KAIST

EE 209: Programming Structures for EE

Assignment 6: A Unix Shell

(Acknowledgment: This assignment is borrowed and slightly modified from Princeton COS 217)

Purpose

The purpose of this assignment is to help you learn about Unix processes, low-level input/output, and signals. It will also give you ample opportunity to define software modules; in that sense the assignment is a capstone for the course.

Rules

Signal handling (as described below) is the "on your own" part of this assignment. That part is worth 8% of this assignment.

You will get an extra 20% of the full score if you implement file redirection. See the extra credit section below.

Background

A Unix shell is a program that makes the facilities of the operating system available to interactive users. There are several popular Unix shells: `sh` (the Bourne shell), `csh` (the C shell), and `bash` (the Bourne Again shell) are a few.

Your Task

Your task in this assignment is to create a program named `ish`. Your program should be a minimal but realistic interactive Unix shell.
A Supplementary Information page lists detailed implementation requirements and recommendations.

You can work on this assignment with a partner in your class. A team should be no more than 2 people in the same class. Submit just one copy to KLMS if you work in a team.

If you choose to work alone (e.g., without a partner) on this assignment, you will receive extra credit as described below.

Building a Program

Use `-D_BSD_SOURCE` and `-D_GNU_SOURCE` options when building your program.

Initialization and Termination

When first started, your program should read and interpret lines from the file `.ishrc` in the user's HOME directory, provided that the file exists and is readable. Note that the file name is `.ishrc` (not `ishrc`), and that it resides in the user's *HOME* directory, not the *current* (alias *working*) directory.

To facilitate your debugging and our testing, your program should print each line that it reads from `.ishrc` immediately after reading it. Your

program should print a percent sign and a space (`% `) before each such line.

Your program should terminate when the user types Ctrl-d or issues the `exit` command. (See also the section below entitled "Signal Handling.")

Interactive Operation

After start-up processing, your program repeatedly should perform these actions:

- Print to the standard output stream a prompt consisting of a percent sign followed by a space.
- Read a line from the standard input stream.
- Lexically analyze the line to form an array of tokens.
- Syntactically analyze (i.e. parse) the token array to form a command.
- Execute the command.

Lexical Analysis

Informally, a *token* should be a word. More formally, a token should consist of a sequence of non-white-space characters that is separated from other tokens by white-space characters. There should be two exceptions:

- The special character '|' should form separate token.

- Strings enclosed in double quotes (") should form part or all of a single token. Special characters inside of strings should not form separate tokens.

Your program should assume that no line of the standard input stream contains more than 1023 characters; the terminating newline character is included in that count. In other words, your program should assume that a string composed from a line of input can fit in an array of characters of length 1024. If a line of the standard input stream is longer than 1023 characters, then your program need not handle it properly; but it should not corrupt memory.

Syntactic Analysis

A *command* should be a sequence of tokens, the first of which specifies the command name.

The '|' token should indicate that the immediate token after the '|' is another command. Your program should redirect the standard output of the command on the left to the standard input of the command on the right. If there's no following token after '|', your program should print out an appropriate error message. There can be multiple pipe operators in a single command.

Execution

Your program should interpret four shell built-in commands:

`setenv` *var* [*value*]	If environment variable *var* does not exist, then your program

	should create it. Your program should set the value of *var* to *value*, or to the empty string if *value* is omitted. Note: Initially, your program inherits environment variables from its parent. Your program should be able to modify the value of an existing environment variable or create a new environment variable via the `setenv` command. Your program should be able to set the value of any environment variable; but the only environment variable that it explicitly uses is HOME.
`unsetenv` *var*	Your program should destroy the environment variable *var*.
`cd` [*dir*]	Your program should change its working directory to *dir*, or to the HOME directory if *dir* is omitted.
`exit`	Your program should exit with exit status 0.

Note that those built-in commands should neither read from the standard input stream nor write to the standard output stream. Your program should print an error message if there is any piped commmand or file redirection with those built-in commands.

If the command is not a built-in command, then your program should consider the command name to be the name of a file that contains code to be executed. Your program should fork a child process and pass the file name, along with its arguments, to the `execvp` system call. If the attempt to execute the file fails, then your program should print an error message indicating the reason for the failure.

Process Handling

All child processes forked by your program should run in the foreground. Your program need not support background processes.

It is required to call wait for every child that has been created.

Signal Handling

[NOTE] Ctrl-d represents EOF, not a signal. Do NOT make a signal handler for Ctrl-d.

When the user types Ctrl-c, Linux sends a SIGINT signal to the parent process and its children. Upon receiving a SIGINT signal:

- The parent process should ignore the SIGINT signal.
- A child process should not necessarily ignore the SIGINT signal. That is, unless the child process itself (beyond the control of parent process) has installed a handler for SIGINT signals, the child process should terminate.

