CS330-001: Assignment 1

Fall 2019 (201930)

#### Due Date and Time: Wednesday, September 18, 2019 at 4:20 PM

**(100 marks)** When you log onto Hercules, the operating system starts running a *program* called the *shell*. The standard shell for the Linux operating system running on Hercules is called the *C shell* (the program name is tcsh)n. The purpose of the shell is to package system services under the control of an *interactive command interpreter* whose external interface is suitable for human consumption. When the shell program (a *passive* entity) is run, it becomes a shell *process* (an *active* entity). The shell process (hereafter just called the shell) outputs a *command prompt* to the screen and then “waits” to read input entered from the keyboard. The shell is not actually waiting, it is temporarily suspended until you type some characters. After you type some characters and press the return key to end the line, the shell re-starts and reads these characters into an internal array. The shell then analyzes the contents of the internal array to extract *commands* and *arguments*. For some commands, such as cd (i.e., change the current directory), the shell can execute the command directly because the command is *built-in* to the shell. When the shell finishes executing a built-in command, it outputs another command prompt to the screen. For other commands, such as ls (i.e., list the contents of the specified directory), the shell searches directories specified in the shell path environment variable, looking for a file named “ls” (type env at the command prompt to see the contents of your path environment variable and other environment variables). If the file named “ls” *exists* and is *executable*, the shell will invoke a service of the operating system to run the file (i.e., execute the command), thereby creating a new process. An & character can be used to terminate a command line. If there is no & character terminating the command line, the command is run in *foreground*. In foreground, the shell is temporarily suspended while it waits for the new process to finish executing. Once the new process is finished, the shell re-starts and outputs another command prompt to the screen. If there is an & character terminating the command line, the command is run in *background*. In background, the shell is not suspended and immediately outputs another command prompt to the screen. The new process runs *concurrently* with the shell.

A shell program is not really special in any way. In fact, there are actually many different shells available to choose from. And if you don’t like any of the available shells, you could write your own given the opportunity and sufficient motivation (e.g., see below). Read some C shell documentation to familiarize yourself with what a shell is, the features a shell provides, and the types of operations a shell supports. Some useful documentation can be found at <http://docs.freebsd.org/44doc/usd/04.csh/paper.html>, <http://linuxfinances.info/info/unixshells.html>, or by typing man tcsh at the Hercules command prompt.

In this assignment, you are required to write a shell program, called toyshell, that runs on the Linux operating system (specifically, one of the Linux machines in CL115 or equivalent). You can log onto these machines from your Hercules account. To find the list of available Linux machines in CL115, enter

cs\_clients CL115

at the Hercules command prompt. The output from this command will be a list of Linux machine names that look something like

a049402 a049403 a049404 a049405 a049406 a049407 a049408 a049409 a049410 a049411 a049412 a049413 a049414 a049415 a049416 a049417 a049418

To log onto one of these machines, enter

ssh a0494*xx*

at the Hercules command prompt, where a0494*xx* is one of the Linux machine names. If you are unable to log onto a particular machine, try a different one.

**PART 1**

Your toyshell program must provide services to support the following commands and features:

**Note**: The commands are shown in uppercase font for emphasis only, you don’t have to program them that way. Similarly, the parameters to the commands are enclosed between the symbols < and > for emphasis, but the symbols are not actually part of the parameters and should not be programmed that way.

1. When toyshell is first loaded, it should display the following command prompt:

toyshell[1]>

The command prompt consists of *three* parts: (1) the first part, toyshell, is the *shell name*, (2), the second part, [1], is the *command counter*, and (3) the third part, >, is the *terminator*. The command counter keeps track of and displays the number of the next command to be entered on the command line in the current session (i.e., since toyshell was last loaded). For example, the command prompt shown above indicates the next command to be entered on the command line in this session will be the first. Similarly, the command prompt

toyshell[27]>

indicates the next command to be entered in the current session will be the twenty-seventh. If the user presses enter at the command prompt, without actually entering anything on the command line, the command counter should not change. In all other cases, whether the command entered is valid or not, the command counter should be incremented.

1. STOP: Terminates execution of the current toyshell session.
2. SETSHELLNAME <*shell\_name*>: Sets the shell name in the toyshell command prompt to *shell\_name*. The new shell name should be persistent. That is, it should be used for the remainder of the current session and any subsequent sessions.
3. SETTERMINATOR <*terminator*>: Sets the terminator in the toyshell command prompt to *terminator*. The new terminator should be persistent. That is, it should be used for the remainder of the current session and any subsequent sessions.
4. HISTORY: Lists the commands that have been entered in the current session (i.e., since the last time toyshell was run). The commands should be stored in a structure of size 10 (i.e., it can hold up to ten commands). Commands should be added to the structure in sequence until it becomes full. Any command entered at the command prompt when the structure is full will require that the oldest command be removed from the structure before it can be added to the structure.
5. ! | <*n*>: Re-executes a command stored in the history structure. For example, the command ! 6 will cause the command in the 6-th position in the history structure to be executed again. The re-executed command should be added to the history structure as it was originally issued on the command line.
6. NEWNAME <*new\_nam*e> | <*new\_name*> <*old\_name*>: Manages a structure containing command aliases. The aliases should be stored in a structure of size 10 (i.e., it can hold up to ten aliases). The first option deletes a previously defined alias. The second option defines an alias for another command. For example, the command NEWNAME mymove deletes the alias for mymove, and the command NEWNAME mycopy cp defines mycopy as the alias for the cp command. If an alias for a command already exists, then the new alias replaces the old alias.
7. *Alias Substitution*: Every command entered on the command line should be checked to determine whether it is an alias by scanning the alias structure. When an alias is detected in a command, the old name should be substituted into the command before the command is executed. For example, assume the following aliases are defined as shown below:

