CSE 333 - OPERATING SYSTEMS
Programming Assignment # 2

Akif Batur – 150111854

prompt: 333sh:</CurrentWorkingDirectory>$ [command and its arguments]

Step by Step:

As the first step, the program begins with clearing the screen, in the main() function. After this system call, we are getting the path of the self executable (which is our shell itself) to set the SHELL environment variable. So where ever we execute the shell, there will be our SHELL path.

Then in an infinite while loop we are reading the command and its arguments into a char array, via using the fgets() function. After reading the whole line, we are setting up an char array that is dedicated to keep the previously issued commands. We will mention about history array later on this report...

When we finish setting up the history we then split the command line from white spaces, tabs or new line characters while counting the argument size. In the same loop, the shell will be trying to detect if the user is trying to executing a PIPE or an IO operation at the same time. If so, then two variables PIPE or IO is set to 1.

```
if(args[0]!=NULL) //If first argument is not NULL
{
```

First Step: Check if it's a background process or not

Before we begin to explain our code, we may need to mention about how do we differentiate processes if it's a background or a foreground process, and it's easy as 1,2,3...

```
if(!strcmp(args[argSize],"&")) //If last argument is an &
{
    background = 1; //It's a background process
    args[argSize] = NULL; //No need & in argument list
    argSize -=1; //Decrement argument size
}
```

B. Internal (build-in) commands:

We begin to explaining with the internal commands because at each iteration, we check the first argument to determine the operation and determining the internal commands are the easiest part. We just checking the equality between the first argument and the command string (except CTRL^C) such as: cd, clr, print etc... If any of them is detected the special block of code begins to executing appropriate operations.

```
if(!strcmp(args[0],"cd")) //Change current working directory
```

{

1. If the second argument is NULL get the current working directory and print it.
2. If the second argument is not NULL then try to change the current working directory. If change is fails that means the directory is not exist.

}
else if(!strcmp(args[0],"clr")) //Clear the screen
{

1. Clear the screen with “clear” system call.

}
else if(!strcmp(args[0],"print")) //Print all of the environment variables or a specific one
{

1. If the second argument is NULL print all of the environment variables.
2. If the second argument is not NULL then get the desired environment variable and print its value. If the environment variable is not exist, print the error message.

}
else if(!strcmp(args[0],"set")) //Try to set an environment variable
{

1. If the second argument is NULL then print the error message.
2. If the second argument is not NULL then keep the environment variable name and the value of the given environment variable.
3. Try to set the environment variable with the given value.
4. If it fails, that means there is no environment variable with the given name.

}
else if(!strcmp(args[0],"where")) //Find the path of the given command
{

1. If the second argument is not NULL, get the value of the PATH environment variable
2. Backup the PATH value
3. Split the PATH value from (:) in a while loop.
4. Concatenate a slash and the command name to the each piece of the PATH at each iteration.
5. Check if an executable is exist with the piece of PATH.
6. If it is not exist, check the current working directory.
7. If it's not exist in PATH pieces or in the current working directory, then print the error message.
8. If it is exist then print the path of the command.

}
else if(!strcmp(args[0],"path")) //print PATH env. or append/remove some given path to it
{

1. If the second argument is NULL then print the error message.
2. If the second argument is (+) then,
    - If the third argument is NULL then print the error message
    - If the third argument is not NULL then get the current PATH value, concatenate a (:) sign and then concatenate the new path value piece.
    - Print the new PATH value.

3. If the second argument is (-) then,

- If the third argument is NULL, then print the error message.
- If the third argument is not NULL, then get the current PATH value,
- Split the PATH from (:) in a while loop
- Check the equality between the PATH piece and the given value.
- If they are not matching then concatenate the PATH piece to a char array and concatenate a (:) sign.
- If they are matching, then continue.
- After this operation, there will be an extra (:). We then just assing a zero value as the last character of the array

