Creating a Shell or Command Interperter Program
CSCI411 Lab

Adapted from Linux Kernel Projects by Gary Nutt and Operating Systems by Tannenbaum

Exercise Goal: You will learn how to write a LINUX shell program. This will give you the opportunity to learn how new child processes are created and how parent processes can follow up on a child process.

Contents

Introduction

A *shell* or *command line interpreter* program, is a mechanism wit which each interactive user can send commands to the OS and by which the OS can respond to the user. Whenever a user has successfully logged in to the computer, the OS causes the user process assigned to the login port to execute a specific shell. The OS does not necessarily have a built-in windows interface. Instead, it assumes a simple character oriented interface in which the user types a string of characters (terminated by pressing the Enter or Return key) and the OS responds by typing lines of characters back to the screen

Once the shell has initialized its data structures and is ready to start work, it clears the display and prints a prompt in the first few character positions. Then the shell waits for the user to type a command terminated by **enter** or **return** character (in Linux, this is represented internally by the **NEWLINE** character '**\n**'). When the user enters a command, it is the shell's job to cause the OS to execute the command embedded in the command line.

Your assignment is to create a linux shell program that identifies itself in the prompt and handles a few specific commands. Any commands not recognized will be sent to the linux operating system, as written, for processing. The lab is divided into 4 parts as each one addresses a specific concept:

1. General shell processing
2. Redirecting I/O for a command
3. Creating a new process from the shell
4. Handling a system signal interrupt

Problem Statement 1: Create your own shell

Write a small program that loops reading a line from standard input and checks the first word of the input line. If the first word is one of the following internal commands (or aliases) perform the designated task. Otherwise use the standard system function to execute the line through the default system shell.

1. The command line prompt must be one of your design. It should clearly identify our shell program as different from the default shell program. A suggestion is that it contain your name. It can also contain the pathname of the current directory or whatever you wish.
2. The shell should keep a history file of all commands processed. Open the file when the shell is started and write to it when each command is read. When the shell program ends (either with the quit command or when ctl-c is pressed to exit the shell), the history file should be closed and printed.
3. The shell must support the following internal commands: A built-in command is one for which no new process is created but instead the functionality is built directly into the shell itself. You should support the following built-in commands:
    i. **myprocess –**
    ii. **allprocesses**
    iii. `chgd <directory>` -
    iv. `clr` - clear the screen
    v. `dir <directory>` -

i. **`environ`** - list all the environment strings
ii. **`quit`** - quit the shell
iii. **`help`** - display the user manual.

Problem Statement 2: File Redirection

Create a command called repeat that works like an *echo* command

`repeat <string>` - outputs the string to the console. If the user types a redirection operator '>; after the string, then redirect the string to the file specified.

For example **`repeat “hello world”`** results in “hello world” redisplayed on the screen. While **`repeat “hello world” > outfile.txt`** results in “hello world being written to the file outfile.txt

Problem Statement 3: Fork and Wait and Pipes

Create a command called '**`hiMom`**'. This command should be called the shell **fork**ing and **exec**ing the programs as its own child processes The child will then send a message to the parent. This message will be sent through pipes. Once sent, the child can exit.

The parent will listen for the message and print it out when it arrives. When the child exits, the parent will close the pipe.

Problem Statement 4: Signal handling

Create a signal handler that will listen for the termination interrupt signal. Signals are software interrupts delivered to a process by the operating system. Signals can also be issued by the operating system based on system or error conditions.

Your signal handler should print out that the shell exited due to a signal interrupt.

AND

Your signal handler should close the history file, print it to the screen

Attacking the Problem

Create your own shell

Consider the following steps needed for a shell to accomplish its job

1. Print a prompt

2. Read the command line. Note that cin will only read up to the first white space. You'll need to use getline to ensure the read is not terminated until the NEWLINE '\n' character is read.
3. Parse the command. It is acceptable to assume there is only one space between the command and any parameters. If you read the command in as a string, you can find the first blank space to delineate the command and parameter. (Hint: the string class has a find method…)
4. Execute the command: Options
   i. Process the command yourself
   ii. Use the system function to invoke the real linux command for this function
   iii. Read the following help for each of the commands you need to implement:

Command Implementation Help

vi. **myprocess -**Return the current process ID from the getpid function:

iv. **allprocesses** Return all current processes using the system function **ps.** use the **system** function to execute the real sytem function.

v. **`chgd <directory>`** - change the current working directory to **`<directory>`**.

If the **`<directory>`** argument is not present, no change is required. The current directory remains the working directory. If the directory does not exist an appropriate error should be reported. This command should also change the **`PWD`** environment variable. There is a function in unistd.h called chdir(char* path), that causes the directory named in the *path* argument to become the current working directory.

If the director is present, this will require the system command to run the cd command. You will have to pass the directory name.

vi. **`clr`** - clear the screen Use the **system** function to execute it

vii. **`dir <directory>`** - list the contents of directory **`<directory>`**

List the current directory contents (**ls -al <directory>**) - you will need to provide some command line parsing capability to extract the target directory for listing. Once you have built the replacement command line, use the **system** function to execute it.

viii. **`environ`** - list all the environment strings

List all the environment strings - the environment strings can be accessed from within a program by specifying the POSIX compliant environment list:
extern char **environ;
as a global variable.

environ is an array of pointers to the environment strings (in the format "***name=value***")
terminated with a NULL pointer.

