Name: _____

CS 0449 – Memory Management Lab

Background

To implement dynamic data structures, programs must be able to allocate storage locations on demand and free them when they are no longer needed. These storage locations are *anonymous* in that they are not associated with any program variable and can only be tracked through pointer variables. These storage locations are stored in an area of process memory called the *heap*.

The purpose of this lab is to implement a very simple heap that always allocates memory at the top of the heap by extending it. We will do this through sbrk() call that modify program break which points to the top of the heap. Obviously this is a very inefficient implementation of a heap since it never recycles any memory deallocated through free calls and always asks for more memory from the OS, but it is a working implementation nonetheless. Your job is to write the my_malloc and my_free functions for this simple heap that are counterparts to the malloc and free functions in the C library. Through this exercise you will deepen your understanding of pointer arithmetic and how to use it to update arbitrary locations in memory you have allocated, as well as learning the basics of heap management. At below is the manpage description of sbrk() for your reference:

```
void *sbrk(intptr_t increment);
```

DESCRIPTION

```
brk sets the end of the data segment to the value specified by  end_data_segment,
when that value is reasonable, the system does have  enough memory and the process
does not exceed its max data size (see setrlimit(2)).

sbrk increments the program's data space by increment bytes. sbrk isn't a system
call, it is just a C library wrapper. Calling sbrk with an increment of 0 can be used
to find the current location of the program break.
```

RETURN VALUE

```
On success, brk returns zero, and sbrk returns a pointer to the start of the new
area. On error, -1 is returned, and errno is set to ENOMEM.
```

Introduction

1. Login via SSH to `thoth.cs.pitt.edu`
2. Create a directory for this lab and change into that directory
3. Compile the skeleton code for your heap by following the below directions:

```
cp ~wahn/public/cs449/heap/*.c ./
gcc main.c heap.c –o ./heaptest
```

4. Now try running heaptest:

```
thoth $ ./heaptest
this is a test program
this * a * program
* * * * *
```

Success? No, this only ran successfully because the test code in main.c used the C library `malloc()` and `free()`. You need to change them over to your own implementations. You will do it in 4 steps.

What to do

5. **Step 1.** Modify the `MALLOC(n)` and `FREE(n)` macros at the top of main.c such that they respectively are translated to `my_malloc1(n)` and `my_free(n)`. In order for it to compile properly, you will also have to declare the prototypes of these functions at the top of main.c. If you run the new binary, you will get a segmentation fault because `my_malloc1` has not yet been implemented. Implement `my_malloc1` such that it allocates space by simply bumping up program break by the allocation size. Use the sbrk() call for this purpose. In this version of the heap, there are no data structures to mark the used and unused portions of the heap. We will add those later. After all is done, your heaptest should have the same output as the original `malloc()` and `free()`.
6. **Step 2.** Modify `MALLOC(n)` in main.c such that it is translated to `my_malloc2(n)`. Implement `my_malloc2` such that it is identical to `my_malloc1` except that now you reserve some space for the block header at the front of the allocated block. The structure of the block header is given in `struct Block`. Note that `my_malloc2` should return a pointer to the usable portion of the block which comes after the block header. The output of heaptest should still remain the same.
7. **Step 3.** Modify `MALLOC(n)` in main.c such that it is translated to `my_malloc3(n)`. Implement `my_malloc3` such that it is identical to `my_malloc2` except that now you fill in the block header. Once you are done filling in the block header, you should have a double-linked list linking together all the blocks in your heap. Now we are finally ready to modify the `DUMP_HEAP()` macro to dump_heap() in main.c. The dump_heap() function traverses the list to dump the contents of your heap. After these changes, your output should look like:

```
thoth $ ./heaptest
head->[1:0:29]->[1:29:27]->[1:56:26]->[1:82:29]->[1:111:32]->NULL
this is a test program
head->[1:0:29]->[1:29:27]->[1:56:26]->[1:82:29]->[1:111:32]->NULL
this * a * program
head->[1:0:29]->[1:29:27]->[1:56:26]->[1:82:29]->[1:111:32]->NULL
* * * * *
head->[1:0:29]->[1:29:27]->[1:56:26]->[1:82:29]->[1:111:32]->NULL
```

If you do not update the block headers properly, you may get an assertion fail while running heaptest that looks like the following:

```
thoth $ ./heaptest
heaptest: ./heap.c:110: dump_heap: Assertion `cur->next->prev == cur'
failed.
head->[1:0:29]->
Command terminated
```

Assertions are statements that you can put in your program to make sure that the program satisfies a certain condition at that point in the program. This particular assertion is telling you that the pointers in the double-linked list have not been updated properly. Assertions are a useful form of documentation that describes the behavior of a program that also gets checked automatically at execution time. Good programmers use assertions whenever applicable.

8. **Step 4.** Now we are finally ready to implement the final piece: `my_free`. Implement by using pointer arithmetic to calculate the location of the block header for the block getting freed and set the occupancy bit to 0. The final output of heaptest should look like the following:

```
thoth $ ./heaptest
head->[1:0:29]->[1:29:27]->[1:56:26]->[1:82:29]->[1:111:32]->NULL
this is a test program
head->[1:0:29]->[1:29:27]->[1:56:26]->[1:82:29]->[1:111:32]->NULL
this * a * program
head->[1:0:29]->[0:29:27]->[1:56:26]->[0:82:29]->[1:111:32]->NULL
* * * * *
head->[0:0:29]->[0:29:27]->[0:56:26]->[0:82:29]->[0:111:32]->NULL
```

What to Hand In

Submit your source files in a tarball:

```
tar zcvf USERNAME_lab_heap.tar.gz heap.c main.c
cp USERNAME_lab_heap.tar.gz ~wahn/submit/449/RECITATION_CLASS_NUMBER/
```