When the user types Ctrl-₩, Linux sends a SIGQUIT signal to the parent process and its children. Upon receiving a SIGQUIT signal:

- The parent process should print the message "Type Ctrl-₩ again within 5 seconds to exit." to the standard output stream. If and only if the user indeed types Ctrl-₩ again within 5 seconds of wall-clock time, then the parent process should terminate.
- A child process should not necessarily ignore the SIGQUIT signal. That is, unless the child process itself (beyond the control of the parent process) has installed a handler for SIGQUIT signals, the child process should terminate.

Error Handling

Your program should handle an erroneous line gracefully by rejecting the line and writing a descriptive error message to the standard error stream. An error message written by your program should begin with "*programName*: " where *programName* is `argv[0]`, that is, the name of your program's executable binary file. Note that `argv[0]` typically will be `ish`, but need not be so.

The error messages written by your program need not be identical to those written by the given `sampleish` program. However, the error messages written by your program should be at least as descriptive as those written by `sampleish`.

Your program should handle all user errors. It should be impossible for the user's input to cause your program to crash.

Memory Management

Your program should contain no memory leaks. For every call of `malloc` or `calloc`, eventually there should be a corresponding call of `free`.

Extra Credit 1 (extra 20% of the full score of this assignment)

You are going to implement redirection of standard input and standard output as an extra credit.

- The special character '<' and '>' should form separate token in lexical analysis.
- The '<' token should indicate that the following token is a name of a file. Your program should redirect the command's standard input to that file. It should be an error to redirect a command's standard input stream more than once. It should also be an error to redirect a command's standard input stream after a pipe token (See Redirection section in supplementary information page).
- The '>' token should indicate that the following token is a name of a file. Your program should redirect the command's standard output to that file. It should be an error to redirect a command's standard output stream more than once. It should also be an error to redirect a command's standard output stream before a pipe token (See Redirection section in supplementary information page).
- If the standard input stream is redirected to a file that does not exist, then your program should print an appropriate error message.
- If the standard output stream is redirected to a file that does not exist, then your program should create it. If the standard output stream is redirected to a file that already exists, then your program should destroy the file's contents and rewrite the file from scratch. Your program should set the permissions of the file to 0600.

Extra Credit 2 (extra 30% of your earned score including the extra credit1)

If you do this assignment on your own without a partner, you will receive extra credit which is worth 30% of (your basic score + extra credit1). Here is an example. If your score is 50 and you got extra credit 1, your earned score is 50 + 20 = 70. If you worked alone, you will

receive 30% x 70 = 21 additional points as extra credit 2. So, your total score will be 91.

Testing

Test your program by creating multiple files containing lines which your program should interpret, repeatedly copying each to the .ishrc file in your HOME directory, and restarting your program. The file `.ishrc` contains a sequence of commands that can serve as a minimal test case. You should develop *many* more test files.

Of course you also should test your program manually by typing commands at its prompt.

Logistics

Develop on lab machines. Use `emacs` to create source code. Use `make` to automate the build process. Use `gdb` to debug.

An executable version of the assignment solution is available in `sampleish`. Use it to resolve any issues concerning the desired functionality of your program. We also provide the interface and implementation of the `DynArray` ADT that we discussed in precepts. You are welcome to use that ADT in your program.

Your `readme` file should contain:

- Your name and the name and the student ID of your partner. If you worked alone, only your name is needed.

- Description of work division between you and your partner.
- A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated, as prescribed by the course "Policies" web page.
- (Optionally) An indication of how much time you spent doing the assignment.
- (Optionally) Your assessment of the assignment.
- (Optionally) Any information that will help us to grade your work in the most favorable light. In particular you should describe all known bugs.

Submission

Use KAIST KLMS to submit your assignments. Your submission should be one gzipped tar file whose name is

YourStudentID_assign6.tar.gz

Your submission need to include the following files:

- Your source code files. (If you used `DynArray` ADT from precepts, then submit the `dynarray.h` and `dynarray.c` files as well.)
- A `makefile`. The first dependency rule should build your entire program. The `makefile` should maintain object (.o) files to allow for partial builds, and encode the dependencies among the files that comprise your program. As always, use the `gcc209` command to build.
- A `readme` file.
- NEW: observance of ethics document

Please do not submit `emacs` backup files, that is, files that end with '~'.

Grading

We will grade your work on quality from the user's point of view and from the programmer's point of view. From the user's point of view, your program has quality if it behaves as it should. The correct behavior of your program is defined by the previous sections of this assignment specification and by the given `sampleish` program. From the programmer's point of view, your program has quality if it is well styled and thereby simple to maintain. See the specifications of previous assignments for guidelines concerning style. Proper function-level and file-level modularity will be a prominent part of your grade. To encourage good coding practices, we will deduct points if `gcc209` generates warning messages. Remember that the Supplementary Information page lists detailed implementation requirements and recommendations.