UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering

The Nutshell Term Project

COP4600 – Operating Systems

Overview

This team project involves writing a command interpreter for a Korn shell-like command language in C using Lex and Yacc running under Unix. Your shell will parse command lines and execute the appropriate command(s). The core of the shell consists of simple commands, pipes, I/O redirection, environment variables, aliases, pathname searching, and wild-carding. You will do well on the project if these features work well in your shell. For extra credit you may implement tilde expansion and automated command completion. Each requirement in the core section will be explained in the rest of this handout.

You are *required* to use Lex and Yacc for this project.[1] You should not hand-build tokenizers or parsers in your implementation.

It would be in your best interest to start working early on this project. **Only teams of two will be accepted. Teaming should be finalized by 11:59PM Friday March 19, 2021. Guidelines about how to pair up into teams will be provided via Announcements. Pairing is not allowed across the UF Online Degree section (section 06EH) and any other section.**

We will be grading your projects by running them against a set of test files that will exercise various features of the shell which we think are important. You should test your shell extensively covering all features individually and in combinations.

Your project will be graded on correctness, completeness, your level of testing, and the organization of your code.

Thursday discussion classes will cover the following to prepare you to carry on this project:

- Thursday March 18: Lex and Yacc
- Thursday March 25: Git II
- Thursday April 1: Linux Filesystem/Lab 6 or Open Support Session
- Thursday April 8: Open Support Session

All students should make themselves available during Thursday discussion sessions on April 15 and April 22. Grading some group projects might require a Zoom meeting with the group and the TA's and PM's. There will be no rescheduling allowed for these grading sessions.

The Shell

Your shell will accept commands from standard input (terminal or files) and execute them. The type of commands your shell will accept are detailed below, but first let us define *words* and *metacharacters* - important ingredients for shell commands.

[1]Or use flex/bison, or lex++/yacc++.

Words, Special Conventions and Metacharacters

- **Word.** We will use *word* to refer to a sequence of characters which are treated as a logical unit, sometimes referred to as a *token*. Words are separated by white space, newlines, and metacharacters. Any other character is valid in a *word*. If you wish a *word* to contain white space, then you must put double quotes around it. For example, your scanner should interpret `echo test > foo` as 3 words and one metacharacter. However, the command `echo "test > foo"` would be interpreted by the scanner as 2 words, `echo` and `test > foo` (note that the "" have been removed from `"test > foo"`).
- **White space.** White space consists of any combination of the characters: space and tab.
- **The character .** (dot) alone or as a first component of a path name refers to the current working directory.
- **The characters ..** (dot dot) alone or as a first component of a path name refer to the parent directory.
- **The tilde character** ~ alone or as a first component of a path name refers to and substitutes for the user's home directory.
- **Metacharacter.** Metacharacters are characters which have special meaning to the shell, and stand only for themselves. Metacharacters cannot be part of a *word* unless they are preceded by a \ or are inside quotes. The following are metacharacters:

  `< > | " \ &`

Built-in Commands

- **setenv variable word** This command sets the value of the variable *variable* to be *word*.
- **printenv** This command prints out the values of all the environment variables, in the format `variable=value`, one entry per line.
- **unsetenv variable** This command will remove the binding of *variable*. If the variable is unbound, the command is ignored.
- Mandatory environment variables include **HOME** which shows the home directory of the user and **PATH** which shows the list of paths to be searched to find a command's executable file.
- **Specific conventions regarding the PATH environment variable.** Your shell should interpret the value of PATH to be a list of colon-separated words (for instance, word:word:word). Your shell should reparse and do tilde (~) expansion at the beginning (first character) of each of these words whenever the value of the variable is reset
- **cd word** This command changes the current directory to word, where word is a directory name. The directory name may be absolute (starts with root which is /) or relative to the current directory. You must handle *cd* with no arguments, to take you back to the home directory, i.e., it should have the same effect as *cd* ~ (see Tilde Expansion).
- **alias name word** Adds a new alias to the shell. See the subsection on aliases for more information.
- **unalias name** Remove the alias for name.
- **alias** lists all available aliases
- **bye** Gracefully quit the shell. The shell should also exit if it receives the end-of-file character.

Other Commands

`cmd [arg]* [|cmd [arg]*]* [< fn1] [ >[>] fn2 ] [ 2>fn3 || 2>&1 ] [&]`

Any command of this form can be accepted along with its arguments, pipes, and I/O redirection if present. Note that the I/O redirection can only appear at the end of the line in your shell. The construct `2>file` redirects the standard error of the program to `file`, while `2>&1` connects the standard error of the program to its standard output. If *cmd* does not start with a /, the shell must check (search for) the directories on the path that is the value of the environment variable PATH for the command. The *cmd* should be run only if the file exists and is executable. It is a common practice to include the current directory denoted by "." in the PATH variable to allow for the search to include the current working directory. You are required to always include the current working directory in your PATH. You must also be able to handle I/O redirection on builtin commands: printenv and alias. If & exists at the end of the command line, then the shell will execute this command in the background. If & doesn't exist, then the shell will wait for this command to finish.

