OS Lab Assignment 2

Implementation of mini unix terminal/interpretor

Code:

#include<string.h>

#include<unistd.h>

#include<stdlib.h>

#include<stdbool.h>

#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<signal.h>

#include<sys/wait.h>

#define BUFFERSIZE 256 //Max amount allowed to read from input

#define INITIAL\_PROMPT "\t\tLab2\_Assignment - Mini Linux Terminal\n" //initial display message

#define PROMPT "lab2\_assignment >> " //Shell prompt

#define PROMPTSIZE sizeof(PROMPT)

#define ERROR -1 //for when an error is encountered

pid\_t pid;

void display\_Prompt(){printf("%s", PROMPT);}

// // Function to manage file I/O redirection

// void fileIOManager(char \*\*argv, char \*source, char \*destination, int option, char \*\*data\_){

//   if((pid == fork()) == ERROR) {perror("Error: Unable to create child process.\n");return;}

//   if(pid == 0){

//     int fd; //file descriptor

//     //file output redirection

//     if(option == 0){

//       fd = open(destination, O\_CREAT | O\_TRUNC | O\_WRONLY, 0600); //create a file for writing only

//       dup2(fd, STDOUT\_FILENO); //standard input is replaced with our file

//       close(fd); //close file

//     }

//     //file input and output redirection

//     if(option == 1){

//       fd = open(source, O\_RDONLY, 0600); //create a file for reading only

//       dup2(fd, STDIN\_FILENO);

//       close(fd);

//       fd = open(destination, O\_CREAT | O\_TRUNC | O\_WRONLY, 0600); //same process for output redirection

//       dup2(fd, STDOUT\_FILENO);

//       close(fd);

//     }

//     if(execvp(argv[0], argv) == ERROR) {

//       perror("Error: Command not found.\n");

//       kill(getpid(), SIGTERM);

//     }

//   }

//   waitpid(pid, NULL, 0);

// }

// int FileManager(char \*\*inputcommands,int n){

//   pid\_t pid;

//       if(strcmp(inputcommands[2],">") == 0 && n==4){ //Output Redirection

//           dup2(1,3);//Storing the standard output on ID 3

//           int fd = open(inputcommands[3],O\_WRONLY); //create a file for writing only

//           dup2(fd,1); //standard input is replaced with our file

//           close(fd); //close file

//           inputcommands[2]='\0';

//           inputcommands[3]='\0';

//           execvp(argvAux[0], argvAux) == ERROR

//           dup2(3,1);//Bring back the standard output to terminal

//           printf("--Output Redirection Successful--\n");

//           return 1;

//       }

//       else if(strcmp(inputcommands[1],"<")==0 && strcmp(inputcommands[0],"cat")==0 && n==3){  // Input Redirection

//           // char \*cmd[3];

//           // cmd[0]=inputcommands[0];

//           // cmd[1]=inputcommands[2];

//           // cmd[2]='\0';

//           // execute(cmd);

//               FILE \* file;

//               if (file = fopen(inputcommands[2], "r"))

//               {

//                 char c = fgetc(file);

//               while (c != EOF)

//                 {

//                   printf ("%c", c);

//                   c = fgetc(file);

//                   }

//                   fclose(file);

//                 }

//               else

//               {

//                 printf("--File doesn't exist--\n");

//               }

//           return 1;

//       }

// }

// Function to manage piping

void pipeManager(char \*\*argv){ //  ifconfig | grep -w inet -> 2 processes

  // fd = file descriptor

  int fd1[2], fd2[2], commands\_count = 0, aux0 = 0, aux1 = 0, aux2 = 0, end\_of\_Command;

  char \*commTok[BUFFERSIZE];

  for(int i = 0; argv[i] != NULL; i++)

    if(strcmp(argv[i], "|") == 0) commands\_count++;

  commands\_count++;

  while(argv[aux0] != NULL && end\_of\_Command != 1) {

    aux1 = 0;

    //using auxiliary variables as indices and a pointer array to store the commands

    while(strcmp(argv[aux0], "|") != 0) {

      commTok[aux1++] = argv[aux0++];

      if(argv[aux0] == NULL) {end\_of\_Command = 1;break;}

    }

    commTok[aux1] = NULL; //to mark the end of the command before being executed

    aux0++;

    //connect two commands' inputs and outputs

    if(aux2 % 2 == 0) pipe(fd2);

    else pipe(fd1);

    pid = fork();

    //close files if error occurs

    if(pid == ERROR) {

      if(aux2 != commands\_count - 1) {

        if (aux2 % 2 == 0) close(fd2[1]);

        else close(fd1[1]);

      }

      perror("Error: Unable to create child process.\n");

      return;

    }

    if(pid == 0) {

      //first command: replace standard input

      if(aux2 == 0) dup2(fd2[1], STDOUT\_FILENO);

      //last command: replace standard input for one pipe

      else if(aux2 == commands\_count - 1) {

        if(commands\_count % 2 == 0) dup2(fd2[0], STDIN\_FILENO);

        else dup2(fd1[0], STDIN\_FILENO);

      }

      else{

        if((aux2%2) == 0) {

          dup2(fd1[0], STDIN\_FILENO);

          dup2(fd2[1], STDOUT\_FILENO);

