POSIX SHELL
Team: Pointers

Introduction

The fundamental UNIX interfaces have become so important that the IEEE has defined the POSIX standard to specify what essential interfaces constitute a fully POSIX-compliant operating system. In addition to various UNIX distributions, this includes most Linux distributions.
In Linux, a shell offers an interface for a Unix system that allows you to execute commands or utilities more easily. A shell collects an input from a user and executes a program according to that input. We can use a shell to perform various operations. In this project we have developed a working POSIX compatible shell with a subset of feature support of our default shell.

Description

To develop a working POSIX compatible shell.
Following features have been implemented:-

1) Basic shell commands (ls, echo, touch, mkdir, grep, pwd, cd, cat, head, tail, chmod, exit, history, clear, cp)
2) I/O redirections (IO redirection with '>>' and '>' will be done for one source and one destination only. Example: cat gfg.cpp > lol.txt)
3) Tab autocompletion
4) Generic piping support (For any number of pipes
   Example: cat gfg.cpp | head -10 | tail -4 | sort)
5) Piping with IO Redirection
   (Example:cat gfg.cpp | head -10 | tail -4 | sort > lol.txt)
6) History : Stores all the valid commands entered by user

7) Alarm: (Example: alarm k message : This command will remind us the message after k seconds)
8) Background : (For background command execution, '&' can be passed as last token of current command
   Example: ls -l &, tar -czf home.tar.gz . &)
9) Foreground :
10) Maintain a configuration file (.myrc) which our program reads on startup and sets the environment accordingly.
    This file contains alias and default applications that we use to open any file.
11) Provided support for the initialization variables like PATH, HOME USER, HOSTNAME, PS1
12) Association of "~" with the HOME variable.
13) Prompt look via PS1 is handled.
14) Export : The export command exports the variable to the environment of child processes
    Example: export v=3 (This command will export v to other child processes)

Solution Approach

1) Firstly, a command is entered which will be kept in a string .If string length is not null, then store the string in the ***history***.
2) Before going to the next step, we will check whether the input command contains any pipe character or not.
3) If there are pipes in the input string(i.e, the given input command), we are handling pipe commands in a different function.

   i) In this function, we are splitting the total input command into a ***two dimensional vector***, in which each row contains a simple built-in command.

   ii) First input command is completed by simply creating a ***fork*** and calling ***execvp*** for executing command , and its output goes to the input of next command or next row of the two dimensional vector.

   iii) And so on until the last row is completed.

iv) Our main output is the output of the last command executed in this function.

4) In the next step, a given input string would be ***parsed***, i.e, split the given command into individual strings and individual strings would be stored in a ***vector*** of string.
5) Otherwise, either it is asking for built-in commands or wrong commands entered.
6) If it is not asking for built-in commands , then simply print "***invalid command***".
7) Else we are executing system commands and libraries by forking a child and calling ***execvp***.
8) Step1 to step 7 are repeated until the exit command is entered.

Work Distribution

1) **Mahesh Dudhanale**
   - Set up input and output pipelines.
   - Configure tty (teletypes) and pttys to pass input and output properly.
   - Implement write redirection. (>)
   - Handle SIGTTIN and SIGTTOU.
   - Autocomplete with trie data structure.
   - Change prompt for normal and root user.
   - Support environment variables.
   - Implement history functionality.
2) **Sudipta Halder**
   - Set up a generic pipelining approach.
   - Handle input and output the newly forked processes.
   - Implement input redirection(<).
   - Implement alarm.
   - Implement alias functionality.
   - Handle SIGCHLD when child processes complete.
3) **SK Abukhoyer**
   - Implement redirection for append (>>).

- Change process group id for newly forked processes.
- Implement alarm.
- Implement PS1 prompt variable.
- Implement export() functionality.
- Handle quit() command gracefully.

4) **Pulkit Gupta**
- Handle background processes. (&)
- Handle bringing a process back to foreground.
- Setup .myrc file.
- Populate user, host and other fields in the prompt.
- Configure default applications for various file types.
- Input sanitization and null checks.

Problems Faced & Shortcomings

1) Alarm command implementation:-
   We faced a problem here for using ***sigaction*** structure, now this has been overcome easily by ***SIGALRM*** signals.
2) Alias command implementation:-
   Problem faced due to making it persistent in our ***POSIX*** shell.
3) Pipe implementation:-
   Memory was leaking when multiple pipes were used.
   Fixed it by using serial approach instead of depth first approach.
4) Pipe implementation:-
   Faced issue of unwanted spacing in the tokens while implementing piping. It was leading to invalid commands due to spaces. Fixed it by using proper trimming.
5) Invalid or incomplete commands:-
   Faced a lot of issues to handle invalid commands or incomplete commands. We Handled it giving proper nested if else cases so that the code does not break even if we give invalid or incomplete or broken commands.

6) File Handling:-

Faced issues while handling the files due to improper closing and different modes(read, write, append). Fixed it by using proper modes and closing files at proper place.

Learnings

- Process groups.
- Learned about differences between canonical and raw mode.
- Learned about tty, teletypes terminals.
- Learned about pseudo ttys, and shell emulators.
- Input sanitization and parsing.
- String manipulation using c default libraries.
- Data structure trie and its usage in autocompletion.
- Terminal escape sequences.
- Using escape sequences for controlling behavior of terminal.
- Signal Handling for below signals
    - SIGTTIN
    - SIGTTOU
    - SIGCHLD
    - SIGCONT
    - SIGINT
    - SIGKILL
    - SIGSTOP
    - SIGTERM
    - SIGPIPE
    - SIGIGN
    - SIG_DFL
- Usage of kill(pid, signal) & signal(signum, handler) for signal handling
- Background and foreground processing.
- Child process creation using fork and exec
- exec family of syscalls, execvp, execl, execlp, execle, execv, execvpe
- Pipes & file descriptors.

Results

- Learned a lot about how the OS communicates with user space programs using signals. The traps raised during execution are heard by the OS and then it refers to the interrupt table initialized during boot time to identify proper actions to execute.
- We can even provide our own handlers for messages from the OS.
- Learned about the /proc file system and used it for handling running processes in background and foreground.
- Learned about setting up process groups.

Finally, we were able to create a POSIX compatible shell which is usable and customizable to the user's needs.

Conclusion

Implementing POSIX specification gives us a lot of flexibility as to which systems can run our program.
For example, if pipes are not used then for the same task we would have to write a lot of code.
But using pipes and forking to create a new child simplifies the entire process.
Or even use of IO redirection is a great example, without it we human speed is a bottleneck for execution of programs. But IO redirection ensures automation of input. It is even useful when debugging files.
Learned a lot about how the OS communicates with user space programs using signals.
