CS 270 Systems Programming Spring 2021

Project 4: Simple Shell

***Out**: Mon 12 April 2021* ***Due**: Wed 28 April 2021*

For this project you will implement a simple shell, called `simpsh`. The shell is a program that reads commands typed by the user and then executes them. It also allows users to set *variables* that can be used in subsequent commands. You may work in groups of two on this project (but you may also work individually if you wish). While we expect that each member of the group will take responsibility for writing some portion of the project, we also expect that each will educate the other member about the code that he/she wrote. Both members of the group will receive the same grade. You may write `simpsh` in C or C++.

1 Overview

`Simpsh` is similar to existing shells such as csh, bash, ksh, and tcsh, in that it reads input a line at a time from the standard input file (terminal), and produces output on the standard output file. However, it lacks much of the functionality found in other shells. It also has a different command syntax

> **Note:** In the following, when we refer to a "token" we mean either a single word or a string surrounded by double-quotes. Tokens are subject to variable substitution, as described later.

2 Commands

`Simpsh` supports the following commands. **Each input line must begin with one of the following commmands.**

- # *anytext* — The # command introduces a comment. `simpsh` ignores the rest of the line. The # must occur as the first token on the line; any # occurring other than the first token is a syntax error.
- *cd dirName* — This command changes `simpsh`'s current directory to *dirName*. You do *not* need to handle *cd* with no parameters—unlike a shell like `bash`, which understands the absence of parameters to "change to the home directory". (See the `getwd(3)` and `chdir(2)` system calls.) After this command, any program that `simpsh` starts will run with *dirName* as its working directory. This command also updates the value of the variable `CWD`, see below.
- *variable = value* — Here *variable* is any reasonable variable name (first character is a letter, others are letters or numbers), and the *value* is a single token. The spaces around the = token are required. The effect of this command is to store the value of the variable inside `simpsh` for later substitution. Certain special variables are maintained by `simpsh` itself, as described below.
- *lv* — this command ("list variables") causes `simpsh` to print a list of all the variables that have been assigned a value, including the predefined ones (see below).
- *unset variable* — the command causes the value of a user-specified variable to be forgotten. It is an error to try to unset any of the built-in variables.

- *! cmd param* [infrom: file0] [outto: file1]* This command tells `simpsh` to execute a program. The "!" must be the first token on the command line. Here, *cmd* is a token that specifies the filename of the program the user wants to execute. It is followed by zero or more tokens specifying parameters. The optional token pairs *infrom: file0* and *outto:file1* indicate that the standard input (respectively standard output) should be redirected from *file0* (respectively, to *file1*). Note that the colon is the final character in each token. Either *infrom:* or *outto:* or both may be omitted, and either may come first. If present, the keyword-filename pair must be the last tokens on the line. `simpsh` looks for *cmd* in a list of directories indicated by the special variable `PATH`, whose value is a colon-separated list of directories. By default, `PATH` has the value `/bin:/usr/bin`; the user can change that value by assigning a new value to `PATH`. However, if *cmd* starts with a "/" character, it is a full path name starting at the root of the filesystem. Similarly, if *cmd* starts with "./", it is understood to be a path name starting in the current directory.
- *quit* — This command causes `simpsh` to exit, with status 0.

Example Command Lines

```
MYDIR = $CWD/calvert
! ls /etc/passwd
! myprog infrom: test0.txt outto: outfile
lv
cd /etc
# this doesn't do anything!
```

3 Built-in Variables and Substitution

Certain variables are already defined when `simpsh` starts up. Some can be changed by assignment, but none of them can be unset.

PATH This is a colon-separated list of directories to be searched for commands to be executed via the `!` command. Initially set to `/bin:/usr/bin`.

CWD This is maintained by `simpsh` to always have the value of the current working directory. Cannot be changed (except indirectly, via the `cd` command).

PS The prompt string. This is printed before reading each line of input from the standard input.

When a line is an assignment to a new variable (i.e., three tokens, of which the first is a valid variable name (say `FOO`), the second is "=", and the third is anything), `simpsh` stores the name and value.

When `simpsh` encounters a variable name, preceded immediately by "`$`" (no space between `$` and the name) either as a token or a substring of a token, it should replace both the `$` and the variable name with the corresponding stored definition. It is a syntax error if `$` occurs before a string that is not the name of a variable.

It is valid to modify the definition of a variable by a later definition.

As an example, after the command "`FOO = gnus`", the result of the subsequent command

```
! wc infrom: $CWD/$FOO
```

would result in execution of the command `/bin/wc` with input redirected from the file `gnus` in the current directory. However, the command

```
! echo $FOOBAR
```

would result in an error message unless a variable named `FOO` had previously been given a value.