}

else if(!strcmp(args[0],"exit")) //try to exit

{

1. Check the background process counter
2. Background process counter is decremented in the exitSignal() function and incremented at the system command execution part.
3. If the background process counter greater than zero, then print that user cannot exit.
4. If the background process counter less than zero, call the exit() function and terminate the shell.
5. This command depends on catching the signal when a process is terminated and we handle that in the exitSignal(int sgnl) function that catches the SIGCHLD. Body of the function is goes like this:

```
//return status information immediately, without waiting for the specified
        //process to terminate.
int pid = waitpid (WAIT_ANY, &status, WNOHANG);
if(pid > 0) //a background process is terminated
{
        if(backgorundSize > 0) //do not fall into negative values
                backgorundSize--; //decrement bacground process counter
}//do nothing for the foreground processes
```

}

C. I/O Redirection

else if((args[1]!=NULL)&&(IO)) //I/O Redirection

{

dup2() function arguments:

- **STDIN_FILENO:** Standard input value, stdin. Its value is 0.
- **STDOUT_FILENO:** Standard output value, stdout. Its value is 1.
- **STDERR_FILENO:** Standard error value, stderr. Its value is 2.

open() function arguments:

- **O_RDONLY:** Open for reading only.
- **O_TRUNC:** If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length shall be truncated to 0, and the

mode and owner shall be unchanged.

- **O_CREAT:** The file shall be created
- **O_RDWR:** Open for reading and writing. The result is undefined if this flag is applied to a FIFO.
- **O_WRONLY:** Open for writing only.
- **O_APPEND:** If set, the file offset shall be set to the end of the file prior to each write.

1. Since IO is detected at the top, we need the third argument at minimum possiblity such as: command > fileIn. If the
2. If any of the these <,>,>> sign is detected at the beginning of the main function, this block of code will be executed.
3. First we determint the option by checking <,>,>> signs in a while loop.
4. Then we get the command, fileIn and fileOut values in the another while loop.
5. Create a child process.
6. Parent will be waiting for the child process to be terminated.
7. **Check the option:**
    - If the third argument is NULL then continue to get the new command.
    - If the third argument is not NULL, then;
        - Now there will be 5 possibilities each of which has a unique pattern such that:
            1. command < fileIn → <
            2. command > fileOut → >
            3. command >> fileOut → >>
            4. command < fileIn > fileOut → <>
            5. command < fileIn >> fileOut → <>>
        - These patterns determines our option.

If option == “<”

1. Open fileIn for read only.
2. Print error message if fileIn does not exist
3. Set the fileIn as the new standard input by using dup2() function.
4. Get the command path.
5. Execute the command with execv() function.
6. exit()

If option == “>”

1. Open fileOut for write.
2. Create fileOut if does not exist
3. Set the fileOut as the new standard output by using dup2() function
4. Get the command path.
5. Execute the command with execv() function.
6. exit()

If option == “>>”

1. Open fileOut for write and append.
2. Create fileOut if does not exist
3. Set the fileOut as the new standard output by using dup2() function

4. Get the command path.
5. Execute the command with execv() function
6. exit()

If option == “<>“

1. Open fileIn read only.
2. If fileIn does not exist print error message.
3. Open fileOut for write.,
4. Create fileOut if does not exist.
5. Set fileIn as the new standard input.
6. Set fileOut as the new standard output.
7. Get the command path.
8. Execute the command with execv() function.
9. exit()

If option == “<>>”

1. Open fileIn read only.
2. Print error message if fileIn does not exist.
3. Open fileOut for write and append.
4. Set fileIn as the new standard input.
5. Set fileOut as the new standard output.
6. Get the command path.
7. Execute the command with execv() function.
8. Exit()

}

D. PIPE

else if((args[1]!=NULL)&&(PIPE)) //PIPE
{

dup2() function arguments:

- **STDIN_FILENO:** Standard input value, stdin. Its value is 0.
- **STDOUT_FILENO:** Standard output value, stdout. Its value is 1.
- **STDERR_FILENO:** Standard error value, stderr. Its value is 2.

