Lab Assignment 6

Creating your own SHELL

Published: 28/9/15, Submission deadline: 5/10/2015, 11:59 AM

Assignment ID: **LA6**

1 Introduction

Shell is a user interface between user and the internal of OS. So a shell's job is to intercept user's command and then trigger system calls to ask OS to accomplish the user's tasks. Here is a list of available system calls in linux. A shell mainly consists of two parts: parsing user requests and accomplishing user request with system calls' help.

In this assignment, you will write your own command shell to gain further experience with some advanced programming techniques like process creation and control, file descriptors, signals and possibly pipes. We have provided a simple parser for you to use and you need to implement certain features in your shell.

2 Objectives

- Understand how a command shell works
- Understand Linux system insight, like IO stream, process etc.
- Get familiar with Linux system calls (recall system calls discussed in the last lab)
- Get familiar with C Language and its syntax, particuliarly the concept of pointers and dynamical memory allocation (most of you know, but refreshing those concepts will be very useful)

3 Required Features

Between this simple pseudocode and full featured shells, there are many optional features. Here are the features you should support in your shell. (Note: These are not necessarily in order of easy to hard, don't feel that you have to implement them in the order as they are.):

1. The prompt you print should indicate the current working directory. For example:

   The directory: /usr/foo/bar

   It may also indicate other things like machine name or username or any other information you would like.

   Try getcwd(char * buf, size_t size) .

2. You should allow the user to specify commands either by relative or absolute pathnames. To read in the command line, you may want to consider the readline function from the GNU readline library as it supports user editing of the command line.

   Linux TIP! Example of a relative command df, same command with absolute path: /bin/df. To find the absolute path of a command use which, which ls gives /bin/ls

   Try execvp it will search the path automatically for you. First argument should be pointer to command string and the second arguments should be a pointer to an array which contains the command string as arg[0] and the other arguments as arg[1] through arg[n].

3. You do not need to support setting of environment variables. However, you may find it useful to know about these variables especially PATH which is the list of directories in which to look for a specified executable. You may use execvp to have the system search the PATH inherited by your own shell from its parent.

   Linux TIP! Use printenv $PATH to see what's in your executable path

4. You should be able to redirect STDIN and STDOUT for the new processes by using < and >. For example, foo $< infile >$ outfile would create a new process to run foo and assign STDIN for the new process to infile and STDOUT for the new process to outfile. In many real shells it gets much more complicated than this (e.g. >> to append, > to overwrite, > & redirect STDERR and STDOUT, etc.)! You do not have to support I/O redirection for built-in commands (it shouldn't be too hard but you don't have to do it.)

   Note: one redirect in each direction is fine, not ls > foo < foo2

First open the file (use open or creat, open read only for infiles and creat writable for outfiles) and then use dup2. 0 is the filedescriptor for STDIN and 1 is the file descriptor for STDOUT. Examples: dup2 (fdFileYouOpened, fileno(stdin)) dup2 (fdFileYouOpened, fileno(stdout))

5. You should be able to place commands in the background with an & at the end of the command line. (You do not need to support moving processes between the foreground and the background (ex. bg and fg). You also do not need to support putting built-in commands in the background.) Try waitpid(pid, status, options).

6. You should maintain a history of commands previously issued. The number of previous commands recorded can be a compile time constant of at least 10. This could be a FIFO list. For example, if storing 10 commands, the latest issued 10 commands will be displayed. The oldest command is numbered as 1 and the latest command is 10.

   Linux TIP! The history command can show you the remembered commands by linux bash shell.

7. A user should be able to repeat a previously issued command by typing !number where number indicates which command to repeat. !-1 would mean to repeat the last command. !1 would mean repeat the command numbered 1 in the list of command returned by history.

   Note: You can probably think of better syntax for this, but I thought it was good to stay as close as possible to syntax used by real shells

8. A built-in command is one for which no new process is created but instead the functionality is build directly into the shell itself. You should support the following built-in commands: jobs, cd, history, exit, kill, and help.

