Ryan Hamamoto
rkhamamo
5/30/18
CS111
Professor Harrison

Lab 3 – User Processes & System Calls

Process.c:

In push command, to first pass the args-none test, I hard coded the argv pointers, arg count, and copied the entire command line into *esp – 10 (because I knew the exact location on the stack in which argv[] will go) to make sure I knew how to pass the tests. I also implemented a semaphore that is initialized in process_execute, goes down at the end of process_execute, and goes back up in start_process, so that only one thread at a time executes.

After args-none was passed, I then implemented tokenization to break up the command line and allow for multiple arguments. I tokenized it once, and used it to get argc, correlating with the amount of tokens that are created. I then used argc to allocate just the right amount of space that argv would need, and tokenized the command line again, this time copying the tokens into the stack after decrementing the total string length of the token + 1 (for the padded zero). I placed each memory location address into argv. I then put a zero into the stack, then placed each element of argv into the stack (pointers to each location of each token), then placed the address of the start of argv into the stack, then argc, then the false return address. A major error I ran into was that during the second tokenization, I did not use a separate copy of the command line and began tokenizing tokens. I had to print my tokens and print my esp pointers to figure out where I was going wrong. I got a lot of page faults for this, and often times my argv print loop would only print the first element. I then just assigned separate char * pointers for the saved pointer location and token to ensure that no overlapping occurred. All other errors that I encountered were pointer arithmetic errors.

In order to execute even one argument I had to make sure the correct file name was being passed to process_execute. Therefore I had to take the first token of the command line string (which is always a file name) and passed this argument to thread create. I had to do the same thing in start_process so that the load function has the function passed, not the entire command line.

Syscall.c:

Then, just for the sake of passing the test, I modified syscall.c so that each case is addressed in the switch statement with nothing actually happening in each switch case. This allowed me to pass create-normal, open-normal, open-twice, and close-normal.

To pass write-zero, I added the case in which sys_write would not write to stdout. In this instance, we must edit a file in the current thread's open file list, which I added to thread.h. Before accessing any files, the system makes sure to acquire a lock, then release it. To pass read-zero, I copied the write_handler and sys_write code for read_handler and sys_read so that it

checks to see if it will be reading from stdin, and if not, acquires a lock, reads the designated file in the current thread's open files, and then releases the lock. It's the same function just for reading.

In order to pass all of the open file tests, I created open_handler in which it takes the file name from the command line, opens it, and places it into the current thread's list of open files where there is an open slot. The eax element of f is set to the index the open file occupies in the current thread's open file list. In order to pass open-missing, this element needs to be set to -1 if the function is passed a null file.