        }

        else{

          dup2(fd2[0], STDIN\_FILENO);

          dup2(fd1[1], STDOUT\_FILENO);

        }

      }

      if(execvp(commTok[0], commTok) == ERROR) {

        perror("Error: Unknown command entered.\n");

        kill(getpid(), SIGTERM); //terminate signal if an error is encountered

      }

    }

    //close file descriptors

    if(aux2 == commands\_count - 1) {

      if(commands\_count % 2 == 0) close(fd2[0]);

      else close(fd1[0]);

    }

    else if(aux2 == 0) close(fd2[1]);

    else{

      if((aux2%2) == 0) {

        close(fd1[0]);

        close(fd2[1]);

      }

      else{

        close(fd2[0]);

        close(fd1[0]);

      }

    }

    waitpid(pid, NULL, 0);

    aux2++;

  }

}

bool compare(char \*s){

  if((strcmp(s,">") == 0) ||(strcmp(s,"<") == 0) ||(strcmp(s,"&") == 0)) return true;

  return false;

}

// Function to handle commands from user's input

int Command\_Execution(char \*argv[]){

  char \*argvAux[BUFFERSIZE-1]; //since its a string and the last char in string is \n is removed from calc or command

  //SHELL COMMANDS execution

  if(strcmp(argv[0], "clear") == 0) system("clear"),printf("%s", INITIAL\_PROMPT);

  else if(strcmp(argv[0], "exit") == 0) exit(0);

  else if(strcmp(argv[0], "cd") == 0) chdir(argv[1]);

  else{

    int i = 0, j = 0, background = 0, aux1, aux2, aux3;

    //puts the command into its own array by breaking from loop if '>', '<' or '&' is encountered

    while(argv[j] != NULL){

      if(compare(argv[j])) break;

      argvAux[j++] = argv[j];

    }

    while(argv[i] != NULL && background == 0){

      // check for any pipe in commands

      if(strcmp(argv[i], "|") == 0) {

        pipeManager(argv); return 1;

      }

      // cat file1 > file2

      else if(strcmp(argv[i], "&") == 0) background = 1;//file I/O redirection

      else if(strcmp(argv[i], "<") == 0){

        aux1 = i+1;

        aux2 = i+2;

        aux3 = i+3;

        //if arguments after '<' are empty, return false

        if(argv[aux1] == NULL || argv[aux2] == NULL || argv[aux3] == NULL){

          perror("Error: Insufficient amount of arguments are provided.\n");

          return -1;

        }

        else{

          //'>' would be two indices after '<'

          if(strcmp(argv[aux2], ">") != 0) {

            perror("Error: Did you mean '>' ?\n");

            return -1;

          }

        }

        //file output redirection

        fileIOManager(argvAux, argv[i+1], argv[i+3], 1);

        return 1;

      }

      else if(strcmp(argv[i], ">") == 0){

        dup2(1,3);//Storing the standard output on ID 3

        int fd = open(inputcommands[argv[i-1]],O\_WRONLY); //create a file for writing only

        dup2(fd,1); //standard input is replaced with our file

        close(fd); //close file

        execvp(argvAux[0], argvAux);

        // execute(inputcommands);

        dup2(3,1);//Bring back the standard output to terminal

        printf("--Output Redirection Successful--\n");

        return 1;

        // fileIOManager(argvAux, NULL, argv[i+1], 0);

        // return 1;

      }

      i++;

    }

    argvAux[i] = NULL;

    if((pid = fork()) == ERROR) {

      perror("Error: Unable to create child process.\n");

      return -1;

    }

    //process creation (background or foreground) - CHILD

    if(pid == 0){

      signal(SIGINT, SIG\_IGN); //ignores SIGINT signals

      //end process if non-existing commmands were used, executes command

      if(execvp(argvAux[0], argvAux) == ERROR) {

        perror("Error: Command not found.\n");

        kill(getpid(), SIGTERM);

      }

    }

    //PARENT

    if(background == 0) waitpid(pid, NULL, 0); //waits for child if the process is not in the background

    else printf("New process with PID, %d, was created.\n", pid);

  }

  return 1;

}

int main(int \*argc, char \*\*argv[]){

  char commandStr[BUFFERSIZE];//user input buffer

  char \*commandTok[PROMPTSIZE]; //command tokens

  int numTok = 1;//counter for number of tokens

  pid = -10;  //a pid that is not possible

  printf("%s", INITIAL\_PROMPT);

  while(1){

    //print defined prompt

    display\_Prompt();

    memset(commandStr, '\0', BUFFERSIZE); //memset will fill the buffer with null terminated characters, emptying the buffer

    fgets(commandStr, BUFFERSIZE, stdin); //stores user input into commandStr

    //considers the case if nothing is typed, will loop again

    if((commandTok[0] = strtok(commandStr, " \n\t")) == NULL) continue;

    //reset token counter to 1, then count all command tokens

    numTok = 1;

    while((commandTok[numTok] = strtok(NULL, " \n\t")) != NULL) numTok++;

    Command\_Execution(commandTok);

  }

  exit(0);

}

Output Screenshots:









