Please **read this entire assignment**, **every word**, before you start working on the code. There might be some things in here that make it easier to complete.

This lab is October 9th by midnight. Submit a single gzipped tar file to **TEACH**. Submitting your solution before October 9th will earn you a 10% bonus. If your submission is not a gzipped `tar` file, I will not grade your assignment. Your code must run on the EECS Linux server `os2`. If it does not run on `os2`, it will be graded as a zero.

This lab is worth 250 points.

Heap management – `beavalloc`

You know you've been wanting to do this, so finally, here is your opportunity. Write you own `malloc` package. You are going to call it `beavalloc()` and `beavfree()`. Those are the basic functions. You also need to implement `beavcalloc()` and `beavrealloc()`, but those are implemented using `beavalloc()` (with some extra code). **Your code must not make any use of any of the library `malloc()` functions.** You can use `brk()` and `sbrk()` for requesting or returning portions of the heap. In fact, you must use `brk()` and `sbrk()`. You will manage the memory that is allocated.

You may notice this line in the `man` page for `brk()/sbrk()`:

> Avoid using `brk()` and `sbrk()`: the `malloc(3)` memory allocation package is the portable and comfortable way of allocating memory.

Ha!!! As an accomplished C programmer and kernel hacker, you are empowered to use `brk()` and `sbrk()` to write your own comfortable heap manager.

There are a lot of various examples of `malloc()` source code out there. I encourage you to not rely on those and learn this on your own.

Your implementation of `beavalloc()` (and the other functions) must be done in a file called `beavalloc.c`. You must put your `main()` in a separate file. I'll provide you with a `main()` that calls a number tests for your `beavealloc()` functions. I have provided a file `beavalloc.h`, read the comments in that file.

You must have a `Makefile` that builds your project. I have placed a sample `Makefile`, the `beavalloc.h` file, and sample `main.c` in the following directory on `flip/os[12]`:

```
~chaneyr/Classes/cs444/Labs/Lab2
```

Do not make changes to the `beavalloc.h` file. Do what you will with the `Makefile`, just don't hurt my feelings. **Do not remove any of the flags for `gcc` (`CFLAGS`)**. I expect to build the `beavalloc` program by using the following two commands:

```
make clean
make all
```

Be sure to test your code. The `main.c` file has a few tests in it. Run all the tests and check to make sure the output is correct. Just because your code does not seg-fault, it does not mean the code correctly operates. The tests can either be run all at once or individually. I recommend you run them individually; it makes them easier to validate.

You can use whatever data structure to manage the heap you like. I decided to use a doubly-linked list. It worked very well for me. I would say that it is he most space or time efficient, but that is not the objective.

This is not an exercise to see if you can develop the fastest or most compact `malloc()` replacement. Many people have spent years working on that; you don't need to. This is a simple 1-week assignment; don't make it more than that.

Functions you need to write are:

```
void *beavalloc(size_t size);
```

The basic memory allocator. If you pass `NULL` or `0`, then `NULL` is returned. If, for some reason, the system cannot allocate the requested memory, set the `errno` global variable to `ENOMEM` and return `NULL`. You must use `sbrk()` or `brk()` in requesting more memory for your `beavalloc()` functions to manage. **When calling `sbrk()` for more memory, always request 1024 bytes.** It is best if you use a macro for this value, in case it needs to change. Making kernel calls is expensive, so we want to minimize them.

Notice that the parameter type is `size_t`, which is unsigned. If you happen to pass a negative value, odd things may occur.

When `beavalloc()` is called, only call `sbrk()` to request more memory form the kernel when there is not enough space in the currently allocated space to accommodate it. You must be able split existing blocks within the allocated space. If an existing block can be split to hold a new request, split it and use it. Correctly splitting blocks is probably the second most challenging portion of the project. You will be using a first-fit algorithm.

First-fit algorithm – when receiving a request for memory, scan along the list for the first block that is large enough to satisfy the request. If the chosen block is significantly larger than that requested, then it is split, and the remainder added to the list as another free block.

`void beavfree(void *ptr);`

A pointer returned from a previous call to `beavalloc()` must be passed. If a pointer is passed to a block than is already free, simply return. If `NULL` is passed, just return. Blocks must be coalesced, when possible, as they are free'ed. Correctly coalescing may be the most challenging portion of this project.

`void beavalloc_reset(void);`

Completely reset your heap back to zero bytes allocated. You are going to like being able to do this. Implementation can be done in as few as a single line of C code, though you will probably use more to clean up some variables and reset any stats you keep about heap. The only time you actually give memory back to the OS is when you make this call. A call to `beavfree()` will only mark the space available for reuse.

After you've called this function, everything you had in the heap is just **__GONE__**!!! You should be able to call `beavalloc()` after calling `beavalloc_reset()` to restart building the heap again.

`void beavalloc_set_verbose(uint8_t);`

I like to have the ability to enable and disable runtime diagnostic messages. A call of `beavalloc_set_verbose( TRUE );` will enable runtime diagnostic messages. A call of `beavalloc_set_verbose( FALSE );` will disable runtime diagnostic messages. **All runtime diagnostic messages must go to `stderr`, not `stdout`.**

`void *beavcalloc(size_t nmemb, size_t size);`

This function should perform exactly as the regular call to `calloc()` would perform. You can read the `man` page to see what it does. If `nmemb` or `size` is 0, then `beavcalloc()` returns `NULL`. Make use of your `beavalloc()`; don't reinvent anything for this. Consider the `memset()` function.

`void *beavrealloc(void *ptr, size_t size);`

This function should perform exactly as the regular call to `realloc()`. You can read the `man` page to see what it does. If `size` is `NULL`, return `NULL`. Consider the `memcpy()` or `memmove()` functions. When this function is called with NULL, it means the pointer has never been allocated. Since this we expect the memory block size to change (likely upward, this is `realloc()` after all), when the `ptr` is `NULL`, actually allocate 2 times the size passed as a parameter.

`void beavalloc_dump(uint leaks_only);`

This is the function you must use to display the contents of your heap. It is a large and messy function. I understand that you may need to make a few adjustments to it so that it will work with your data structures. However, make as few as possible and **DO NOT remove any of the columns of data shown** in the original implementation.

A longer description of the output from the `beavalloc_dump()` is shown in an addendum on page 5. This function can be found in the file `beavalloc_dump.c`. I recommend you copy it into your `beavalloc.c` module.

Look in the `beavalloc.h` file for declarations of the above functions.

Below is a raw plan on how you can proceed. This plan is ***basic*** and not required, but it is pretty much how I worked through the project.

1. Start your `Makefile`. Initiate revision control.
2. When `beavalloc()` is called, just call `sbrk()` for the new space. Place that new space into your data structures to manage the space. Don't worry about the other calls for now. Check your code into revision control.
3. Make sure `beavaalloc_reset()` works and deallocates all space previously allocated using `sbrk()`. This will probably be a lot easier than you think. Consider doing a little `brk()` dancing for this. A `static` variable within your `beavalloc.c` file could make this really easy. Check your code into revision control.
4. When `beavefree()` is called, set the block as free/unused in your data structures. Check your code into revision control.
5. Split blocks that are large enough when memory is requested. Check your code into revision control.
6. Coalesce blocks when adjacent blocks are free-ed. Check your code into revision control.
7. Implement `beavcalloc()`. This is pretty simple. Check your code into revision control.
8. Implement `beavrealloc()`. Slightly more involved that `beavcalloc()`. Check your code into revision control.
9. Make sure all the tests work. Just because a test does not seg-fault does not mean the test ran correctly. Break out a calculator and make sure things add up. Most computer-based calculators have a "programmer" mode that will allow you to calculate the hex addresses. I know both Mac and Windows have this feature in the calculator. Check your code into revision control.
10. Celebrate. This was my favorite step.

I recommend you make use of the `assert()` statement in your code. You can see some examples of it in the testing code I provide and by looking at the man page. Remember, testing is good.

You must use the following `gcc` command line options (with `gcc` as your compiler) in your `Makefile` when compiling your code.

```
-g
-Wall
```

```
-Wshadow
-Wunreachable-code
-Wredundant-decls
-Wmissing-declarations
-Wold-style-definition
-Wmissing-prototypes
-Wdeclaration-after-statement
```

Be sure your code does not generate any errors or warnings when compiled. Hunt down and fix all warnings in your code. **You may not use any `` `std=…` `` command** line options for `gcc`. Just use the default standard, i.e. `c90`.

For a comparison, my `beavalloc.c` file contains less than 500 lines of code. I have some comments and some extra things in there taking up space. You don't have to (completely) write the `beavalloc_dump()` function. All in all, it is not a huge chunk of code.

Final note

The labs in this course are intended to give you basic skills. In later labs, we will ***assume*** that you have mastered the skills introduced in earlier labs. **If you don't understand, ask questions.**

Addendum – Format of the `beavalloc_dump()` output.

