# Assignment 3

## Setup

This is an individual assignment. You may discuss questions and potential solutions with your classmates, but you may not look at their code or their solutions. If in doubt, ask the instructor.

The assignment is only tested on the CSUG machines, as are the scripts for packaging. Any issues that arise from trying to use other machines will be ignored or dealt with on a low priority.

If you want to take a slip day, as permitted by your syllabus, your email request must be sent to the instructor before the deadline. The instructor will note it, but will not acknowledge it immediately.

### Goal

There are two parts to this assignment. The first part requires you to build a doubly-linked list.

The second part requires you to build a pool-based memory allocator that uses the doubly-linked list to track allocations and free objects.

### Setup and Smoke Test

After you have downloaded and unpacked the assignment, you should run the following commands and see the following output.

```
$ cd a3/dbll
$ make
cc -std=c99 -Wall -g -I . -I ../th -O dbll_test.c dbll.c ../th/test_helper.c
-o dbll_test
$ ./dbll_test
FAIL: dbll_create return value ((nil)) must be non-NULL
=== DONE
```

To check the memory allocator, do the following (assuming you're in `a3/dbll`):

```
$ cd ../poolalloc
$ make
cc -std=c99 -Wall -g -I ../dbll -I . -I ../th -O pa_test.c poolalloc.c
../dbll/dbll.c ../th/test_helper.c -o pa_test
```

```
$ ./pa_test
FAIL: mpool_create returned non-null ((nil))
```
If you see the above, you're all set to begin working.

## Doubly-Linked List

To work on the doubly-linked list, edit the file `dbll.c`, and fill out the functions.

Look at the `main` function in `dbll_test.c` to figure out the order in which you should do the assignment. Since we're dealing with memory accesses and pointers, the tests will stop immediately as soon as they detect an error.

Your workflow should look like this:

1. Edit `dbll.c`
2. Run `make` to compile `dbll_test`
3. Fix compiler errors if any, goto 2.
4. Run `dbll_test`
5. Fix any run-time errors using `gdb`, `valgrind`, etc. (See the notes section below)
6. Goto 1, fix any failing tests.

If you pass all these preliminary tests, you should get a message `ALL DONE` when you run `dbll_test`. You should submit to Gradescope at this time.

## Memory Allocator

To work on pool-based allocator, edit the file `poolalloc.c`, and fill out the functions.

Look at the `main` function `pa_test.c` to guide your efforts. Again, the tests will stop as soon as they detect an error.

The goal of a pool-based allocator is to allocate a large chunk of memory at the beginning (this is the "pool"), which is done by `mpool_create`. Then all future allocations obtained using `mpool_alloc` are carved out of this pool.

Essentially, all memory allocators work by keeping track of free blocks of memory and allocated blocks of memory. In the beginning, your allocator will begin with one big block of memory in the `free_list` whose size is equivalent to the pool size. As allocations are made, this big block of memory will be broken up into smaller chunks that are tracked using `alloc_list`. Blocks that are `mpool_free`d are moved back into the free list.

Read the comments in the `poolalloc.c` to understand what each function must do. The comments outside each function describe the behaviour of the function, and your implementation must follow that. The comments inside each function body describe a possible way to implement a function. You can ignore these if you find them unhelpful.

You can also take a look at `pa_test.c` to see how the pool allocator is supposed to behave. I've also written up a document that graphically describes the state of the pool allocator for various actions.

Reading Section 9.9 of the textbook is highly recommended, but the pool-based allocator you're implementing in this assignment is simpler than the one described there.

## Helpful notes

### Segmentation faults

It is almost certain you will encounter segmentation faults when doing this assignment.

First, try to find out where in your source code segmentation faults are occuring. For this, use a debugger.

For example, run your program inside `gdb`:

```
$ gdb dbll_test
(gdb) run
...
```

If `gdb` encounters a segmentation fault, it will show you where the segmentation fault occurred.

Recall that segmentation faults occur when you deference pointers incorrectly. Try to identify which pointer is causing the segmentation fault, and look at its value to understand why (use the `p` command to print out the value of the variable).

You may find the GDB tutorials here helpful.

### State Dump

You may find it useful to dump out the state of your memory allocator.

To do this, write a function that simply prints out all the items in `alloc_list` and in `free_list`.