aa ls –l

bb grep toyshell

cc grep .cpp

dd aa | bb

ee dd

Then, for the following sequence of commands at the command prompt:

aa

aa | bb

aa | bb | cc

dd | cc

ee

alias substitutions should result in the following sequence of commands being executed:

ls –l

ls –l | grep toyshell

ls –l | grep toyshell | grep .cpp

ls –l | grep toyshell | grep .cpp

ls –l | grep toyshell

1. NEWNAMES: Outputs all the aliases contained in the alias structure. Each command/alias pair should be shown on one line. For example, the possible aliases for a few commands are shown below:

mymove mv

mycopy cp

chkshell ls –l | grep toyshell.cpp

1. SAVENEWNAMES <*file\_name*>: Stores all currently defined aliases in a file whose name is *file\_name*.
2. READNEWNAMES <*file\_name*>: Reads all aliases stored in the file whose name is *file\_name* and adds them to the aliases defined in the current session. If an alias in *file\_name* already exists in the alias structure (i.e., it is a duplicate), it should be ignored.
3. <*Linux\_command*>: Executes the Linux command *Linux\_command*, corresponding to any valid Linux command. One approach to implementing this is to use the system function. If the first token on a command line is not a built-in toyshell command, assume that it is a Linux command.
4. *Error handling*: Your approach should effectively identify and recover from errors. For example, bad input and/or the inability to execute a command should not cause your toyshell to crash. To get credit for this component of the assignment, you must be able to demonstrate that you can handle at least ten unique errors.

**Note**: You must handle all the built-in commands with exactly the same syntax as shown above. Thus, an important part of the toyshell program will be to parse commands entered at the command prompt to break them down into their component parts. Once a command has been parsed, the component parts can be checked to ensure that a valid command has been entered and that it adheres to the required syntax. One approach to effective parsing is to use the C-string strtok function (to be covered in detail in the first lab).

**PART 2 (60 marks)**

For programming problems, the Results are worth 70%. So, for this problem, the results are worth 70 marks out of the 100 marks available. Demonstrate that your toyshell works and that it can handle all of the requirements described in Part 1, except for the last one (i.e., the error handling), as you will demonstrate that in Part 3. So, this part will be worth 60 marks out of the 70 marks available. Each correctly executed command will be worth one mark. Your demonstration should be captured in a script file.

Run your toyshell. It should display the command prompt shown below:

toyshell[1]>

Then, in your toyshell, enter the numbered sequence of commands given below:

1: ls –l

2: HISTORY (*type* 10 *spaces before entering the command*)

3: ! 1

4: SETSHELLNAME tsh

5: who

6: HISTORY

7: ! 2

8: ! 20

9: HISTORY

10: ! 8

11: NEWNAME mycopy cp

12: *press* Enter/Return *with a blank command line*

13: NEWNAME dog cat

14: SETTERMINATOR :

15: SAVENEWNAMES myaliases

16: NEWNAMES

17: dog myaliases

18: pico myaliases

*exit* pico

19: mycopy myaliases myfile

20: HISTORY

21: NEWNAME history history

22: ! 1

23: dog myfile

24: cat myfile

25: HISTORY

26: NEWNAME h history

27: NEWNAME mycopy rm

28: ls

29: mycopy myfile

30: ls

31: SAVENEWNAMES myaliases

32: dog myaliases

33: SETSHELLNAME myshell

34: NEWNAME mycopy

35: NEWNAME mycopy cp

36: mycopy myaliases mysavedaliases

37: SETTERMINATOR -:

38: READNEWNAMES myaliases

39: NEWNAMES

40: dog

a

b

c

*press* ctrl/D

41: h

42: ! 2

43: mycopy myaliases

44: h

45: ! 8

46: h

47: NEWNAME aa ls –l

48: NEWNAME bb grep toyshell *or* toyShell *or* ? (*whatever you called your* .cpp *file*)

49: NEWNAME cc grep .cpp

50: NEWNAME dd aa | bb

51: NEWNAME ee dd

52: aa

53: aa | bb

54: aa | bb | cc

55: dd | cc

56: ee

57: STOP

Run your toyshell. It should display the command prompt shown below:

myshell[1]-:

Then, in your toyshell, enter the numbered sequence of commands given below:

58: readnewnames mysavedaliases

59: newnames

60: stop

**PART 3 (10 marks)**

Finally, demonstrate that your toyshell can handle errors (i.e., the last requirement in Part 1). This will be worth ten marks out of the 70 marks available. In order to receive all ten marks, you must be able to demonstrate that you can handle at least ten unique errors. Each correctly handled error condition will be worth one mark. Your demonstration should be captured in a script file.

**WHAT TO SUBMIT**

*If you are working alone*, submit to UR Courses: (1) all your source code files (i.e., *only* the .cpp and .h files) zipped into a single file called cppandhfiles, (2) a single script file called part2script showing the compilation and execution of your toyshell from Part 2, and (3) a single script file called part3script showing the compilation and error handling capabilities of your toyshell from Part 3.

*If you are working with a partner*, *one of the partners* should submit to UR Courses: (1) a file named partners that provides the names and student numbers of the partners and the relative contributions of the partners (should total 100%), (2) all our source code files (i.e., *only* the .cpp and .h files) zipped into a single file called cppandhfiles, (3) a single script file called part2script showing the compilation and execution of your toyshell from Part 2, and (4) a single script file called part3script showing the compilation and error handling capabilities of your toyshell from Part 3. *The other partner* should submit to UR Courses: (1) a file named partners that provides the names and student numbers of the partners and the relative contribution of the partners (should total 100%). Note that *both* partners need to submit the partners file.