Project 1

Alex Benasutti

CSC345-01

September 20 2019

C Code:

```
//
//  project1
//  main.c
//
//
//  Alex Benasutti
//  CSC345
//  September 20 2019
//

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <wait.h>

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
   char *args[MAX_LINE/2 + 1]; /* command line arguments */
   char str[MAX_LINE/2 + 1];   /* placeholder for shell input */
   char *token;                /* holds string for argument parsing */
   char *hist = "No recent commands in history";   /* holds previous command history */
   char *cmd;                  /* holds arguments to be parsed */
   int should_run = 1;         /* flag to determine when to exit program -- user must run
"exit" */
   int background_process;     /* flag to determine when to run a process in the background --
& */
   int redirect_out;           /* flag to determine file redirection OUT -- > */
   int redirect_in;            /* flag to determine file redirection IN  -- < */
   int pipe_process;           /* flag to determine if the pipe operator is used */
   int argc;                    /* counts the number of arguments */
   int len;                     /* counts the length of the input submitted by the user */
   int pipefd[2];              /* pipe file descriptor */

   pid_t pid;                  /* process id */
   pid_t pid_pipe;             /* process id for pipe */

   while (should_run)
   {
       /* Reset shell flags and argc for next command */
       background_process = 0;
       redirect_out       = 0;
       redirect_in        = 0;
```

```
        pipe_process       = 0;
        argc               = 0;

        /* set up user input and flush stdout */
        printf("osh>");
        fflush(stdout);

        /* get user input */
        fgets(str,MAX_LINE,stdin);
        len = strlen(str);

        /* allocate memory for command string */
        cmd = malloc(MAX_LINE/2+1 * sizeof(char));

        /* set newline character '\n' to NULL */
        if (str[len-1] == '\n') str[len-1] = '\0';

        /*
        * execute previous command
        * echo command on users screen
        * command should be placed into history buffer as next command
        * basic error handling i.e. no commands in history
        */

        if (strcmp(str,"!!") == 0)
        {
            /* process previously submitted command */
            cmd = hist;
            hist = malloc(strlen(cmd) * sizeof(char));
            memcpy(hist,cmd,strlen(cmd));
            printf("%s\n", cmd);
        }
        else if (str[0] != '\n' && str[0] != '\0')
        {
            /* process command as normal */
            cmd = str;
            hist = malloc(MAX_LINE/2+1 * sizeof(char));
            memcpy(hist,str,len);
        }

        /* tokenize the command */
        token = strtok(cmd," ");

        while (token != NULL)
        {
            /* set each token to a different element of args */
            args[argc] = token;

            /* Find any pipes - can't set | to NULL just yet since that would terminate the
loop */
```

```
    if (strcmp(token,"|") == 0) pipe_process = argc;

    token = strtok(NULL, " ");
    argc++;
}
args[argc] = '\0'; // set final argument to NULL so execvp() may terminate

/* exit only if met as the first argument */
if (strcmp(args[0],"exit") == 0)
{
    should_run = 0;
    // break; honestly you should have just required a break statement
}

/* directory change (cd) only if met as the first argument */
if (strcmp(args[0],"cd") == 0)
{
    chdir(args[1]);
}

/* background process handling only if met as the last argument */
if (strcmp(args[argc-1],"&") == 0)
{
    background_process = 1;
    args[argc-1] = '\0';
}

/* redirection handling for 2nd to last argument i.e. [some lengthy command] > out.txt */
if (argc > 2)
{
    if (strcmp(args[argc-2],">") == 0)
    {
        args[argc-2] = '\0';
        redirect_out = 1;
    }
    else if (strcmp(args[argc-2],"<") == 0)
    {
        args[argc-2] = '\0';
        redirect_in = 1;
    }
}

/* time for fork our parents and children */
pid = fork();

if (pid == 0) // CHILD PROCESS #1
{
    if (argc > 0)
    {
        /*
```

```
 * use > or < redirection operators for filestream management - use dup2()
 * assume commands will only contain one operator or the other and not both
 */

if (redirect_out)
{
    // redirect data out
    int fdOut;
    if ( (fdOut = creat(args[argc-1], 0644)) < 0 )
    {
        printf("File error - could not create file.\n");
        exit(0);
    }
    dup2(fdOut, STDOUT_FILENO);
    close(fdOut);
}
else if (redirect_in)
{
    // redirect data in
    int fdIn;
    if ( (fdIn = open(args[argc-1], O_RDONLY)) < 0 )
    {
        printf("File error - could not open file.\n");
        exit(0);
    }
    dup2(fdIn, STDIN_FILENO);
    close(fdIn);
}

 /*
* use | pipe to allow output of one command serve as input for another
* have child create another child and use dup2()
* assume one pipe per command and NO redirection operators
*/

if (pipe_process)
{
    /* create the pipe and handle any errors it presents */
    if (pipe(pipefd) < 0)
    {
        printf("Failed to create the pipe.\n");
        exit(1);
    }

    /* fork another process for the child to execute */
    pid_pipe = fork();

    if (pid_pipe == 0) // CHILD PROCESS #2
     {
        /* redirection for STDOUT to pipe */
```

```
                    dup2(pipefd[1], STDOUT_FILENO);
                    args[pipe_process] = '\0';
                    close(pipefd[0]);

                    /* execute that first argument */
                    execvp(args[0],args);
                }
                else if (pid_pipe > 0) // PARENT PROCESS #2
                {
                    wait(NULL);

                    /* redirection for STDIN to pipe */
                    dup2(pipefd[0],STDIN_FILENO);
                    close(pipefd[1]);

                    /* execute those args after the pipe "|" */
                    execvp(args[pipe_process + 1], &args[pipe_process + 1]);
                }
            }
            else /* invoke execvp() */
                execvp(args[0],args);
        }
        else
        {
            fprintf(stderr,"You have zero arguments.\n");
        }
    }
    else if (pid > 0) // PARENT PROCESS #1
    {
        /* invoke wait unless command included & */
        if (!background_process) wait(NULL);
    }
  }
  return 0;
}
```