Below is an example of running my code on test 5 from the suite of tests. The heap map consists of several columns of data. The `beavalloc_dump()` function has 2 purposes: 1) help grade the assignment, 2) help debug the assignment. The columns in the output are:

blk no: I start numbering the blocks managed by `beacalloc()` from zero, so the initial block number is block 0. For me, this is the start of the doubly-linked list I use to manage the heap. It is also the address of the beginning of the heap.

block add: this is the beginning address of a block `beavalloc()` is managing in the heap. For me, it is the beginning of an instance of the structure in my doubly-linked list. This address is shown as hex.

next add: the doubly-linked list I use contains 2 pointers for the previous and next elements in the list. This is the address of the next element. If the next address is (nil), it means that it is the head of the list. This address is shown as hex.

prev add: the doubly-linked list I use contains 2 pointers for the previous and next elements in the list. This is the address of the previous element. If the previous address is (nil), it means that it is the end of the list. This address is shown as hex.

data add: in my list element data structure, I keep a pointer to where the data for the block begins. This is that pointer; it is a little wasteful to keep this around as a pointer. This address is shown as hex.

blk off: this is the offset (in decimal) where the block begins from the beginning of the heap.

dat off: this is the offset (in decimal) where the data begins from the beginning of the heap.

capacity: this represents the total number of data bytes available in the block (in decimal). Some or even most of the bytes may not be used. See the size column below.

size: this represents the number of data bytes (in decimal) requested by the user's call to `beavalloc()`. This is the number of bytes the user can use in the block. If the user goes beyond size bytes, then they have actually gone out of bounds of their allocation. The block may have a capacity larger than the size. The additional bytes could be used in a call to `beavrealloc()` or the block could be split (if large enough).

blk size: the total number of bytes used by the block (in decimal). This will be the sum of the capacity and the size of the data structure used to manage the heap.

status: this simply represents whether a block is currently in use (allocated to the user) or is free (awaiting reuse).

The row of data below the per block information shows (in decimal) the total number of bytes of data capacity that are currently allocated, the total number of bytes the user currently has in use, and the total number of bytes are allocated in the heap.

The final line in the output from `beavalloc_dump()` shows: how many blocks are allocated to the user, how many block are marked as free, what is the address of the beginning of the heap is (Min heap), and what the address of the end of the heap is (Max heap).

```
% ./beavalloc -t 5
beavalloc tests starting
  running only test 5
base: 0x12b3000
*** Begin 5
      5 allocs 3 frees
  ptr1 : 0x12b3030
  ptr2 : 0x12b3430
  ptr3 : 0x12b3830
  ptr4 : 0x12b3c30
  ptr5 : 0x12b4030
heap map
  blk no     block add   next add    prev add    data add    blk off     dat off      capacity        size            blk size        status
  0          0x12b3000   0x12b3400       (nil)   0x12b3030   0           48           976             510             1024            in use
  1          0x12b3400   0x12b3800   0x12b3000   0x12b3430   1024        1072         976             530             1024            in use
  2          0x12b3800   0x12b3c00   0x12b3400   0x12b3830   2048        2096         976             550             1024            in use
  3          0x12b3c00   0x12b4000   0x12b3800   0x12b3c30   3072        3120         976             570             1024            in use
  4          0x12b4000       (nil)   0x12b3c00   0x12b4030   4096        4144         976             590             1024            in use
  Total bytes used                                                                    4880            2750            5120
  Used blocks: 5  Free blocks: 0  Min heap: 0x12b3000    Max heap: 0x12b4400
heap map
  blk no     block add   next add    prev add    data add    blk off     dat off      capacity        size            blk size        status
  0          0x12b3000   0x12b3400       (nil)   0x12b3030   0           48           976             510             1024            free    *
  1          0x12b3400   0x12b3800   0x12b3000   0x12b3430   1024        1072         976             530             1024            in use
  2          0x12b3800   0x12b3c00   0x12b3400   0x12b3830   2048        2096         976             550             1024            free    *
  3          0x12b3c00   0x12b4000   0x12b3800   0x12b3c30   3072        3120         976             570             1024            in use
  4          0x12b4000       (nil)   0x12b3c00   0x12b4030   4096        4144         976             590             1024            free    *
  Total bytes used                                                                    4880            2750            5120
  Used blocks: 2  Free blocks: 3  Min heap: 0x12b3000    Max heap: 0x12b4400
*** End 5


WoooooooHooooooo!!! You survived test 5.

      Make sure it is correct.
```