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 is an individual assignment.

This lab is February 5th by midnight. Submit a single gzipped tar file to **Canvas**. If you don't know how to create a gzipped `tar` file, you need to learn before you submit this assignment. Consider reviewing the `basic-xv6-setup` video and use of the '`make teach`' command for your `Makefile`. If your submission is not a gzipped `tar` file, I will not grade your assignment. Do not just `tar` up all the junk in your development directory and submit that into TEACH. **It will be a 50% deduction if I see a bunch of junk like `.o` files, executable files, revision control files, or editor droppings in the tar file you submit. HINT: You could also look at the end of this document.**

I implore you to do you development on the os2 server. Trying this in Visual Studio or on a Mac is likely to lead to hair loss and frustration. Development on a different Linux system than os2 could cause your code to fail to correctly compile on os2. I will do all testing on os2. If your code does not work correctly on os2, then I consider it not working, regardless of how it may have worked on some other system.

This lab is worth 200 points.

Part All – `beavalloc`

You know you've been itching to do this, so finally, here is your opportunity. Write you own `malloc()` replacement. You are going to write functions called `beavalloc()`, `beavfree()`, `beavcalloc()`, `beavrealloc()`, `and beavstrdup()`. Your code must not make any use of any of the `malloc()` functions. You can use `brk()` and `sbrk()` for requesting or returning portions of the heap. 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 the `brk()` and `sbrk()` system functions 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 have provided files `beavalloc.h`, `beavalloc.c`, `main.c`, and `Makefile`. Read the comments in those files.

Anytime you call `sbrk()` to request more memory from the kernel for your code, you must call with a multiple of 1024 bytes. Use the smallest multiple of 1024 that will hold the number of bytes requested by the user (plus any overhead from heap management data structure). Avoid making extra calls to `brk()` and `sbrk()`. Each call to either of those functions requires your process to switch into kernel mode and then back to user mode, which slows it down. My `beavalloc` code has exactly one call to `brk()` and exactly 1 call to `sbrk()`. The testing code is exempt from this (testing is expected to be slow).

You must have a `Makefile` that builds your project. I have placed a sample `Makefile`, the `beavalloc.h` file, `beavalloc.c`, and `main.c` in the following directory on `os2`:

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

Do not make changes to the `beavalloc.h` file or the `main.c` file. Do what you will with the `Makefile`, just don't hurt my feelings. It should work on `os2` for you without modification.

Be sure to test your code. The `main.c` file has a few tests in it and it will evolve as I generate more tests. The file `main.c` may be updated with more tests when I grade your assignment. You can check back with it. Just because a test does not fail (seg-fault), it is not a guarantee that the result is correct. You need to actually look at the output your code produces from `beavalloc_dump()`.

Functions you need to write are:

```
void *beavalloc(size_t size);
void *beavcalloc(size_t, size_t);
void *beavrealloc(void *, size_t);
void beavfree(void *ptr);
void beavstrdup(const char *s);
void beavalloc_reset(void);
```

Look in the `beavalloc.h` file, in the `Lab2` directory on `os2`, for descriptions of the above functions. The file `beavalloc.c` gives you the empty functions (stubs) to begin writing the code.

You'll notice that the `beavalloc.c` gives you a couple extra functions: `init_streams()`, `beavalloc_set_verbose()`, `beavalloc_set_log()`, and `beavalloc_dump()`. The `init_streams()`, and `beavalloc_set_log()` work together so you can specify an output stream for all diagnostic out from your code. **All diagnostic output must be written to the stream `beavalloc_log_stream`.** This defaults to `stderr` but can be changed with a call to `beavalloc_set_log()`, as is done in the `getopt()` code in `main.c`. The output from the `beavalloc_dump()` function is the primary way that I will use to check the correctness of your code. You should not make changes to it. There are a couple module level variables and macros at the top of the `beavalloc.c` file to guide your start.

You must use the data structure `mem_block_t` exactly as shown in the `beavlloc.c` file. Do not make any changes to the structure. Do not move it into the `beavalloc.h` file.

You won't be able to make any calls to the regular `malloc()` functions in your code. if you do, you will see seg-faults galore.

You must use the following `gcc` command line options (with `gcc` as your compiler) in your `Makefile` when compiling your code (which are already in the provided `Makefile`, so this should be easy):

```
    -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`. Warnings from compilation of your code will be a 20% deduction.

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. There seems to be a repeating theme in these steps. I can't quite place my finger on it.

1. Copy the `beavalloc.c`, `beavalloc.h`, `main.c`, and `Makefile` form the `Lab2` directory on os2. 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 than `beavcalloc()`. Check your code into revision control.
9. Implement `beavstrdup()`. This is really simple. Check your code into revision control.
10. Make sure all the tests work. Just because a test does not seg-fault does not mean the test produced the correct output. 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.
11. Check your code into revision control. Submit your code into TEACH.
12. Celebrate. This was my favorite step. Go ahead and check your code into revision control.

This is a challenging lab. Don't think that the duration you have to complete it means that you should wait until the day before the due date to begin. If you wait that long, you probably won't get to step 12.

There are also notesets and videos related to this project on Canvas. If you are struggling with how to split a block or how to coalesce blocks, those notes/videos should be the first place you look.

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.**

Supplement: How to Create a gzipped tar File

If you do not submit `gzip` compressed tar file, it is very likely that your assignments will not be graded (therefore give you a zero for all your hard work). So, you should make sure you know how to:

1. Create a `gzip` compressed tar file.
2. Extract files from a `gzip` compressed `tar` file.
3. List the contents of a `gzip` compressed `tar` file.

Read the `man` page on `tar` and make sure you understand the following command line options for tar: `x`, `c`, `t`, `f`, `v`, and `z`. Ponder the deeper meaning of the following lines:

- `tar cvfz Lab2.tar.gz *.[ch] ?akefile`
    - Creates a gzipped tar file. The `c` in the list of options means **Create**. The file will contain all the `*.c` and `*.h` files. It will also contain the `makefile` or `Makefile`.
- `tar xvfz Lab2.tar.gz`
    - Extracts the files from a gzipped tar file. The `x` in list of options means **eXtract**. All the files will be extracted. The gzipped file is unaffected by the extraction. Any files with the same name as extracted files will be replaced.
- `tar tvfz Lab2.tar.gz`
    - Will list a table of contents for the gzipped tar file. The `t` in the list of of options means **Table of contents**. The name of each file contained in the gzipped tar file will listed.