Kyrstn Hall, EJ Lilagan

Dr. Xiaoyu Zhang

CS433 - Operating Systems

24 September 2022

Assignment 2 Report

Submitted Files:

- prog.cpp

How to Compile/Run the Program:

To compile: make
Execute: ./prog2

Features implemented:

- This project consisted of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process.

- **int main(int args, char *argv[])**
  - char command[] is where the user command is stored, char *args is for the command after it is split up into arguments, should_run is our flag to know whether to stop the program or not, and our history vector is also initialized here
  - If user enters the command "exit", should_run changes to 0 and ends the program
  - If command is "!!", it will display "No command history" if there is no previous commands, otherwise, it will display the last command and execute it
  - For regular commands - It will first go through a loop whether a space is needed before an &, for example, "ls&". It will then parse the command and execute it.

- **Splitting command to arguments - parse_command(char command[], char *args[])**
  - We first initialize a pointer named *tok that will go through a command, and point to arguments, as well as count that counts how many arguments there are in the command
  - tok = strtok(NULL, " ") is first executed in which tok is pointing at the argument before the first space. The arguments tok is pointing at will be placed into the args array and increments count, unless it includes a '<', '|', or '>', which is skipped over and doesn't increment count. Tok is also updated to point at the next argument until it is NULL
  - After the command is split into arguments in args[], we check the last index whether it includes an ampersand or not. If there is an ampersand, we set our bool ampersand to true, and set that index to NULL

- **Executing a command - execArgs(char command[], char *Args[])**
  - **Before it gets forked, it gets checked for the pipe (First part)**
    - First, set two integer variables to represent two child processes (child1 & child2) and another integer to represent the pipe file descriptor (fd[2].)
    - Next, set an if statement with the pipe function (pipe(fd)) for exception handling, which will abort.
    - Setting a switch statement that will read the following test cases from the fork of child1. If statement wouldn't work to fork child1 because the result of an assignment as a condition would not be performed. The following test cases check if the case is -1 (or child1 <= 0) to abort, and case 0 to close the first child (or file descriptor 0) to exit. Set to default when all is satisfied.
    - Similar to the first child, we will have a switch statement for the second child to check the following test cases to be performed. The following will be the same except for case 0. First, declare a signed size integer called nread that will be used to read the pipe, but before reading the pipe, it must close the second child (fd[1].) Next, nread will be assigned to the read(fd[0], command, sizeof(*command) - 1) where it reads the first child, with the command and the size of the command line. In the read func, it's important to include an asterisk inside the sizeof() since it will return the size of the char* instead of char[]. Then, there will be two condition statements where it aborts if nread < 0, and another condition where it will terminate the command line, and after the else statement, exit. Set to default and do nothing once all met. After all the cases, close both child processes (fd[0] & fd[1].) Once both are closed, check if child1 and child2 still have data to put one of the child processors at the wait stage. Proceed to the next part once completed.
  - **After pipe/no pipe found on command, it will get forked to normal (Second part)**
    - In the 1st step, we are forking a child process using fork(). The value is stored in pid.
    - If pid is a negative value, there was an error with the fork and displays a "Fork failed" and executes exit(1).
    - In step 2nd step (pid is 0), the child process will invoke execvp(). Execvp will pass args[0], where the command is stored, as well as args, or the whole arguments array. Execvp also returns a negative value if it is unable to run, thus, we are also checking if the value is less than 0 and displays a "Command not found" if unable to run, and an exit(1).
    - In the 3rd step, parent will invoke wait() unless the command includes an & at the end. It will check whether our bool ampersand is false. If false, it will invoke wait().

- **In_redirect - in_redirect(char* file)**
  - The passed parameter file is used to serve the input that will be opened.
  - As it is significant to use the standard input (<), it will be redirected to a command from a file such as "osh> sort < in.txt"

    - In the function, integer i will be set with the open function including file and O_RDONLY (Open for reading only.)
    - It will duplicate the open function to create a copy of an existing file descriptor, and will be followed up with a close function from integer i.

- **Out_redirect - out_redirect(char* file)**
    - Same as the in_redirect function it will have the same passed parameter to serve the output that will be opened.
    - As it is used for the standard output (>), the output will redirect the output of a command to a file.
    - In the function, integer o will be set with the open function including file, O_WRONLY (Open for writing only), O_APPEND (force the file pointer to point at the end of file only), O_TRUNC (cause the file to be truncated/shorten if it exists), O_CREAT (call to open() with a mode argument), and 0600 (octal 600 or 384 in decimal in order to perform a signed char.)
    - It will follow the same format after assigning integer o and will be followed up with a close function.

Additional:

- Two global variables were used in this programming assignment, which was ampersand and MAX_LINE.
    - The ampersand variable is a bool type that will be used to determine whether the parent will invoke wait(). It only invokes wait if a user enters an ampersand after their command
    - The MAX_LINE variable is defined as the maximum length command, which will be addressed in any of the character arrays that will be used for viewing the characters of any command typed.

Choice of Data Structure: We decided to use vectors for this programming assignment because it is easily utilized to insert new elements inside. As a result, we would assign our history variable as a string type to be used for tracking the characters that are saved in the shell. It will use the empty function that will check if the history is empty, the size function to get the last index element recorded to retrieve the last command recorded, and the push_back function that will get the last command recorded and push it into the history.

Lessons Learned/Re-learned:

Kyrstn Hall

- At first, it was confusing even with the starter code because I was unfamiliar with C. I had to do a lot of searches and reading to understand a lot of the C functions we used. After attending office hours, it became so much clearer on what the program is trying to do, especially what execvp() does. Being able to talk to classmates also helped, especially with those who had the same errors on gradescope. Because of this, we were able to get our ampersand to work, as we were unaware we needed to take into account a test case where there are no spaces. Me and my partner also started this assignment early and met up in the library computer lab to dedicate time in this program which was beneficial. I

also thought of having our own separate files as we had different approaches and to avoid editing each other's code.

EJ Lilagan

- Without any experience with C, it was a bit difficult to translate what is expected for this programming assignment. After reading the instructions from zybooks and other supporting websites on canvas, it gave me a better understanding as to what is needed. Additionally, attending office hours with Kyrstn, would clear any unclear or unsure questions that were in the way such as "Is the history feature supposed to run a certain way? How do we check the ps -aef issue on gradescope?" Furthermore, it resulted with benefits as we would end up getting full credit for the code section, due to time constraints, we would somewhat be unaware of what the issue may be on the pipe, but then with further research, the pipe would soon work on the program. As to what Kyrstn said, we would tend to meet at our available times to work on the assignment as early as possible and doing so made us balance our time to knock each step of what is noted in zybooks. While we did work on different files to do this programming assignment, we would compare and converse about the best way to implement the assignment and debug the issues one of us may have. In summary, this was a great way to practice pair programming and a great experience to work alongside with Kyrstn to learn and work as a team.

Sources:

- Other ways to parse a command in C++/C
- How the strtok function works
- Used to reference for A2
- What the '\0' does of a character string
- What cpu-api does and reference for coding this programming assignment
- Use of fgets() in C
- strcmp function
- How to use ssize_t from the <sys/types.h> library
- How to implement pipes in C