Aliases

You must implement a simplified version of the *ksh* alias mechanism. Your version will consist only of simple string substitutions, and will not provide any rearrangement of the arguments. The commands are of the following types:

alias The alias command with no arguments lists all of the current aliases.
alias name word This alias command adds a new alias to the shell. An alias is essentially a shorthand form of a long command. For example, you may have an alias `alias lf "/bin/ls -F"` set up so that whenever you type `lf` from the command line, the command that is executed is `/bin/ls -F`. Note that alias expansion is only performed on the *first* word of a command. However, aliases may be nested. Your shell has to detect when an infinite-loop alias expansion occurs.
unalias name The unalias command is used to remove the alias for `name` from the alias list.

It should be obvious that all alias expansions must be done before parsing the command or searching any paths for the command binaries to execute.

Environment Variable Expansion ${variable}

It is also possible to include environment variables as part of words inside command lines. The shell reads all the characters from ${ to the next } and assumes it is the name of a variable. The value, if any, of the variable is substituted.

Wildcard Matching

Many shells do filename generation with wildcarding. You will implement a subset of the functionality found in most shells. Before a command is executed, each command word should be scanned for the characters * and ?. If any of these characters appears the word is regarded as a pattern. The word is

replaced with alphabetically sorted filenames that match the pattern. If no filename is found that matches the pattern, the word is left unchanged but with the wildcard characters removed.

A * matches any string, including the null string. A ? matches any single character.

Examples of Commands

```
setenv PATH .:/usr/bin:/usr/local/bin:~ghi/bin:~/bin
setenv ARGPATH .:~ghi/bin:~/bin
cd ./bin
cd ~/bin
cd ~sjc/bin
cd ../misc/old
cd src/proj/first
ls project1
ls "~project1"
wc -l f1 f2 f3 | sort | page
command1 arg1 arg2 | command2 | command3 < file_in > file_out 2>&1 &
alias lo logout
alias rot13 "tr a-zA-Z n-za-mN-ZA-M"
rot13 < foo > bar
ls *.c foo.?
alias lo jj
alias
alias jj "ls -al"
lo
setenv this .
setenv lsthis "jj ${this}"
${lsthis}
bye
setenv LIB ~/bin
nm ${LIB}/libxc.a
```

Shell Prompt

You should design a simple prompt for your shell so that the user is aware when the shell is ready to take on new commands. The design could be for a fixed string or even a character (e.g., “% “) or a string composed from some variable such as the user name.

Helpful Hints

In order to parse the input line, you should use Lex and Yacc (documentation and tutorial will be provided). A suggested scheme is for the parser to build a command table which contains for each **basic** command: the command name, a count of its arguments, a pointer to a list of null terminated arguments, an input file name, and an output file name. (Note that you *may* not want to use this table for built-in command.) If a table is produced (i.e., the command is syntactically correct), then the shell should set up the pipes and I/O redirection as indicated by the table (if any), and fork and exec processes as needed, wait for children to exit, report any errors detected and repeat the process for the next command line. The table should be reset before use with every new command. Built-in commands should be taken care of inside the shell program itself. Errors should be handled gracefully, and the offending line number should be printed out if the session is not interactive. **In other words, the shell should never die, crash or exit unexpectedly.** If an error is detected, a message should be printed, the current command purged, and the prompt displayed for the next command. You may use neither `execlp` nor `execvp` within your shell, and you should not fork another shell (e.g., ksh, csh, sh, etc.) nor use the `system` nor `popen` system calls to run your commands.

You should implement only the features described in this assignment handout for your shell, not all the features of existing shells such as csh or ksh.

Extra Credit

Tilde Expansion

~**name** should be replaced with the home directory of user name.
~ when not followed by a user name should be replaced by the home directory of the current user.

You should only do tilde expansion at the beginning of a word. The rule for tilde expansion can be summarized as follows: find the substring starting with the character after the ~ and ending with either the end of the string or a /, whichever comes first. If this substring is null, use the value of the HOME environment variable, else look up the substring in /etc/passwd using `getpwnam()` and extract the user's home directory from the returned struct. You should not do tilde expansion inside quoted strings.

File Name Completion

When typing a command to be executed by your shell, your shell should complete a partially-typed filename or user name. If the word immediately preceding the cursor expands to an unambiguous filename (with the current directory providing the default context) your shell should do the expansion when the ESC character is typed. When the last (partial) word begins with a tilde, your shell should complete it with a user name instead. For example

```
cd ~russ ESC
```

expands to

```
cd /homes/russo
```

Helpful Unix Commands

`csh`
`open`
`close`
`dup (and dup2)`
`access`
`fork`
`execl`
`execve`
`chdir`
`ioctl`
`pipe`
`strings`
`perror`
`malloc`
`exit (and _exit)`
`getenv`
`getpwnam`
`environ(5v)`
`getpgrp`
`setpgrp`
`termio`
`wait`
`kill`

Turning in Your Shell

You will submit this project through Canvas. You will use the sample Makefile provided to you to generate an executable called “nutshell" when the command `make` is typed in the directory containing all your sources and the Makefile. You must also have a README file that *clearly* specifies the features that you have NOT implemented followed by a list of features that you have implemented. The first feature you must try to implement in your shell is the I/O redirection from and to files. This is because we will be testing your project by running it through a program that will invoke your shell with its standard input redirected to come from each of several test files. Should this feature not work well you will earn our wrath for making testing your shell a grading nightmare.

The README file should include in the first few paragraphs, a declaration of what each team member did in the project. This should be detailed enough for us to understand how to grade each of the team members individually if contributions are not balanced.