CSC 60. Fall 2019. Lab 10. **Write your own UNIX Shell, part 2.**

Student will work with process management and some basic system calls.
Important note: please use sp1, sp2, sp3, or atoz servers for this lab.

UNIX Shell
In Lab9 we did the 3 built-in commands: cd, pwd, exit.
Now we need to implement: a fork, an exec, and code to handle redirection.

FILES TO COPY- Different Instructions this time:
To get the file you need, first move to your class folder by typing: **cd csc60**
Type: **cp -R /gaia/home/faculty/bielr/files_csc60/lab10 .**
Spaces needed: (1) After the **cp** ↑ Don't miss the space & dot.
(2) After the **-R**
(3) After the directory name at the end & before the dot.
You have now created a lab10 directory and copied in three sample files:
execvp.c, redir.c, waitpid.c
Stay in directory **csc60**, and you need to:
type: **chmod 755 lab10**
type: **cp lab9/lab9.c lab10/lab10.c**
We have copied lab9 code and renamed it to lab10.c for you to start work on it.

Next move to **lab10** directory by typing: **cd lab10**
and type: **chmod 644 ***
This will set permissions on all the files.
Your new lab10 directory should now contain: lab10.c, waitpid.c, redir.c, and execvp.c
The files: waitpid.c, redir.c, and execvp.c, contain code examples that may help you in completing Lab10.

A lot of code to be used in Lab10 is currently commented out.
Use the file **lab9-10 RemoveCommentsGuide.docx** (on Canvas) to guide you to properly remove the extra comments…without removing Every Comment!

Pseudo Code (Yellow highlight indicates the code from Lab9.)

```
/*----------------------------------------------------------*/
int main (void)
{
   while (TRUE)
   {
        int childPid;
        char *cmdLine;

        print the prompt();     /* i.e. csc60mshell > , Use printf*/
```

→ **more on next page**

```
        fgets(cmdline, MAXLINE, stdin);
        /* You have to write the call. The function itself is provided: function parseline */
        Call the function parseline, sending in cmdline & argv, getting back argc

        /* code to print out the argc and the agrv list to make sure it all came in. Required.*/
        Print a line. Ex: "Argc = %i"
        loop starting at zero, thru less than agrc, increment by one.
            print each argv[loop counter] [("Argv %i = %s \n", i, argv[i]);]

        /* Start processing the built-in commands */
        if ( argc compare equal to zero)
            /* a command was not entered, might have been just Enter or a space&Enter */
            continue to end of while(TRUE)-loop

        // next deal with the built-in commands
        //   Use strcmp to do the test
        //   after each command, do a continue to end of while(TRUE)-loop
        if ("exit")
            issue an exit call
        else if ("pwd")
            declare a char variable array of size MAX_PATH_LENGTH to hold the path
            do a getcwd
            print the path
        else if ("cd")
            declare a char variable dir as a pointer (with an *)
            if the argc is 1
                use the getenv call with "HOME" and
                        return the value from the call to variable dir
            else
                variable dir gets assigned the value of argv[1]

            execute a call to chdir(dir) with error checking. Message = "error changing directory"

        else    /* fork off a process.     This section was commented out for lab9. */
        {
            pid = fork();
            switch(pid)
            {
                case -1:
                    perror("Shell Program fork error");
                    exit(1);
```

→ **more on next page**

```
                case 0:
                    /* I am child process.
                    * I will execute the command, call: execvp */
                    process_input(argc, argv);
                    break;
                default:
                    /* I am parent process */
                    if (wait(&status) == -1)
                        perror("Shell Program error");
                    else
                        printf("Child returned status: %d\n",status);
                    break;
            } /* end of the switch */
        }     /* end of if-else-if that starts with EXIT
    }       /* end of the while(TRUE)-loop
}           /* end of main
```

---

void process_input (int argc, char **argv) // Child Process

{

call **handle_redir** passing it argc and argv

call **execvp** passing in argv[0] and argv and return a value to an *integer* variable (Example: ret)

if (ret == -1)

error check and do _exit(EXIT_FAILURE)

}

---

void handle_redir(int argc, char *agrv[]) // Child Process

{

You need two integer variables to keep track of the location in the string of the redirection symbols, (one for **out_loc** (>), one for **in_loc** (<)). Initialize them to zero.

for loop from 0 to < argc

if (“>” == 0) // use strcmp function

if out_loc not equal 0

Cannot output to more than one file. fprint error. _exit failure.

else if loop_counter compares equal 0

No command entered. print error. _exit failure.

set out_loc to the current loop_counter.