   - *jobs* provide a numbered list of processes currently executing in the background.
     Try waitpid with WNOHANG option to check without blocking. You can either check when jobs called or each time through the main while loop.
   - *cd* should change the working directory.
   - *history* should print the list of previously executed commands. The list of commands should be numbered such that the numbers can be used with ! to indicate a command to repeat.
   - *exit* should terminate your shell process.
   - *kill* %num should terminate the process numbered in the list of background processes returned by jobs (by sending it a SIGKILL signal).
     Try kill (pid, SIGKILL) .
     Note: Usually kill num refers to the process with ProcessId num; while kill %num refers to the process in the jobs list with number num
   - *help* lists the available built-in commands and their syntax. (If you don't follow the syntax expected, then a help function would let the graders understand.)

9. If the user chooses to exit while there are background processes, notify the user that these background processes exist, do not exit and return to the command prompt. The user must kill the background processes before exiting.

10. You may assume that each item in the command string is seperated on either side by at least on space (e.g. prog ¿ outfile rather than prog>outfile).

4 Shell Code Base Structure

The first thing you will need is a parser. We are giving you a complete parser and the skeleton code of a shell here. The parser and shell skeleton code is a zip file that contains a Makefile that will get you started compiling on a UNIX/Linux platform.

- Download the file on your local machine;
- Uncompress the contents of SourceCodeShell.zip in a new directory shellWithParser;
- Simply cd shellWithParser to get into the shellWithParser directory;
- Type make to compile shell.c into the executable shell;
- Execute shell by typing ./shell on the command line;
- You can remove the executable by typing make clean.
- From experience using a command shell, you should be able to write basic pseudocode for a shell: (Note: The pseducode below uses the UNIX style fork/exec not the Windows style CreateProcess/WaitForSingleObject.):

```
int main (int argc, char **argv)
    {
        while (1){
        int childPid;
        char * cmdLine;

        printPrompt();

        cmdLine= readCommandLine(); //or GNU readline("");

            cmd = parseCommand(cmdLine);

            record command in history list (GNU readline history ?)

            if (isBuiltInCommand(cmd)){
                 executeBuiltInCommand(cmd);
                 } else {
                 childPid = fork();
                 if (childPid == 0){
                     executeCommand(cmd); //calls execvp
                     } else {
                       if (isBackgroundJob(cmd)){
                           record in list of background jobs
                           } else {
                             waitpid (childPid);
                             }
                            }
                    }
    }
   }
```

If you cannot get your code compiled, probably you are missing some library packages.
If you are in Ubuntu, try to type the two commands below to install the packages.

1. sudo apt-get install build-essential
2. sudo apt-get install lib64readline-gplv2-dev (For 64-bit)

   OR

   sudo apt-get install libreadline-gplv2-dev (For 32-bit)

5 Writing your Parser

For the Shell assignment you'll first need to start by writing a command line parser. The parser's job will be to break up the individual parts of the command line from the shell's input.

Remember! Your shell doesn't have to execute anything yet! It only has to parse up the command line and print out the information.

5.1 Basic Pseudocode for the Parser

```
/**
 *  parse.c
 */

  void init_info(parseInfo *p) {
      initialize parseInfo struct
  }
```

```
  /* parse a single command */
  void parse_command(char * command, struct commandType *comm) {

  }

  /*   parse commandline for space separated commands */
  parseInfo *parse (char *cmdline) {
    foreach cmd in cmdline {
      if (cmd == command) {
        parse_command(cmd, type)
      }
    }
  }

  /* prints out parse struct */
  void print_info (parseInfo *info) {
     foreach type in parseInfo {
       print "type_name: type"
     }
  }

  /*  free memory used in parseInfo */
  void free_info (parseInfo *info) {
     foreach memory_block in parseInfo
        free(memory_block)
  }
```

```
/**
 *  shell.c
 */
 int main(int argc, char **argv) {
    while(1) {
       cmdLine = readline(printPrompt());

       info  = parse(cmdLine);

       print_info(info);
    }

    free_info(info);
 }
```

5.2 Required Features in the Parser

The following features have been listed to build a minimal parser.

