CSCD 340
Homework 3
Writing your own shell

1 Assignment

Your boss requires a lean-and-mean UNIX shell, and none of the existing shells (sh, bash, ksh, csh, tcsh, zsh, ash, or even the infamous "adventure shell") seem to meet her requirements. So, you are stuck writing a shell almost from scratch! It does not need to support job management, environment variables, or most other standard shell features. However, it must be able to run programs, read the history and support file redirection and pipes. Your shell will be named **ssh**, meaning 'simple shell'.

Your file that contains main will be named `cscd340s14hw3.c`.

2 Shell Specification

This section outlines your boss's requirements for the shell that you are writing. It is fine to discuss the specification and its interpretation with other students, but do not share code. This specification is simply formalizing a simple version of the UNIX shell syntax that you have always used.

2.1 The Shell Language

Lexical structure

- The input to the shell is a sequence of **lines**. The shell must correctly handle lines of up to 100 characters. If a line containing more than 100 characters is submitted to the shell, it should print some kind of error message and then continue processing input at the start of the next line.
- Each line consists of **tokens**. Tokens are separated by one or more spaces. A line may contain as many tokens as can fit into 100 characters.
- There are two kinds of tokens: **operators** and **words**. The only operators are !, <, >, and | (pipe).
- Words consist of the characters A–Z, a–z, 0–9, dash, dot, forward slash, exclamation point and underscore. If a word in a line of input to the shell contains any character not in this set, then the shell should print an error message and then continue processing input at the start of the next line.
- The only legal input to the shell, other than lines consisting of valid tokens, is the end-of-file.

Parsing the shell language:

- Lines of input are divided into **token groups**. Each token group will result in the shell forking a new process and then execing a program.
- Every token group must begin with a word that is called the **command**. The words immediately following a command are called **arguments** and each argument belongs to the command it most closely follows. The order of arguments matters.
- It is permissible for the arguments in a token group to be followed by one or more **file redirections**. A file redirection consists of one of the operators < or > followed by a single word called the **filespec**. A file redirection containing the < operator is an **input file redirection** and a file redirection containing the > operator is an **output file redirection**.
- Token groups are separated by pipe operators. In other words, each valid line of shell input must begin and end with a valid token group, and the only place pipe operators are allowed is in between token groups.
- Valid token groups may be preceded by a pipe operator **or** they may contain an input file redirection, or neither, but not both. Similarly, valid token groups may be followed by a pipe operator **or** contain an output file redirection, or neither, but not both. You must handle up to two pipes with your shell.
- Lines of shell input that violate any of the parsing rules should cause the shell to print an error message and then move on to the next line of input.

Here are a few examples of shell language:

- `/usr/bin/emacs`

  is a legal line of input. It contains a single token group containing a single token, which is a command.

- `/usr/bin/emacs|`

  is not a legal line of input because the pipe character is not valid as part of a word (remember that tokens are always separated by one or more spaces).

- `/usr/bin/emacs |`

  is not a legal line of input because the pipe operator has to separate valid token groups. It is not legal for a pipe operator to be at the end of a line of input.

- `ls -l > foo`

  is a legal line of input containing a single token group. In order, the tokens are command, argument, operator, filespec.

- `ls -l > | foo2`

  is not a legal line of input because the output redirection operator is not followed by a word.

- `ls -l > foo1 | foo2`

  is not a legal line of input because the token group is both followed by a pipe and contains an output redirection.

- `> foo`

  is not a legal line of input because it does not begin with a word.

- `ls > foo%`

  is not a legal line of input because it contains an illegal character.

- `ls | grep -i cscd340 | sort`

  is a legal line of input. It contains 3 token groups. In order, the tokens in this line are: command, operator, command, argument, argument, operator, command.

- `cd ..`

  is a legal line of input containing a single token group. In order, the tokens are command, argument.

- `history`

  is a legal line of input containing a single token group. In order, the token is command.

- `!200`

  is a legal line of input containing a single token group. In order, the token is command.

- `!!`

  is a legal line of input containing a single token group. In order, the token is command.

2.2 Interpreting the Shell Language

Only legal lines of input (as defined in the previous section) should be interpreted — when the shell encounters an illegal line it prints an error message and then continues to the next line of input. (Again, your co-worker's parser handles the error messages.)

- Every **command** except the special command `exit` is to be interpreted as a UNIX executable to be `exec`'d.

- When the shell encounters either the command `exit` it terminates.

