Operating System Project #2 Report

Instructor: Asst. Prof. Ali Haydar ÖZER

Mustafa Onur Izmitlioglu 150114012

Ferhat Ozkan 150115009

A simple shell program

What is it and How does it work?

As we are told by our instructors and lab assistants, we implemented a simple shell program that works on terminal, with the given functionalities and components.

After running the executable file of the source code before we implemented, the main functionalities such as obtaining a command from user and dividing these commands into smaller pieces (in other words, tokens) were already existing for us to build our implementation onto it. So, there aren’t any changes made to the setup() function.

First of all, at the beginning of the code, there are a few header files added.

* + - sys/types.h : This header file includes the definition, “pid\_t” ,which is used for determining the process ID’s. We used pid\_t to store and change the childPID, that represents the child’s process ID. Changes on the process ID’s occur when a new process is executed (fork() method call). In our implementation, since we were working on the *foreground* and *background* processes, we should use the information of the process ID whether another command can be prompted on the *myshell:* , or not.
    - sys/wait.h : Header file that includes the wait() function. Wait function is an important and special function that allows the calling process to obtain information about it’s child process. This function suspends the execution of the calling process, until either status information of a terminated child process is available, or a delivery of a signal to execute a signal-catching function or to terminate the process.
  + stdbool.h : This is the header file which includes one of the most important, but not included in the standard C library, the bool data type. We needed to use the bool to implement the single-linked-list data structure. The linked-list data structure is our choice for the B part of the assignment, the bookmark functionality. Each bookmarked command are kept inside this linked-list, that will be explained much more detailed in the following report.
  + fcntl.h : Header file is used for determining the file creation flags and the file access flags. In the C part of this assignment, we worked on *I/O redirection*, where we had the duty to read on and write to files. Flags are also important for the definitions made right after these header file inclusions.

Then, there are some function prototype definitions, which are:

* + CREATE\_MODE, CREATE\_FLAGS, CREATE\_APPENDFLAGS and CREATE\_INPUTFLAGS.

These functions are used in Part C (I/O Redirection) of the program, that will be explained in more detailed in Part C.

After the header file inclusions and prototype definitions, there are the global variables:

* + pid\_t childpid
  + int temp,check,counter

The most important function that we implemented is the execute() function. Inside the while loop of the main function, the execute() function call is made with the parameters char \*args and int background.

Args, which is an array of pointers hold each of the command-line-arguments of the commands, and the background variable of type int holds the information that whether the currently running process is a background process (& is the last index for the background processes), or not.

execute() function is mainly built-on 2 cases, whether the background value is 0 or 1. For each case, the execution of the execute() function differs. The details of the execute() function will be explained for each of the parts given in the assignment ( Part A, Part B and Part C ).

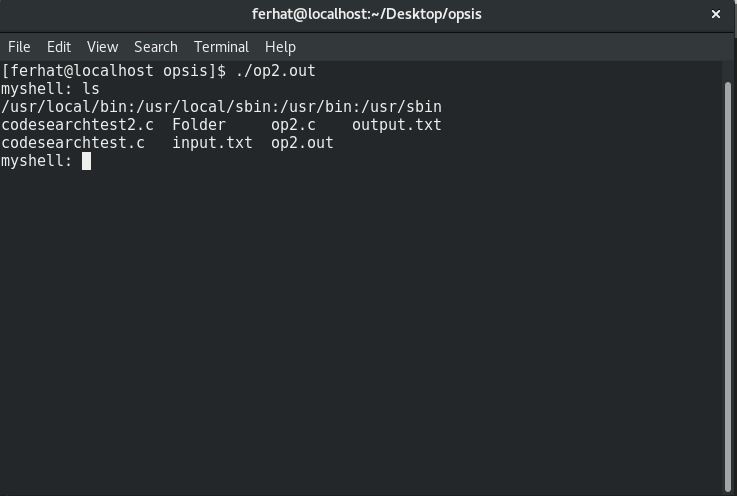
PART A

As it is mentioned earlier, childpid variable is one of the most important global variables that is used in execute() method. Childpid of type pid\_t presents the information of the process.

* + - * + childpid = fork();

fork(): When a fork() method call is made, the running process creates a new process. That new process executes the program as desired by the command. That child process uses execv() function to execute the command.

execv(patharr,args): This function takes a char array(patharr) and a char pointer array(args) to execute the value of char pointer array(which is the command line argument) that is under the path named patharr. To obtain the correct path of the command (e.g. /bin/ls for ls), we had to split the PATH environment variable value and search each of these subdirectories, whether the desired command exists or not.