- Take input in from command line. You'll need to read and return the command line inside the readline function. You should print out any error in the command line length or other problems in the main or a separate function.
- Print argument list. Once you've read the command line you'll want to pass it to the parse function for parsing. This function is in the parse.c file and currently prints a string to show it has run. You'll need to parse up the space separated commands and build your parseInfo struct in here.
- Print if there is input redirection, this symbol: $'<'$. Once your parseInfo struct is complete you'll call print_info and it should run through the struct printing out if there is input redirection. The command would look like this: command < file

- Print if there is output redirection, this symbol: $'>'$. Similarly print_info should declare if there is output redirection. The command would look like this: command > output_file
- Print if in background, this symbol: '&'. Your print_info should also show weather or not the command was passed in the background. A command being run in the background would look like this: command &

All parsing of this information is done in the parse function, the print_info simply prints out what it finds in the parseInfo struct.

5.3 Running your shell and parser programs

Here is a text file of an example run of the parser shell. This should give you an idea of how the shell and parser should look and interact.

Here are the options: ———

```
prog: filename of program to execute
arg1:
arg2:
...
argn:
inpipe: no/filename of file
output: no/filename of file
background: yes/no
```

Here is an example: ———

```
%: filename of program to execute &
arg1: filename
arg2: of
arg3: program
arg4: to
arg5: execute
inpipe: no
output: no
background: yes
```

A zip file `SourceCodeShell` is available on LMS that contains a Makefile that will get you started compiling on a Linux platform. Follow the instructions given below to get started with this assignement. The below instructions are same as given in one of the earlier sections.

- Download the file on your local machine;
- Uncompress the contents of SourceCodeShell.zip in a new directory shellWithParser;
- Simply cd shellWithParser to get into the shellWithParser directory;
- Type make to compile shell.c into the executable shell;
- Execute shell by typing ./shell on the command line;
- You can remove the executable by typing make clean.
- From experience using a command shell, you should be able to write basic pseudocode for a shell: (Note: The pseducode below uses the UNIX style fork/exec not the Windows style CreateProcess/WaitForSingleObject.):
- You can either start your code from scratch, or feel free to use this example as a base for your shell.

6 Helpful Resources

The following system functions are likely to be helpful which we have used in our earlier lab exercises (consult the man pages for details):

fork, exec, execvp, wait, waitpid, signal

A few additions to the above list. manpages are the perfect places to look for more information. Also, see some of the example code given in SystemCallUsageExampleCode.zip

dup, getcwd, chdir, readline, kill

You may have to use a few of the following functions which I am sure you are already aware of.

strncmp, strlen, malloc, free, open, close, gets, fgets, getchar

Sample Output of Shell Commands

```
/usr/foo% jobs
No background jobs.
/usr/foo% ./a.out
Hello World
/usr/foo% ./a.out > hello.txt
/usr/foo% more hello.txt
Hello World
/usr/foo% cd bar
/usr/foo/bar% history
[1] jobs
[2] ./a.out
[3] ./a.out > hello.txt
[4] more hello.txt
[5] cd bar
[6] history
/usr/foo/bar% !-2
bar: No such file or directory
/usr/foo/bar% !2
./a.out: Command not found
/usr/foo/bar% !0
Event not found
/usr/foo/bar% !-50
Event not found
/usr/foo/bar% cd ..
/usr/foo% jobs
No jobs
/usr/foo% ./infiniteLoop &
/usr/foo% jobs
[1] ./infiniteLoop
/usr/foo% kill %1
[1] ./infiniteLoop terminated
/usr/foo% jobs
/usr/foo% help
cd <relative or absolute directory>
kill %jobNumber
history
jobs
exit
/usr/foo% /usr/foo/doIt < infile.txt > outfile.txt
/usr/foo% exit
```

Submission Instructions are given in the following.

You have to implement all the required features. Remember to include a README file to describe your assignment's status, which feature does not work yet, what extra features or work you have done. Anything helpful for grading is welcome. In addition, if you have anything more to add about the way you have implemented/build your code for the given exercise, it will be considered as a useful contribution.

Prepare a single zip file (including the README file) bearing your rollno (for e.g., IMT2013001). Submit your zip file as an attachment by sending an email with a following subject line (which is case-sensitive) to **submityourwork2ltj@gmail.com**

IMT2013-OS-LA6-05102015

In case your email's subject line is different then mentioned above, your assignment will not get submitted at the desired location on the server, and it will not be possible for me to carry out the assignment evaluation.