Requirement Analysis

Child process creation - RED TEXT

By initializing process ids (pids) with the fork() process, we can create both a parent and child process which we can execute based on the value of this process id. If the process id is greater than zero, the original parent process is executing. However, if the process id is equal to zero, the child process is executing. By setting up a conditional for these ids, we can perform many of the different tasks needed for this simple shell program. For example, within the original child process, I am executing the code necessary to implement both the redirection and pipe operators, as well as the user-submitted command. This child process is also executing another fork() for handling the pipe operator, thus creating another child and parent process. This child process handles the first command of the pipe, while the parent process handles the second.

Command history management - ORANGE TEXT

History management within this shell program is not like the normal "history" command in UNIX shells. This command, "!!" outputs and displays the previously used command of the user. A char* hist is initially declared as the holder of any previous command that the user inputs. However, if there are no previous commands (i.e. the first command the user enters = "!!"), then an error "No recent commands in history." will output to the terminal. When the user enters a normal command, *hist* becomes populated by first allocating memory to it, and then having the memory from the entered command be copied into *hist.* If the user were to enter "!!", the set command would become whatever information is loaded onto *hist.* New memory is allocated for this command and is again copied to *hist* in preparation for another use of "!!" in the shell.

Add support of input and output direction - BLUE TEXT

Flags are first declared for determining whether file redirection is OUT or IN. While post-processing each argument within *args*, we strcmp() the second to last argument to either the redirect out ">" or redirect in "<" operators. Once found, the flag is turned on and the argument is set to NULL. In the child process, file descriptors are declared and created (for >) or opened (for <) for the text file argument and dup2() duplicates them onto either standard output or

standard input, respectively. This means once we close our declared file descriptors, our text file arguments can act as an output or input for the command sitting behind the redirection operator.

Allow the parent and child processes to communicate via a pipe - PURPLE TEXT

The pipe process allows the output of one command to serve as the input for another. I initially declare a pipe flag to determine if the pipe operator is used, as well as a pipe file descriptor so the output of one command can communicate with the other. While tokenizing the arguments given by the command, I'm actively searching for the "|" operator using strcmp(), and if found, setting the pipe flag to the current argument number. Once able to actually start a pipe process within the child process, I first set up and create the pipe using the file descriptor. By utilizing the pipe process I talked about earlier, I created another fork() to start a child and parent process, where the child, using dup2() again, redirects standard out to the pipe. By setting the "|" operator argument to NULL, I can execute *args* up to that first NULL, resulting in only the first command being executed. After the child process finishes, the parent process can execute the second command, where we redirect standard in to another element of the pipe. The arguments after the pipe operator can then be executed, using the output of the first args as the input for the next ones.

Keep working directory information (change directory by [cd] command) - GREEN TEXT

By checking the *args* variable's first argument for "cd", we can change the user's current directory by using the chdir() command. So long as there is a second command after the "cd", the user will be taken to that directory.