**CS 4500 Project 1 – System Calls and Processes**

1-1. **Project Members**: Daniel Frey, Justin Hale

**We have neither given nor received unauthorized assistance on this work.**

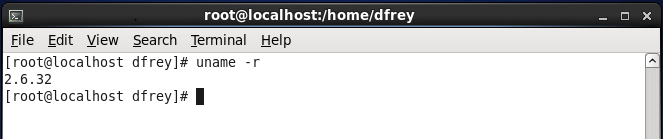
1-2. **VM**: jhale-virtual-machine

**instructor Account Password**: zhuang

1-3.

**Task 0**: This task was rather straight forward. The main issue with this part was getting the VM to boot into this version, however, the note on the assignment sheet initially helped. To not have to remember to select this every time, the default option was changed to make the VM boot into 2.6.32. Instead of typing out the instructions each time to compile the kernel, we made a script that executed these instructions. Furthermore, we made an alias so that we could simply type ‘recompile’ to compile the kernel.

Below is a screenshot to show the new version of the kernel was successful.



**Task 1**: This task, too, was straight forward. It was mainly following the assignment sheet to get this set up. This task allowed us to get familiar with creating and implementing system calls for the following tasks. No real issues were involved with this task other than accidentally mistyping something.

\*To test this system call, use *./test\_syscall* in the *instructor* directory.

**Task 2**: To solve this problem we decided that we must use *current* as indicated in the assignment sheet. After learning how to use current, printing out the name and pid were simply done by accessing those elements of the struct. For printing out the same information for parent processes until init, we created a for loop that iterated over each parent process to display the necessary information. We learned how simple it was to loop over processes.

**\***To test this system call, use *./test\_print\_self* in the *instructor* directory.

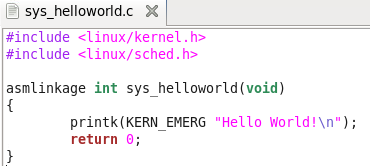
**Task 3**: To solve this problem we looked up the Linux functions on how to get the task\_struct of a specific pid. To do this, we used the functions find\_task() and find\_vpid(). The latter finds the process, and the former gets the task\_struct for that process. Together with these we were able to output the necessary information. To make sure that we had no issues with invalid pids, we checked that the task\_struct was not null for a given process. From this task we learned how to efficiently get the task\_struct of a running process.

The test program for Task 3 required more than the previous ones. The test program uses the command *pgrep bash* to get the arbitrary pid. The output from that command is obtained, converted to an integer, and then passed to the system call.

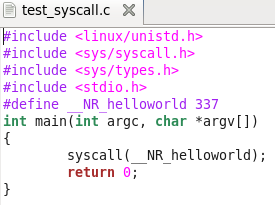
\*To use this system call, use *./test\_print\_other* in the *instructor* directory.

2-1. **Source code for Task 1**

Kernel:

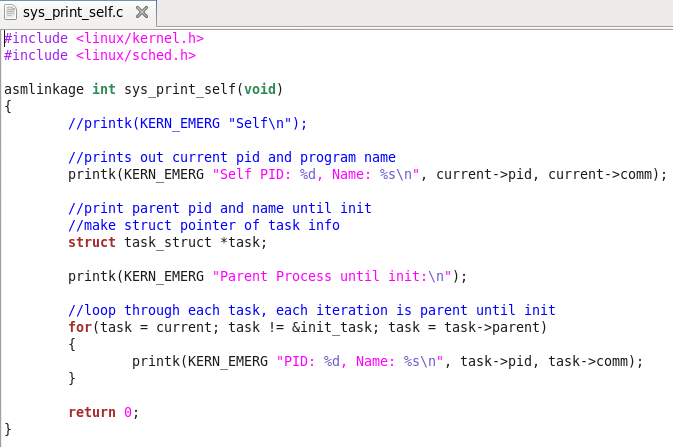


Test:

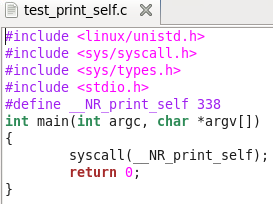


2-2. **Source code for Task 2**

Kernel:

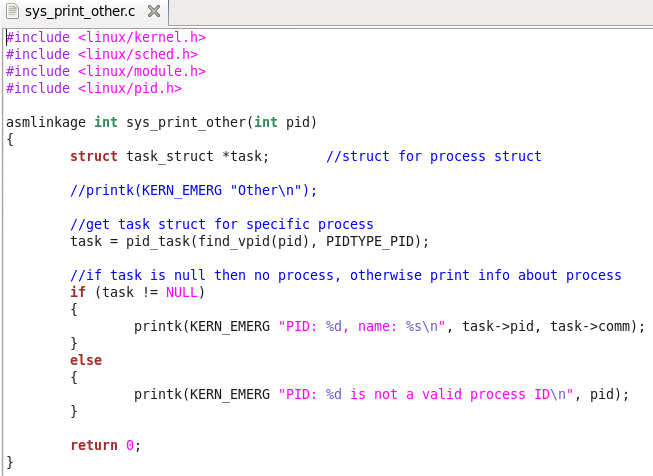


Test:

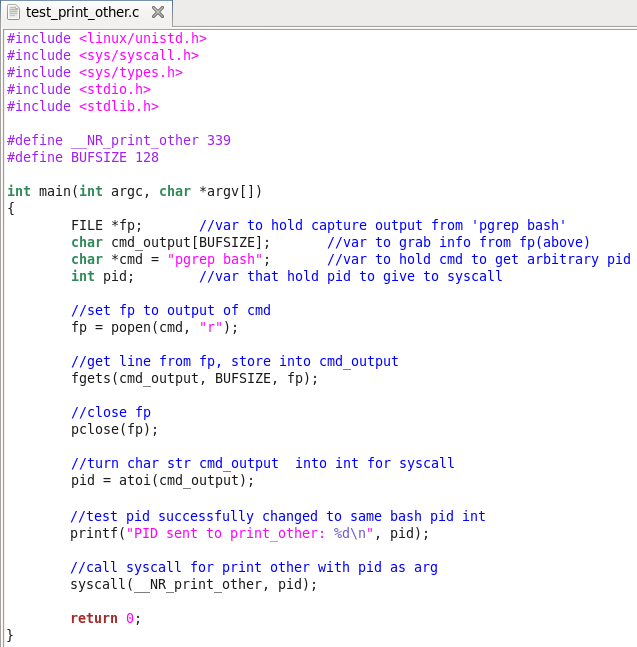


2-3. **Source code for Task 3**

Kernel:



Test:



**\*Note: source codes attached as files as well.**