Lab 3.1

Out: Thursday, February 17th

What you'll learn.

In this lab you'll write your very own memory allocator. You will also get to practice pointers, casting, and memory manipulation.

How you'll do it.

You will write a library called "myalloc" which allows a user to allocate and free blocks of memory. `myalloc` starts by requesting a large block of memory from the heap. You will manage this memory for the user by keeping track of blocks of memory and managing a "free list" of these blocks.

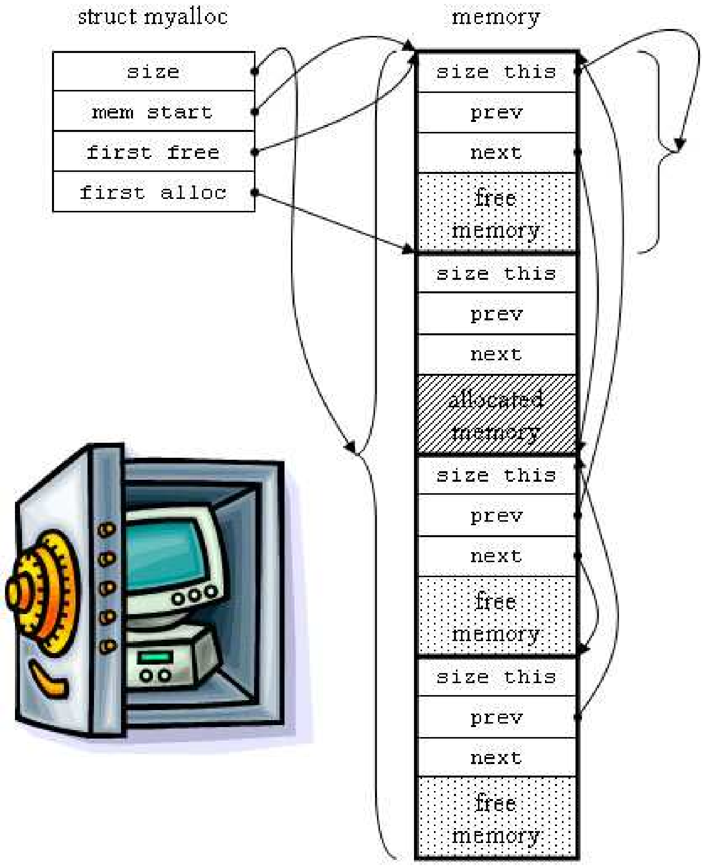
Memory Allocation Systems

In C, the commands `malloc` and `free` are used for memory management of the "heap." From the programmer's perspective, the heap is just a pool of memory that can be called upon at any time to provide a specified amount of memory. Internally, records must be kept to determine what memory has already been allocated, and how to find memory that is available.

Real implementations of malloc also include optimizations and safety-checks to make memory allocation as fast and robust as possible. However, you will be implementing a simple malloc and free that perform only the essential functionality. To simplify things further at first, you will divide the memory into uniform sized blocks. This means you won't have to worry about breaking and merging blocks until Lab 3.2.

If memory allocation still sounds complicated, don't worry. We'll be giving you detailed stencil code with plenty of comments to help you along.

Memory Layout



The above diagram shows how we'll be keeping track of our pool of memory. The box on the left is the `myalloc` struct. It gives us the information we need in order to navigate through the available memory. The `size` field tells us the size of the entire pool of memory. The `mem_start` field tells us where the beginning of the pool is.

The boxes on the right are called "blocks," and they are aligned in consecutive memory. Each block is either allocated or free depending on whether it has already been assigned to the user. If a block is free, it is placed on the "free list." The beginning of the free list is the block referenced by the `first_free` field in the myalloc struct. Similarly, the first allocated block is referenced by the `first_alloc` field. **A block should never be on both lists at once.**

Each block stores its own size in the field called `size_this`. You can use this information to navigate between adjacent blocks in memory. Blocks also have a field for their `prev` and `next` neighbors on whatever list it happens to be on. If a block is the first on a list, its `prev` field is `NULL`. If a block is the last on a list, its `next` field is `NULL`. Note that a block's neighbor on a list may or may not also be adjacent to it in memory.

1 Task 1: Uniform Block Size Initialization

When the user calls `myalloc_init` you will need to create a new `myalloc` struct and a new pool of memory to assign blocks from. To make things simpler at first, we'll be using uniform sized blocks. This means you'll need to format the pool of memory upon initialization. Your first task is to write the initialization functions. When you are finished you should be able to print out the free list containing uniform sized free blocks.

Hint: We *strongly* recommend that you use `assert` liberally when dealing with linked list manipulation. Bugs in your linked list insertions or removals are very hard to track down unless you can catch them immediately with an `assert` statement.

> **Task:** Create a new directory for this lab, and copy the stencil code from `/course/cs034/pub/lab3/`. Fill in `myalloc_init`, `myalloc_cleanup`, and `format_uniform` in `myalloc.c`. In `main.c`, write a main function that calls `myalloc_init`, `myalloc_print`, and `myalloc_cleanup`. We have already written a `Makefile` for you to use in this lab.

2 Task 2: myalloc and myfree

Once your memory is initialized, you're ready to begin allocating and freeing blocks of memory. With uniform block sizes, this process is really just an exercise in linked list manipulation. When allocating memory, simply take the first block off the free list and add it to the beginning of the allocated list. When freeing a block, just remove it from the allocated list and insert it into the correct location in the free list. The free list should always be ordered by increasing memory address of the blocks. This will be important later when you try to merge blocks together.

> **Task:** Fill in the methods for myalloc and myfree. You will also need to fill in the methods for `allocate_block` and `free_block`. Test your code by calling `myalloc_test` in your `main()` function. You should make sure the test passes both with a small pool (e.g. 4096) and a big pool (e.g. 50000). If you run into problems, use `gdb` to debug your program.