- The arguments to a command should be passed to the `exec` call in `argv`; `argv[0]` should always be the same as the string that is being exec'd, with actual arguments passed in slots one and higher.

- The > operator indicates that STDOUT of the associated command should be redirected to the UNIX file named by the filespec belonging to the operator. This file should be created if it does not exist, and the shell should report an error if the file cannot be created. Similarly,

the < operator indicates that STDIN of the associated command should be redirected from the UNIX file named by the filespec. The shell should report an error if this file cannot be opened for reading.

- The pipe (|) operator indicates that STDOUT of the preceding command should be redirected to STDIN of the following command.
- After interpreting a command, the shell should wait for **all** forked subprocesses to terminate before parsing the next line of input.

Here is the interpretation of the legal example commands from the previous section.

- `/usr/bin/emacs`

  The shell forks a new process and in it execs `/usr/bin/emacs`; the main shell process waits for emacs to exit before reading another line of input.

- `ls -l > foo`

  The shell opens the file `foo` for writing, forks a new process, redirects STDOUT of the new process to `foo`, and then execs `ls` with `-l` as argument one.

- `ls | grep -i cscd340 | sort`

  The shell sets up the pipes, forks 3 subprocesses, and then each subprocess execs the command for one of the 3 token groups.

Here is the interpretation of the legal example commands from the previous section that do not require fork and/or exec.

- `cd ..`

  The shell reads the `cd` command and executes the command to change the directory up one level

- `history`

  The shell displays the history which is stored in a file named **.ssh_history**. This file can store a maximum of 200 previous commands. It will function exactly like the history command in `bash`.

- `!200`

  The shell would re-execute the command numbered 200 that was stored in the history file. There may be a fork and exec involved.

- `!!`

  The shell would execute the last command. There may be a fork and exec.

  NOTE: The history command, !200, and !! should act just like the command does under `bash`. You will need to think about how `bash` behaves.

3 Getting Started

Be sure you understand this assignment before starting to write code. Here is a rough outline of steps you might take in solving it.

3.1 History

When the shell starts before you start parsing commands, you will need to open the history file and populate some data structure with the last 200 commands from the file.

3.2 Command-Line Parsing

Read the command from the user and determine if it is a valid command matching the semantics specified above.

3.3 Interpreting Shell Commands

Your shell command loop should properly interpret lines of input. You will need to use the `fork()` and `exec()` functions to do this. The shell process should wait for its children to complete by calling `waitpid()`.

```
read a line of input
parse the line

while (command != exit)
{

   for each command in the line {
      pid = fork();
      if (pid == 0) {
         do redirection stuff
         execve ( command, args , ...);
         oops, why did exec fail?
      } else {
         store pid somewhere
      }
   }
   for each command in the line
{
      waitpid (stored pid, &status);
      check return code placed in status;

   read a line of input
   parse the line
}
```

But remember to practice **incremental development**: get something working, add a small feature, test your code, and then move on to the next feature. For example, you should start out supporting shell input that contains a single command and no redirection. Then, add one of these features and then the other.

Your shell should return results similar to those returned by well-known shells like `bash`.

To support **I/O redirection**, modify the child process created by `fork()` by adding some code to open the input and output files specified on the command line. This should be done using the `open()` system call. Next, use the `dup2()` system call to replace the standard input or standard output streams with the appropriate file that was just opened. Finally, call `exec()` to run the program.

Pipes are a little trickier: you should use the `pipe()` system call to create a pair of pipe file descriptors **before** calling `fork()`. After the fork both processes will have access to both sides of the pipe. The reading process should close the write file descriptor, and the writing process should close the read file descriptor. At this point each process uses `dup2()` to copy the remaining pipe descriptor over STDIN or STDOUT as appropriate.

When you are done, bask in the glory of a working shell! You should now have a good operational understanding of the user-mode side of some of the most important UNIX system calls.

4 Other Odds and Ends

Getting Help: Since this is an upper division computer science course, you are expected to do your own research regarding the usage of various system calls, header files, and libraries. Information is readily available in the man pages, UNIX reference books, and on the web. For example, on any UNIX machine `man pipe` will give you information about the `pipe()` system call. Otherwise, do not hesitate to ask a question if you are unclear about how some part of the assignment is supposed to work.

Be sure to submit the source code files, your makefile, your history file, the executables, and anything you need to ensure your code compiles and runs. Your executable should be called `ssh`. Comment your code thoroughly and clearly.

You should know the name of the zip file by now. Ensure you include a `makefile` so we can easily compile and test your code