4 Scanning and Parsing

Handling a command proceeds in stages. The first step is to read the input line; you may use `fgets()` for this. You may assume that input lines are less than 256 characters long.

The next step is to divide the input line into a sequence of tokens, where a token refers to a sequence of characters that constitute a logical unit. To do this, you should write a *scanner routine*. The scanner has no knowledge about the *meaning* of individual tokens; it only knows how to group characters into tokens. To simplify your scanner, you may assume that tokens are separated by whitespace (spaces or tabs).

The scanner should be able to return a token that is a single word (that is, anything up to the next space or newline, which could be a number, a variable name, a file name, the special characters `!`, `#` and `=`, or anything that doesn't contain a space) and a token that is a string (a series of characters enclosed in double quotes, which might include spaces, but which does *not* include the quote marks at the start and the end).

For example, given the (nonsense) line:

```
  ! /bin/grep "Student Chapter" /etc/rc1.d = out $Param1 infrom: in ###
```

the scanner should produce these tokens:

```
!
/bin/grep
Student Chapter
/etc/rc1.d
=
out
$Param1
infrom:
in
###
```

5 Parsing

`Simpsh` passes the results of the scanner to a parser that determines if the syntax of the command is valid. The parser must determine how each token fits in its command line. For example, the parser might determine whether a word token is one of the above commands, the name of a program to be run, a parameter to a command, or a variable name to be assigned to. Once the parser sees what role each token is playing, it can determine whether the input is a valid command line.

6 Details

Write procedures to break the input line into tokens (scanner), and a parser routine that calls the scanner repeatedly until the scanner returns an end-of-line indication, and then analyzes the token sequence for validity. After it has scanned and parsed the command line, if the input syntax is not valid, `simpsh` should print an error message and prompt for the next line. If the syntax of the line is valid, `simpsh` should take the appropriate action.

If the command is `!`, i.e., a file is to be executed, `simpsh` should fork() and execve() a new process to execute the given file. **You may not use the Unix system() library routine to execute the command line.** After `fork()`ing, the parent should wait for the command it has just started to complete before issuing a prompt for the next command.

If the command includes the *infrom:* or *outto:* directives, after `fork()`ing, the child process should first attempt to open or create the named file(s), and if successful, close the indicated file descriptor (standard input and/or standard output, i.e., 0 or 1) and re-associate the descriptor number(s) with the newly opened file(s). (See the man page for `dup2()`.) Then it can `execve` the command.

Other suggestions:

- As the parser calls the scanner, it builds a list of tokens. When the scanner returns an end-of-line indication, the parser uses that list to analyze the command.
- Naturally `simpsh` should handle errors gracefully. For instance, it should print an informative message if the command line is invalid or if some file (e.g., the *cmd* it was asked to execute, or the file it was to redirect standard input from) does not exist. In addition, `simpsh` should not die, crash or exit (except in response to a *quit* command or EOF (Control-D). If it detects an error, it should print a message indicating the nature of the error and the command it is ignoring, and then display the prompt for the next command.
- ⟨control-D⟩ (end-of-file) on the input stream before any token should be treated exactly as if the user had typed *quit*.
- The C library provides several standard routines for dealing with strings, including `strsep()`, which behaves somewhat like a scanner.
- As always, the `man` program is your friend.
- If you know lex and yacc, (or their Gnu implementations, flex and bison), you may use use them to write the scanner and parser. (lex/flex builds a scanner from a description file, and yacc/bison builds a parser from a description file.) However, if you have not used these tools before, you are probably better off without them. These programs require that you understand regular expressions and context-free grammars, while the syntax of `simpsh` is simple enough that you don't need such heavy-duty tools; they are intended for scanning and parsing full programming languages, like C itself.

7 Turnin

Your code must compile and run on your Ubuntu VM or you will receive no credit.

Only one person from your group should submit the code. List both members from your group (if you are working in a group) in the README file.

Submit all your code plus a documentation file. Do not submit .o files or other binary files. To create your submission, tar and compress all files that you are submitting:

tar czf proj4.tgz proj4/

Your project directory should contain:

- README: a text file (not a Word or PDF document!) that lists your files, names the members of your group, and gives a brief description of your project including the algorithms you used, along with any special features or limitations of novsh. If you do not document a limitation, we assume you do not know about it, and that it is a bug.
- Makefile: typing "make" should compile `simpsh`.
- your C/C++ files
- your header files

You may upload proj4.tgz as many times as you like. Each submission overwrites the previous submission, so we will only have a copy of your last submission.