1. If third argument is NULL, then go to top for the new command
2. Create a child process.
3. Create an integer array with two elements to keep the descriptors.
4. Split the command line from (|) sign.
5. First part will be the first command and its arguments, second part will be the second command and its arguments.
6. Split the first part from spaces to differentiate the arguments.
7. Split the second part from the spaces to differentiate the arguments.
8. Create a pipe by using pipe, and place the descriptors array: pipe(pipeArray)
9. Create a child process.
    1. At the child process block;
        - Set the pipeArray[0] as the new standard input by using dup2() function.
        - Close pipeArray[1]

- Find the command path of the second command part.
- Execute execv() function for second command part.

2. At the parent process block;
   - Set the pipeArray[1] as the new standard output by using dup2() function.
   - Close pipeArray[0]
   - Find the command path of the first command part.
   - Execute execv() function for first command part.

}

A. System commands

else
{

1. Create a child process to execute commands:
2. As the first step we copy the last argument of the command as the command path.
3. As the second step we need to determine if the command will be executed with execl() or execv() function:
   - If the last argument of the command has a slash (/), then;
     - Check if the last element of the last argument is equal to the first argument.
     - If so, the command will be executed with execl() function.
     - Otherwise the command will be executed with execv() function.
4. If it's an execl() execution, command path is already given by the user. Then we will not need the last argument in the arguments list. So we can make it NULL. We then just simply execute the command with execl() function. If there is no such command in the given path, the execl() function will be failed and the user will be warned with an error message.
5. If it's an execv() execution, then we need to find the path of the command and it's a simple task. We just created a function called execvPath() and when we pass the first argument which is the command, it returns the command path if it does exist. Otherwise execv() will be failed and the user will be warned with an error message.
6. In the execvPath() function, we just get the PATH environment variable and split it from (:) sign. At each iteration we concatenate the command with the path piece and we check if an executable is does exist.
7. At the parent code block:

We were already decided if the parent process will be waiting for the child process or not at the beginning of the code.

- If it's a background process:
- Don't wait for the child and print the prompt.
- Increment the background process counter by one.
- If it's a foreground process:
- Wait for the child process to be terminated or stopped.

}
} //end of the main() function

History (Previously issued commands):

We simply keep the previously issued commands in a char array with size of 10. The array is globally defined so at anywhere of the code we can use it, and at anytime the user command can be stored at execution time.

The history array almost works like a stack data structure basically. When the user enter a new command it is stored at the first place of the history array if it is empty. We then simply shift the array one place when the user enter a new command at each iteration. And this is goes till all the empty spaces of the array filled. After that point, every time the user enter a new command, the history array will be shifted one place and the command will be placed on the top of the array.

To accomplish these tasks we need to differentiate if the command will be executed directly or executed previously. We just check if the input of the user has a (!) sign. If it does, then we check if the history array contains a command at given index such as: !n (n is a number from 1 to 10). If there is a command previously issued at that index we just copy it to buffer and place it on the top of the history array once again. If the user enter something like this: !! that means user trying to execute the command at the 0th position of the history array. So we check if it does exist, then as the same process we copy the command to the buffer and place the command on the top of the history array. After we copy the previously issued commands into the buffer we then just start to checking for the appropriate code block to execute the command.

History array processing codes are placed between #HISTORY ARRAY SETTINGS BEGIN# (after reading the user input) and #HISTORY ARRAY SETTINGS END# (before splitting the command into its arguments) comments.

CTRL^C:

We handle the CTRL^C signal by using ctrlcSignal(int sgnl) function. It catches the SIGINT every time the user press CTRL+C buttons, then the control passes to the ctrlcSignal() function. In the function body we are checking the history array. If it has any element we print them all. If the history array is empty we print that the warning message: history is empty. After that we print our prompt to get the new input of the user.

Two functions: leftTrim(char *str) and rightTrim(char* str)

When we try to get the file names at the IO operations we realized that there is a space placed at the beginning of the fileIn and at the end of the fileOut values. So we are using trim those spaces from the file names for the correct operation.

CTRL^D:

At the top of the main while loop, we check the return value of the feof(stdin) function. When the user press CTRL+D buttons at anytime, while loop continues for the new command, but because of feof(stdin) returns true, program terminates.