→ **more pseudo code on next page**

```
        else if ("<" == 0)          // use strcmp function
                if (in_loc not equal 0)
                        Cannot input from more than one file.  print error. _exit failure.
                else if loop_counter compares equal 0
                        No command entered. print error. _exit failure.
                set in_loc to the current loop_counter.
        // end of the if
    // end of the for loop

    if(out_loc != 0)
        if argv (indexed by out_loc +1) contains a NULL
            There is no file, so print an error, and _exit in failure.
        Open the file using name from argv, indexed by out_loc+1,
            and assign returned value to fd.  [See 9-Unix, slides 6-10]
                use flags:  to read/write; to create file if needed;
                        to truncate existing file to zero length
                use permission bits for:  user-read; user-write
        Error check the open. perror &  _exit
        Call dup2 to switch standard-out to the value of the file descriptor.
        Close the file
        Set things up for the future exec call by setting argv[out_loc] to NULL
    // end of if(out_loc != 0)

    if(in_loc != 0)
        if argv (indexed by in_loc +1) contains a NULL
            There is no file, so print an error, and _exit in failure.
        Open the file using name from argv, indexed by in_loc+1
            and assign returned value to fd.  use flags; for read only
        Error check the open. perror & _exit
        Call dup2 to switch standard-in to the value of the file descriptor.
        Close the file
        Set things up for the future exec call by setting argv[in_loc] to NULL
    //end of if(in_loc != 0)
```

Word of warning: In the past many students have duplicated the code for OpenOutputFile to be used for OpenInputFile. If you do that, lots of little things need to be changed. Be careful.

→ **more on next page**

Resources

Useful Unix System Calls: See PowerPoint Slides file named **Lab10 Slides**

C Library functions:

```
#include <string.h>
String compare:
    int strcmp(const char *s1, const char *s2); //Function prototype from string.h

    if(strcmp(argv[0], "exit") == 0)        //Sample. One line completed.
    strcmp(argv[0],"pwd")
    strcmp(argv[0],"cd")
    strcmp(....,">")
    strcmp(....,"<")

print a system error message:
    perror("Shell Program error \n");
```

Compilation & Building your program

The use of *gcc* is just fine. If you want to have the executable with a different name than a.out, type:

gcc –o name-of-executable name-of-source-code

or

gcc name-of-source-code –o name-of-executable

Partnership

Students may form a group of 2 students (maximum) to work on this lab. As usual, please always contact your instructor for questions or clarification. Your partner does not have to attend the same section.

All code files should include both names.

Using **vim**, create a small name file with both of your names in it. When you start your script file, *cat* that name file so both names show up in the script file.

You must BOTH submit your effort. As both of your names occur on everything, when I or another grader find the first submission, we will then give the same grade to the second student.

Marks Distribution

Lab 10 is worth 76 points.

→ **more on next page**

Hints

Our compiler does not like: **for (int i = 0; …..)**

You will receive the following errors:

test_loopcounter.c:6: error: 'for' loop initial declarations are only allowed in C99 mode
test_loopcounter.c:6: note: use option -std=c99 or -std=gnu99 to compile your code

These errors imply that on every "gcc" line, you must add: -std=c99 OR -std=gnu99.

It does like it on two lines:

```
int i;
for (i = 0; ……)
```

Keep versions of your code. This is in case you need to go back to your older version due to an unforeseen bug/issue.

A lot of code to be used in Lab10 was commented out.
Use the file **lab9-10 RemoveCommentsGuide.docx** to guide you to remove a set of the extra comments…without deleting Every Comment.

Deliverables

Submit **two** files to Canvas**:**

1. lab10.c
2. YourName_lab10.txt
   - Your program's output test (with various test cases).
   - Please use the UNIX **script** command to capture your program's output.
   - Details below. (Do not include lab10.c in this file)

→ More on next page.

Preparing your script file:

Be located in **csc60/lab10** directory.
When all is well and correct, type: **script StudentName_lab10.txt**

At the prompt, type: **gcc lab10.c -Wall** to compile the code
type: **a.out** to run the program
Enter in sequence:

1. *If you are on a team*, cat *your name file here.*
2. ls > lsout // should work with output going to file
3. cat lsout // display the contents of the output file

4. ls > lsout > file1 // should produce an error
5. cat foo.txt // should produce an error
6. > lsout // should produce an error
7. < lsout // should produce an error

/* ***wc*** *prints newline, word, and byte counts for each file* */

8. wc < lsout // output will go to the screen.
9. wc < lsout > wcout // output will go to a file
10. cat wcout // display the output
11. wc < lsout < wcout // should produce an error

12. cd ../lab1 // move to lab1 directory
13. gcc lab1.c // show that the exec works
14. a.out // show output of lab1
15. exit // (exit from the shell)
16. exit // (exit from the script)

When finished, submit your two files (the C file and the Script file) to Canvas.

(The script file will NOT contain the contents of lab10.c).

No zip files please.