ELEC 377 Operating Systems F22

Lab 3

Due November 1st, 2022

By: Kevin Yu, Raatik Sharma

Student Numbers: 20203451, 20120770

Problem Statement

The objective for this lab is to use Linux system calls to retrieve system information and start processes by writing lower-level C code.

Step 1

In step 1, we coded the “*skipChar” function, the purpose of this function is to take a pointer to a character array and locate the next non-empty space in the string. The function did this by iterating through each character of the string and checking for a non-empty space character and then returned a pointer to that character. If no skip character is passed as an argument to the function, then the pointer to the character array is just returned immediately.

Step 2

The “*skipChar” function is used to find the index of the next non-space character in the character array and returns a pointer to that empty space.

The “strChr” function is used to find the index of the next space-character in the character array and returns a pointer to that empty space.

Therefore, in this step, we split the user’s input and store the pointer to the beginning of each word in the “args” array. This is done in the splitCommandLine function by having a pointer go through the user’s typed-out command stored in the commandBuffer array and parse out the beginning of each word using the “*skipChar” and “strChr” functions. Additionally, we also set a “\0” every time we find an empty space using “strChr” function, this allows us to indicate when then end of the word has been reached. If there is more than one empty space in succession, only the first empty space would be set to a “\0”.

Step 3

In step 3, we define function prototypes and implement their functionality such as exitFunc, pwdFunc, lsFunc, and cdFunc. The commands array is filled with cmdStruct structs that hold a list of commands with their name and associated command function. Next in the main function, the doInternalCommand is called which loops through the “commands” array and finds which command name matches the first command line argument in “args”. If there is a match in the “commands” array then the associated command function is executed by accessing the “.cmdFunc” property.

Step 4

In step 4, we define a function called “doProgram”, this function is used to execute the internal program after the command has been verified to be an internal command. The function allocates a character array that is large enough in memory to hold the command string. Then “sprintf” is used to send formatted output to the character buffer. Using that character buffer, we pass that into the “stat” function which is used to get status information about a specified file and is stored “status” variable in our code. A few checks are done such as checking if the file exists, if the information was retrieved, if it’s a regular file (eg. S_ISREG(status.st_mode) which is a is a macro used to interpret the values in a stat-struct) (which is a special feature of the C language) and if the file is executable (eg. status.st_mode & S_IXUSR != 0, S_IXUSR is a bitmask used to filter out the bit in st_mode which tells us whether or not the file is executable) (which is also a special feature of the C language). If all of those checks pass, then the “fork” function is called which splits the process into a parent and child process. Then there are three possibilities that can occur:

1. If process is the child and the pid of the process is the child process, then we call the “execv” function which takes the path to the file to execute and replaces the current running program.
2. If process is the parent & forkStatus is the pid, we wait for the child process to complete executing and free the buffer variable.
3. The last case is that the child process could not be created.