* DUE DATE: Fri, March 14, 2019 by 11:59:59pm

**Enhancing A Dynamic Memory Manager**

In class we examined a simple dynamic memory manager based on the original design by Brian Kernighan and Dennis Ritchie. The design used a singly-linked circular list of free storage blocks with a header just before the payload area that contains a pointer to the next block and the total size of the block in multiples of the header size. The list is maintained in order of ascending address to facilitate coalescing adjacent free blocks to minimize exernal fragmentation.

The advantage of this design is its simplicity, but we also noted some problems with the design. In this assignment, you will modify this design in an attempt to address some of these problems and improve the performance. You will produce a new version that makes the improvements described below over the K&R version. This will enable you to compare these improvements to the original design. You may, if you wish, work on this assignment with a team mate. Both of you should be identified in the "README.md" file in your assignment repositories.

Rather than using the *sbrk()* function directly to implement *malloc()*, *realloc()* and *free()*, you will use surrogate function*mem\_sbrk()* found in memlib.c/.h (memlib). This function allocates a large pool using *malloc()* that *mem\_sbrk()* uses to allocate storage. This enables us to isolate our functions from the actual dynamic memory manager. It also enables us to reset the memory manager in order to run multiple tests without restarting the program

A test program "test\_heap.c" exercises our functions, determines that they are functionaing properly, and measures the performance of our functions. The test program takes one or more data (".rep") files that it uses to exercise the memory allocation functions. These data files contain commands to malloc, realloc, and free blocks of memory. For each file the "test\_heap.c" resets the heap back to its initial condition, runs the commands, and reports on the results. At the end, the program shows a sumary report for all the tests. The program has several flags including "-v" to show progress for each test, and "-d" to show a detailed log of each operation performed.

There are are three version versions for reference. "mm\_simple\_heap.c" is a simple allocator that never frees its storage. This version has the least overhead but the worst use of memory. "mm\_malloc\_heap.c" functions call the underlying *malloc()*, *realloc()*, and *free()* funcitons and do not use "mmlib". The third,"mm\_kr\_heap.c" is the one we studied based on K&R but adapted to use "mmlib"for its space. You will be creating a new version based on this third one.

**Doubly-Linked implicit list**

This is similar to the second design shown in the lecture 7 slides. Each block should now also have a duplicate header at the end of the block. This makes it possible to quickly check whether adjacent free blocks can be coalesced.

Modify the header to have just a size in header units and a bit that indicates that the block has been allocated if the bit is "1" and is free if the bit is "0". To avoid wasting extra space, you can "borrow" the lower bit of the size by ensuring that the total block size is always a multiple of two headers (at head and tail) so the size is always even. Alternately, you could replace the size\_t sized *size* field with one that is sizeof(size\_t)-1 bits and a 1-bit field for the allocated flag using C bit fields.

It would be possible to do without an actual free list by scanning all storage bocks for free space. You may want to try this first and benchmark the results. However for this assignment you should implement an explicit linked list of free nodes. To do this, use the space in the block payload area to store pointers to the next and previous node in the free list. To simplify the algorithms, you will want to maintain a "dummy" free list node as the original K&R version did. If you tried the implicity free list first, Benchmark this version against it so see what performance improvement, if any, you realize.

You should also experiment with whether the order of free nodes on the free list makes a difference. If using a "first fit" strategy works well enough, this would elimimnate the cost of finding the right place for a new node in the linked free list. You may also want to try a "best fit" strategy and benchmark the difference in performance.

Also in this version, add some safeguards to ensure that it is not possible to free storage that was already freed or storage that is not part of the heap.

Finally, consider how it might be possible to allow passing in a pointer to a location that is interior to the memory being freed. Although the C standard says that this is undefined, some memory allocator implementations of *free()* and *realloc()* do this. If you decide to implment this capability, it should not be too expensive. For example, it should avoid requiring a linear search of the heap for every call, something we have sought to avoid.

Create a new "mm\_dlink\_heap.c" for these changes. Do not modify the original "mm\_kr\_heap.c" file. You can exclude the other "mm\_\*\_heap.c" files in your project to test this version. How does the performance of this new version compare the the other three?