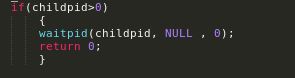




As you can see, each of the paths are divided with a colon, and to do that, we implied a splitter and checker for obtaining the correct path of the desired command.

*BACKGROUND AND FOREGROUND PROCESSES*

Background processes and foreground processes are differentiated in command-line base with the &(ampersand) sign. Input arguments that include & character will set background to 1 by the setup function that initializes args char array. If a process is executed in the background the parent process does NOT wait for the task to be completed. Despite of background processes, the foreground processes are straight-forward processes that keeps the terminal busy until they are terminated, so that the terminal waits for these processes to be executed and terminated.



PART B

bookmark: This internal command allows the myshell user to save a command for the current session of the terminal. Bookmark command has some functionalities which are listing, showing by index value, and deleting operations.

Listing operation is done by “bookmark -l” command.

Showing operation is done by “bookmark -i indexNum” command, where indexNum is the index number of the bookmarked commands.

Deleting operation is done via “bookmark -d indexNum” command, as in the showing operation, indexNum holds the indexNumber of the bookmarked command to be deleted.

While we were working on the bookmark case, we checked the second argument (args[1]) that the inputBuffer (whole command) has, there are 4 cases for args[1]:

* -l
* -i
* -d
* *command*

The case that checks whether the first argument (args[0]) is equal to “bookmark” in each of the background check cases. While working on the bookmark case, we implemented a linked list data structure to hold each of the bookmarked command. Linked lists helped us to maintain each of the commands, if a new bookmark is added its index will change depending on its corresponding place in the link list. At the same time if a delete command is invoked the link list structures will hold the order of the bookmarks.



For “bookmark -i indexNum” case, we implemented a new function named executebookmark(). The reason that we implemented such a function for only this case of bookmark is that, we should take care of the double quotes because the bookmarks hold the commands with these double quotes, and while we try to execute these commands by using -i indexNum, we try to run a command that is double quoted. Since commands do not include double quotes, we get rid of the double quotes inside that function, and we call execute() function inside executebookmark() function. This also allowed us to get flexibility on the new commands we have implemented such as codesearch, print and set commands.

This structure is some kind of a recursive call of execute function.



codesearch: This internal function allows the user to make a specific search in all of the source codes. The file format of these source codes are limited as .c, .C, .h and .H files.

In the implementation, we initially check if the first argument of the command is “codesearch” as we did in “bookmark” internal function. Note that, for each built-in function implementation, this check is made in the beginning.

After this check is made, then to protect the functionality of codesearch, we check if the second argument (args[1]) is equal to “-r”. This argument determines whether the desired search will be made recursively for each directory and its subdirectories. If there is no “-r” argument, the search will be done for the current directory.

In the “-r” case, “the NULL value of the search value” error is checked. If there is no error like this, then the args[2], the key to be searched, is taken and the “ “(double quotes) ” are deleted.

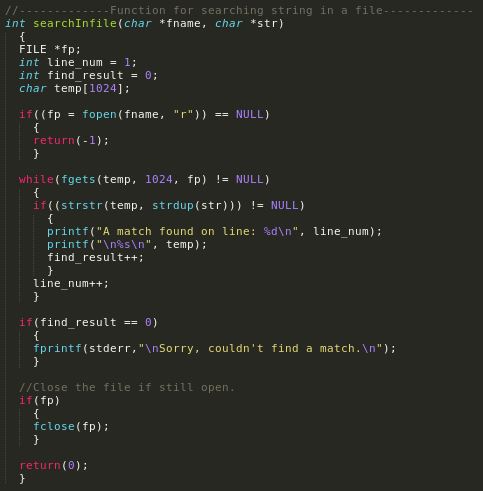
The searchstring[] holds the key to be searched, and the search is made with the function that we implemented, “recursivecodesearch ()”, if the args[1] value is equal to “-r”.

recursivecodesearch() function takes the parameters of char \*path, and char\* str. Path parameter determines the current working directory and str parameter holds the string to be searched. In recursivecodesearch () function, we open the directory with given path by using opendir() function, and we check whether if we are able to read content of the directory, by using the “readdir(DIR \*d) != NULL” statement. If we are able to read directory contents and if we have found our desired files, we call the searchInfile() method that we implemented to search for the given string in these files. Note that, recursivecodesearch() function is recursively called, to perform the search operation in subdirectories.

searchInfile() function takes char \*fname and char \*str parameters as inputs. fname holds the file name that is being currently searched, and the str holds the key to be searched, like in recursivecodesearch() function. First we take all the line in a char array called temp. Then we check if the given string is included in this line with the help of function strstr().

