This was a hard lab, I started pretty late. I got the first 40% done.

Here's what I did.

First, I tried finding what files to edit, I saw PHYS_BASE was defined in vaddr.h, but that didn't help me.

I started hunting around in files in the userprog folder.

After some hunting combined with looking at the secret sauce, I found that push_command() is whagt I was looking for, and I found it, in process.c.

I then began to approach manipulating and handling the stack via *esp. After more referring to secret sauce, I managed to figure out how the stack worked. It took a lot of trial and error but I eventually learned that the stack populates addresses below PHYS_BASE and that it keeps growing downward the more you push on.

I also learned that to push stuff onto the stack, you need to decrement esp first to account for the size of whatever I'm pushing on as whatever I push on populates addresses in increasing order.

Because of this, I thought that I had to reverse my strings of args to push them on backwards(this is wrong, and I got rid of this change later on). So I went about detecting my args and breaking them apart.

I first went through cmdline to detect the number of args. I did this by looking for spaces or the end of the cmdline. This I stored in argc.

For storing args, I had an argumentArray which I pushed on char* arrays into the elements. Each char* array was extracted from cmdline by searching through cmdline for either the end of cmdline or a space. Through tricky indexing, I was able to extract the args and insert a null terminator at the end of each string.

Once I had this, I had to word align. I saw in vaddr.h that there was a way to page_round_down, but I didn't know quite how to use it yet, so I implemented my own solution, which was to count the total number of bytes our args+null terminators contained, modulised with %, and took the remainder and made a char array of that size and filled each element with '0'. I then pushed that onto the stack (it took me a bit to realize memcpy allowed us to push char arrays onto the stack). I then made another char* array to store 0. this is the null sentinel. I pushed that onto the stack.

As I pushed my args onto the stack, in right to left order, I saved the *esp for each arg pushed on. After pushing null sentinel on, I pushed each address holding an arg onto the stack, in right to left order.

I then pushed on the address holding the address of argv[0], which should have been the address of the last thing pushed onto the stack.

Then I pushed on argc's value onto the stack.

Once all that was said and done, I pushed a fake address onto the stack, I just pushed 0.

After all that, I thought I was done, but the test failed. So after a bit of hunting, I found that I needed to change process_execute(). After creating a semaphore, I pushed the address of the semaphore into thread_create(). I then modified the thread struct to hold onto the semaphore passed in, this also required modifying thread_start() by creating and pushing the address of a junkSemaphore into the thread_create().

After passing the semaphore into thread_create in the function process_execute(), I then semaphore_down'd the semaphore.

In order for the parent to keep running, I needed to semaphore_up in the child somewhere, and I did that, in thread_exit.

After all of that, it worked, for the first 40%.

CONTINUATION

I got up to 60%

This meant I passed all the arg tests.

I looked to see why if we passed any args to cmdline, our file wouldn't open.

This is because our filename is the cmdline, or at least, that's what the system used to think.

I extracted the first arg, and set this to the thread's name (on thread_create call in execute process). I also extracted it in load and used it to open our file.

Other than minor formatting of push_command(to account for double spaces and to not let references to unit'd char* arrays to happen, we're pretty much set.