Nathaniel Trujillo
12 October 2021
CSCE 313-598

PA2 Report

This program imitates a linux shell by taking in user inputs one line at a time and executing each of the user requests fully by using a variety of functions, system calls, and I/O redirects
In this documentation I will cover my unique approaches to the following program abilities:

1. Continuous prompting with custom shell tag
2. Request class data structure
3. Parse user input and full execution using fork(), execvp, and waitpid()
4. Input/Output redirection
5. Piping between commands
6. Background processes
7. Directory handling commands
8. $-sign expansion

(1) A simple while loop with conditional statements is used to support this. System calls to get the user's name and time are used to implement the custom shell tag.
(2) To parse the user input my code iterates through the line using a string stream and a series of conditional statements to recognize what each of the words are. This nested loop implementation will loop through a command and its arguments. The code will push the existing request to a vector of requests if a "|" is detected and begins creating a new request. Each of these request's rely on my class data structure "Request" that I implemented in my code. This class holds values for a command, vector of arguments, a quote (for " and '), and a vector for I/O redirection requests. Each of these variables are populated by using conditional statements and loops.
(3) Since the user input is parsed into a vector of requests I simply iterate or access this vector any time I want to call an execvp. For each of the execvp() calls exists a fork(). The parameters for execvp() are defined by converting the string variables from the request class to c-strings.
(4) I/O redirection requests are defined at the user input where conditional statements recognize "<" and ">" symbols and store their respective file names in the I/O redirection request vector. This is a vector of structs named "io". Each io struct contains a file name and an operation. These variables are accessed later when the actual I/O redirection happens.
(5) My approach to piping the commands between each other was to iterate through the request vector and create a child process for each of the requests and manage the connection with the parent. When using I/O redirection I implemented another fork() to avoid interrupting the pipes.
(6) The background processes were implemented using conditional statements and creating a child process without a waitpid() in its parent. The pid's were stored in a vector and checked by a non blocking waitpid() using an iteration.
(7) Directory handling was implemented using system calls and conditional statements to recognize user requests.
(8) $-sign expansion was implemented by storing this entire program as a function and passing the ecnaspliation to the function and storing the returned value in place of the $ sign encapsulation where the function completes the updated command with updated argument.

Nathaniel Trujillo
12 October 2021
CSCE 313-598

Git:
https://github.com/CSCE-313-Tyagi-Fall-2021/pa2-implementing-a-linux-shell-n-trujillo/commit/63e790c9e8a91c650bfe2eb616e4a66915539e41

Demo: https://youtu.be/OiWkiRXeCDQ