* (strstr(temp, strdup(str))) != NULL

The statement above is the key statement of codesearch function. If this statement returns true, then we print all of the information of the line that holds the searched key and the line number the string is found.



print: This built-in function allows the myshell: user to obtain the current value of an environment variable.

What is an environment variable?

An environment variable is a named, dynamic value that can affect the behavior of a running process on the computer. This concept came to life with Unix operating systems. In other words, environment variables are stacks (not in data structures, in means of groups) of variables that can be accessed by all of the applications.

For the mainSetup.c, we basically control if the first argument (args[0]) is “print”, if this statement returns true, then we use the getenv(args[1]) function that returns the value of the environment variable, which is in form of (name = value). If args[1] is NULL, which means the argument field is empty, then getenv() function is called with no input argument, that outputs the list of all environment variables with their values.

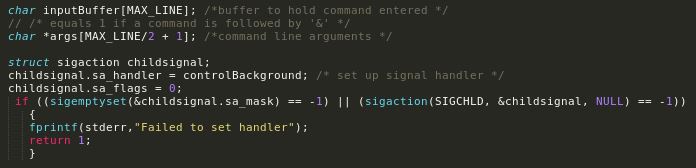


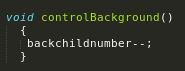
set: This internal function allows the shell user to set an environment variable’s value as the specified one by the user.

In the implementation, the control of first argument is done, whether it is “set”, and then the third argument check is done, whether it is equal to “=”, finally, the setenv() method calls are done for each case. setenv() method takes 3 input parameters; const char \**name,* const char \**value,* int *overwrite.* The first argument takes the name of the environment variable, the second argument takes the new value of the environment variable, and the final argument takes 0 or 1 value that represents if the new value should be overwritten(1) or not(0). Since we wanted to assign a new value, we set the third argument as 1.

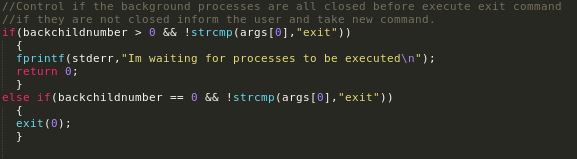


exit: Exit function basically terminates myshell . However, there are two cases that must be considered on the terminal runs. There can be some background processes that has NOT done it’s task, so there is an addition of a statement, in the beginning of the execute() function, that checks whether there are background processes still running, or not. If there are no background processes running and the first argument is “exit”, then the shell is terminated with an exit() function call. If there are background processes running and the first argument is “exit”, then the shell warns the user that “there are processes waiting to be executed” and the wait() function call is made. To perform this, while there are background processes that are running, we needed to use signals, that signal holds the information of the child processes. If the backchildnumber variable, which holds the number of child processes running in background, is greater than 0 (which means there are background processes not terminated), then the shell warns the user that there are still processes to be executed. The control of this backchildnumber is used in the signal handling explained above.





Every time a fork() is called and a new child is created the backchildnumber will increase by one. Since processes not working on background will wait for its child to finish executing the backchildnumber counter will increase by one the child will die and the counter will decrease by one. But on the other side background processes will continue executing in background and wont decrease the counter before it is closed. We can control this by using a signal handler that listens every child process. Every time a process dies the controlBackground function will be invoked and the counter will be handled. By this we could control the number of processes running.



PART C

File I/O Redirection:

In this part of the project, we basically worked on the three file types, which are *stdin*(for the keyboard that input is taken from), *stdout*(for the screen that output is sent to) and *stderr*(for the error messages that are output to the file determined by shell user). I/O redirection helps the user to;

* either write the output of a program to a file in 2 ways;
  + either by truncating to that file,
  + or appending to that file,
* or the user is able to use a content of file as an input to the program that user determined,
* or write the error of a program to a file specified by user,
* or execute a program that takes inputs from a file, and writes the output of this program into another file.

To perform each of these operations, we needed to use flags that are specified for file I/O redirection.

In the beginning of the code, we see that there are file status flags, that we have talked about earlier.

