Section 5
10/26

Thoughts from Lab 1 Grading

- Q5: vectors.S
- Make sure you put the *NetID's* at the top of your txt file

Overview of Lab:
At the end of last lab, we were restrictive on the user's ability to consume resources (Only allocating one page for the user stack, no ability to allocate on the heap, …). In this lab, we will be giving the process more freedom to use memory (within limits of course).

User Level Heap:
In the implementation of `malloc` and `free` that we all (hopefully) did in 351, the underlying system made calls to `sbrk` when there was no user memory readily available, or the user asked for a larger portion of heap space. Internally, the kernel will update the `mem_region` to reflect the new memory and add the page to the page table (see `kernel/vm.c:mappage`). There is already a version of `malloc` and `free` in userspace given to you (`user/umalloc.c`). So one you have completed this phase your user heap should work.

Shell:
The shell portion is laid out fairly clearly in the spec. What's happening here is we are adding an `init (user/init.c)` process to our kernel that will fork off a child and execute the shell, which will be the way programs will be spawned off from here on out. If we want to add new programs, we can copy the binaries directly onto the hard drive (`mkfs.c`), add an inode entry in the file system, and the shell will be able to find it. As long as it's a valid binary, our kernel will be able to fork off an instance of it.

....
MMFile (2GB + File Size)
....
MMFile 2GB
STACK (4KB) at 2GB-4KB
Guard Page (4KB) at 2GB-8KB
...
...
HEAP (Page aligned)
CODE

Memory Mapped IO:

Every process will have the ability to open a file in a memory mapped region which it can read and write from, instead of needing to use the read/write interface. Usually, writes to this will be reflected on disk, but since we have a read only file system they will be discarded on close. In Lab 5 when we add writing to disk, we will revisit this property.

Because writes will be reflected on disk in the future, the most current version of the file will be in memory, so if two processes simultaneously `mmap` a file, both should point to the same physical memory.

Q: What does this mean when one process unmaps the file, but the other still keeps it. How do you know when there are no more consumers of that memory in `munmap`?