CSC 60 Introduction to System
Program UNIX – Writing Your Own UNIX Shell

Due: **March 28, 2016 - by 8:00 am (No late work will be accepted)**

Purpose and rationale

The purpose of this lab is to allow students to learn a user interface aspect of a Unix shell. Student would work with process management and some basic system calls.

Important note: please use sp1, sp2, sp3, or atoz server for this lab.

Lab 4 – UNIX Shell

A simple shell is a basic shell program that supports commands with I/O re-direction. Once, a user enters a command string (ended with a return key), your program should parse the command line and determines whether it has a regular command or contains I/O redirection signified by > for output to a file or < for input from a file. The program has also 3 built-in commands: cd (change directory), pwd (print working directory), and exit (exit shell). The built-in functions are not executed by forking and executing an executable. Instead, the shell process executes them itself. All other command must be executed in a child process.

Your shell is basically an interactive loop: it repeatedly prints a prompt "**csc60msh >** ", parses the input, executes the command specified on that line of input, and waits for the command to finish. You should structure your shell such that it creates a new process for each new command (except for cd, pwd, and exit). Running commands in a new process protects the main shell process from any errors that occur in the new command.

Please follow these steps:

On athena, copy all source files from /home/ftp/nguyendh/lab4 directory to your lab4 directory. Review the source codes, compile, and execute the programs. Examine the output texts to understand the behavior of each program.

I have provided you with a simple shell template (csc60mshell.c) that you can choose to work from. You build your program from it. There is no other resource(s) allowed. Read the template closely to identify the key components and understand its execution flow. You can compile the shell using the following command: gcc *csc60mshell.c*

Input processing, fork, execute: work on the basic core functions by completing the code labeled as step 1 in csc60mshell.c. Here, with a new shell, we want to establish the basic functions by executing simple command like ls, cat, etc. The commands are parsed by a provided function called “parseline”.

- Handling built-in Commands: There are four special cases where your shell should execute a command directly itself instead of running a separate process. First, if the user enters **"exit"** as a command, the shell should terminate. Second, if the user enters **"cd dir"**, you should change the current directory to "dir" by using the chdir system call. Users can run programs with paths relative to the working directory without specifying an absolute path. For example, instead of typing "/a/b/c/myprog", a user could type two commands: "cd /a/b/c" followed by "cd myprog". If the user simply types **"cd"** (no dir specified), change to the user's home directory. The $HOME environment stores the desired path; use getenv("HOME") to obtain this. Third, if the user enters **"pwd"**, print the current working directory. This can be obtained with getcwd() function. Four, if the users enters **"history"**, you will display the top **10** previous executed commands. User then can choose to execute any command from this list via an exclamation mark and its number. For example, enter !7, will execute the command number 7 from the list . This is similar like what you did in your first lab.

  Handling input, output redirection: complete the code for section labeled as step 2. Please also review the sample code for redir.c. You should be able to reuse part of the code here. Below are the high level steps:

  Implementing "<" and ">":

  - Open the file
    - Eg. newfd=open(.....)
  - Assign newfd to 0(if "<") or 1(if ">") using **dup2**
    - Eg. dup2(newfd,0);
  - Close your file
    - Eg. close(newfd);
  - Execute the command using execvp system call
    - The command reads from 0 which is your file now

  Note: Pipe function will be handled in the next assignment.

- Handling input errors: Be very careful to check for error conditions at all stages of command line parsing. Since the shell is controlled by a user, it is possible to receive bizarre input. For example, your shell should be able to handle all these errors:

  csc60mshell > /bin/cat < foo < gub

  ERROR - Can't have two input redirects on one line.

  csc60mshell > /bin/cat <

  ERROR - No redirection file specified.

  csc60mshell > > out.txt

  ERROR - No command. Make sure file out.txt is not overwritten.

  csc60mshell > cat test.c >

ERROR - No redirection file specified.

(Note: even though the Unix shell MIGHT not provide you an explicit error message for the above cases, the actual expected result is incorrect i.e /bin/cat < foo < gub)

- Handling test cases: Your program should be able to handle the following examples of commands. However, your program will be tested with more than these commands (including error handling cases).

csc60mshell > ls > ls.out # redirect ls's stdout to file ls.out

csc60mshell > cat foo.txt

csc60mshell > wc <

ls.out csc60mshell > cd

/usr/bin csc60mshell > ls

csc60mshell > cd ../

csc60mshell > pwd

csc60mshell > /usr/bin/ps

csc60mshell > cd

csc60mshell > find . -name foo1.txt

csc60mshell > wc foo1.txt

csc60mshell > exit

csc60mshell > pwd

csc60mshell > cd hw5

csc60mshell > cat foo1.txt ls.out > out.tx

csc60mshell > gcc –o test test.c –g

csc60mshell > <carriage-return>

csc60mshell >

Marks Distribution

Your documented c program: csc60mshell.c with core functions –no I/O redirection, no build-in function	20 points
Your full documented c program: csc60mshell.c – build-in function	5 points
Your full documented c program: csc60mshell.c – I/O redirection	10 points
Your full documented c program: csc60mshell.c – with error checks	5 points
TOTAL	40 points

Resources

Useful Unix System Calls:

getenv/setenv: get/setenv the value of an environment
variable path = getenv("PATH");
cwd = getenv("PWD");
`setenv("PWD", tempbuf,1);`

getcwd: get current working directory.

chdir: change the current working directory (use this to implement cd)

fork-join: create a new child process and wait for it to

```
exit: if (fork() == 0) { // the child process
        ..........
} else { // the parent process
        pid = wait(&status);
}
```

execv: overlay a new process image on calling process
execvp(full_path_name, command_argv_list, 0);

open, close, dup2: for I/O Redirection: int
fid = open(foo, O_WRONLY|O_CREAT);
close(1);
dup2(fid,1);
close(fid);

Exit shell:

_exit(…); - exit without flushing
stdio exit(); - exit with flushing stdio

C Library functions:

```
#include <string.h>
String compare:
int strcmp(const char *s1, const char
*s2); strcmp(….,"cd")
strcmp(….,"exit")
strcmp(….,"pwd")
strcmp(….,"history")
strcmp(….,">")
strcmp(….,"<")

print a system error message:
perror("Shell Program error");
input of characters and strings:
fgets(cmdline, MAXLINE, stdin);
```

Compilation & Building your program

To compile and build your executable program, please reuse the makefile, for assignment 3, for this new assignment. Please make appropriate changes where they are applicable.

Partnership

Students may form a group of 2 students (maximum) to work on this lab. As usual, please always contact your instructor for question or clarification.

Deliverables

Your source file(s): csc60mshell.c, your program's output text (with various test cases – please use the UNIX **script** command to capture your program's output), and your makefile. Please submit your assignment via SacCT.

Hints

Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. Please see the resources section above.

Remember to get the **basic functionality** of your program working before worrying about all of the error conditions and corner cases. For example, for your shell, first get a single command running (probably first a command with no arguments, such as "ls"). Then try adding more arguments. Next, try working on multiple commands. Make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands. Finally, support for built-in commands, redirection, and pipes (for next assignment).

I strongly recommend that you check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. And, it's good programming practice.

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

I hope you enjoy this project.