The first function file status flag, *CREATE\_MODE*, allows the file flags for the mode declaration to be used. What we mean by mode declaration, is that, there are permissions for each of the system owner, group and others. As you can see in the input parameter of this function, there are 4 variables, which are, S\_IRUSR,S\_IWUSR, S\_IRGRP, S\_IROTH.

S\_IRUSR file mode bit, is for “read permission of owner”.

S\_IWUSR file mode bit, is for “write permission of owner”.

S\_IRGRP file mode bit, is for “read permission of group”.

S\_IROTH file mode bit that is for “read permission of others”.

*CREATE\_FLAGS* allows the file flags for the program output that will be written to the file specified. The flags that are used in *CREATE\_FLAGS* are, O\_WRONLY, O\_TRUNC, O\_CREAT.

O\_WRONLY file mode bit, is for “open for write-only”.

O\_TRUNC file flag bit, is for “checking if file exists and is a regular file, and the file is successfully opened O\_RDWR or O\_WRONLY, its length is truncated to 0”.

O\_CREAT file mode bit, is for “checking the existence of the specified file”.

*CREATE\_APPENDFLAGS* allows the file flags for the program output that will be written to the file specified. The flags that are used in *CREATE\_FLAGS* are, O\_WRONLY, O\_APPEND, O\_CREAT.

O\_WRONLY file mode bit, is for “open for write-only”.

O\_APPEND file mode bit, is for “checking if file exists and is a regular file, and the file is successfully opened O\_RDWR or O\_WRONLY, the file offset will be set to the end of the file prior to each write”.

O\_CREAT file mode bit, is for “checking the existence of the specified file”.

As you can see, the *CREATE\_FLAGS* and *CREATE\_APPENDFLAGS* are used for “>” and “>>” operations in file i/o redirection, only difference is whether the output file will be truncated or appended.

CREATE\_INPUTFLAGS allows the file flags for the input files. The only parameter this function has is O\_RDWR.

O\_RDWR file mode bit, is for “open for reading and writing”.

< > case: In the C code that we implemented, we check if the “<” and “>” characters exist, which stands for the execution of a program that takes inputs from a file, and writes the output of this program into another file. Then we take the relevant files (args[index-3] as inputfile and args[index-1] as outputfile) and opened each file with open() function with the pre-defined CREATE\_INPUTFLAGS, CREATE\_FLAGS and CREATE\_MODE definitions.

< >> case(*READ*): However, we thought that we should also check “<” and “>>” case, whether the output of this program could be written to the outputfile in an appended form, rather than truncated, as a bonus. So we basically followed the same procedure as we did in “<” and “>” case. Rather than CREATE\_FLAGS, we use CREATE\_APPENDFLAGS in this case.

* open(finput,CREATE\_INPUTFLAGS,CREATE\_MODE)
* open(foutput,CREATE\_APPENDFLAGS,CREATE\_MODE)

Since we are working on both input and output files, we open 2 files with the necessary input parameters.

> case: After we started to implement the I/O redirection part of the project, we understood that we almost apply the same steps, we always need to use the open() function, but according to the operation, we call this open() function with different parameters. In > case, we call open() function as:

* open(foutput,CREATE\_FLAGS,CREATE\_MODE),

where, the first input parameter is foutput, that shows we will make changes on output file.

>> case: Rather than “>” case, the output file will be appended, not truncated, so we use CREATE\_APPENDFLAGS flag, not CREATE\_FLAGS(because CREATE\_FLAGS flag is used for truncated write to output file.).

* open(foutput,CREATE\_APPENDFLAGS,CREATE\_MODE);

< case: “<” case is used for the program inputs that will be obtained from a file.

open(finput,CREATE\_INPUTFLAGS,CREATE\_MODE);

2> case: Commands with this statement outputs the errors of a program and writes them to a user-specified file.



In addition to the informations above a important function used for i/o redirection cases is dup2. Dup2 function allows us to redirect the input, output and error of the current process. In this piece of code we have used both input and output redirection since we will take the input from a file and redirect the output to another file. This is done by dup2(fd,STDIN\_FILENO) which makes the process to take input from fd (input file) and dup2(fd,STDOUT\_FILENO) which redirects the standart output of the process to file fd2.

The more detailed explanations of each code segment will be done upon request in demo.

Screenshots of outputs obtained by the code are attached in the zip file in SCREENSHOTS folder.