan Luu. (Link here: http://danluu.com/malloc-tutorial/)This tutorial contains several implementations of the major memory management functions in C. You must use this tutorial as the starting point for your implementations, as this assignment builds on top of this existing code. Begin by implementing this tutorial in your support file (asgn3_support.c). The tutorial is functional as is, any errors at this stage are caused by your implementation. Your code must use the set-up mentioned in this tutorial. Other online resources can be consulted but NOT copied. The tutorial mentions some other implementations for splitting/merging blocks that can only be consulted but not copied.

Improvements:

While Dan's implementation does work, it isn't very efficient. In this assignment, you must implement the following improvements:

1) Dan uses a singly linked list to track the memory segments. Change this to a doubly linked list. You must also improve his functions to properly track the new pointers.
2) The C standard requires that malloc returns "a pointer which is suitably aligned for any built-in type". Dan's implementation doesn't do this. In your implementation, ensure all allocated blocks are of a size that is a multiple of 8.
3) Dan's implementation inefficiently reuses freed blocks, using all of the block, even when only a part of it is needed. Your implementation should split blocks in this situation.
4) Dan's implementation does not account for free blocks which lie next to each other. In your implementation, every time a block is freed, it should merge with any adjacent free blocks. This will result in your list containing no adjacent free blocks.
5) Dan uses a first-fit algorithm for using a free block. Instead, you should use a best-fit algorithm

Main/Driver File:

The main file (asgn3.c) should contain a driver which tests your code. This driver is mostly for your benefit, so it is recommended that you go beyond the following requirements to help you understand how memory management works. The following are the minimum requirements for the main method:

1) Make at least 10 calls to the **malloc()** function.
2) Make at least 10 calls to the **calloc()** function.
3) Make at least 10 calls to the **realloc()** function.
4) Make at least 5 calls to the **free()** function.
5) Print the total memory leakage of your driver running your implementation. Memory leakage is defined as memory which is allocated, but not used.
6) Print the starting and ending addresses of the heap.

This main file should work for both your implementation of memory management and the C standard implementation of memory management (with the exception of the memory leakage segment). This can be tested by switching your "#include "asgn3_support.h" to "#include <stdlib.h>". This is a useful way to see if there are bugs in your implementation.

Support File:

The support file contains an implementation of the following functions:

1) **void *malloc(size_t size) –** Allocate a segment of memory in the heap, of at least the given size, and return a pointer to this segment.
2) **void *calloc(size_t nitems, size_t size) –** Allocate a segment of memory in the heap to store nitems elements of the given size. Set the entire segment of memory to 0. Return a pointer to this segment.
3) **void *realloc(void *ptr, size_t size) –** Resize the memory segment pointed to by ptr to the given size. Return the pointer to the newly sized segment.
4) **void free(void *ptr) –** Deallocate the memory segment pointed to by ptr.

It also contains several supporting functions, as described in Dan's tutorial. You may add additional support functions as needed.