There are two ways to approach this.

Approach one: A preferred way of passing the environment is by a built-in pointer:

```
extern char **environ;  // NULL terminated array of char *

   main(int argc, char *argv[])
   {
      char ** env=environ;
       while (*env)  /* while the pointer is not null
   cout << (*env++) << endl;
   }
```

example output:

```
GROUP=staff
HOME=/usr/user
LOGNAME=user
PATH=/bin:/usr/bin:usr/user/bin
PWD=/usr/user/work
SHELL=/bin/tcsh
USER=user
```

Approach two: is to use the **env** command:

system("env")..

ix. `quit` - quit the shell
Quit from the program with a zero return value. Close the history file and display it to the screen. Use the standard **exit** function

x. `help` - display the user manual. You will need to write this for the commands you support. For those commands defaulting to the system commands, you will need to build a command to call man for that command.

xi. For all other command line inputs, relay the command line to the parent shell for execution using the **system** function

File Redirection

A process, when created, has three default file identifiers: **stdin, stdout,** and **stderr.** If it reads from **stdin,** then the data that it receives will be directed from the keyboard to the **stdin** file descriptor. Similarly, data received from **stdout** and **stderr** are mapped to the terminal display.

The user can redefine **stdin or stdout** whenever a command is entered. If the user provides a filename argument to the command and precedes the file-name with a left angular brace character ,"<," then the shell will substitute the designated file for **stdin;** this is called redirecting the input from the designated file.

The user can redirect the *output* (for the execution of a single command) by preceding a filename with the right angular brace character, ">," character. For example, a command such as **kiowa>** WC < **main.c > program.stats** will create a child process to execute the wc command. Before it launches the command, however, it will redirect **stdin** so that it reads the input stream from the file **main.c** and redirect **stdout** so that it writes the output stream to the file **program.stats.**

Your job is to accept a '>' on repeat and send the data to the file specified, ex

Repeat abc > abc.txt

Will send the test 'abc' to a file named abc.txt

OE APPROACH: The shell can redirect I/O by manipulating the child process's file descriptors. A newly created child process inherits the open file descriptors of its parent, specifically the same keyboard for **stdin** and the terminal display for **stdout** and **stderr.** (This expands on why

concurrent processes read and write the same keyboard and display.) The shell can change the child's file descriptors so that it reads and writes streams to files rather than to the keyboard and display.
You can use the dup2() system call to redirect stdout/stdin

Note that you will have to drop down to a C function for file operations.

Forks and Pipe System Call

Create a child process using fork. The child and parent process then communicate via pipes.

The fork() function creates a new process by duplicating the calling process. The new process is referred to as the child process, while the calling process is the parent. The child process is a copy of the parent process with all data, open files, resources.

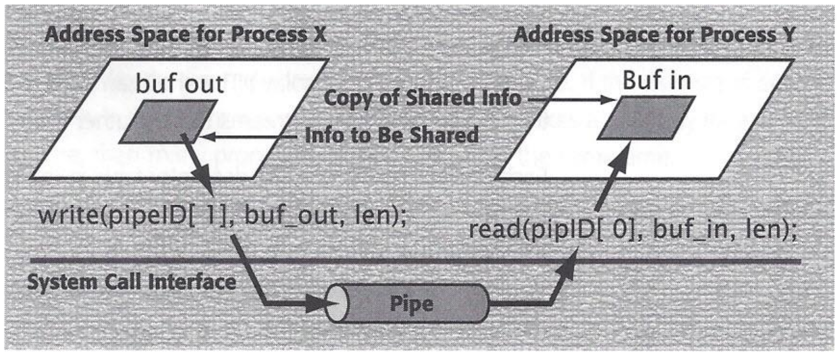
Inter-process Communication Call (IPC): Pipes

Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process

One process cannot read from the buffer until another has written to it

The *pipe* is an IPC mechanism in uniprocessor Linux and other versions of UNIX. Pipes are FIFO (first-in/first out) buffers designed with an API that resembles as closely as possible the file I/O interface. The pipe read and write ends can be used in most system calls in the same way as a file descriptor. By default, a pipe employs asynchronous send and blocking receive operations. As indicated in following figure, a process can send data by writing it into one end of the pipe and another can receive the data by reading the other end of the pipe.

For two or more processes to use pipes for IPC (interprocess communication), a common


Address Space for Process X
Address Space for Process Y
buf out
Buf in
Copy of Shared Info
Info to Be Shared
write(pipeID[1], buf_out, len);
read(pipID[0], buf_in, len);
System Call Interface
Pipe

ancestor of the processes must create the pipe prior to creating the processes. Because the fork command creates a child that contains a copy of the open file table (that is, the child has access to all of the files that the parent has already opened), the child inherits the pipes that the parent created. To use a pipe, it needs only to read and write the proper file descriptors.

Programming notes:

- pipe() is a system call that facilitates inter-process communication. It opens a **pipe**, which is an area of main memory that is treated as a "virtual file". The pipe can be used by the creating process, as well as all its child processes, for reading and writing.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The pipe() system call is passed a pointer to the beginning of an array of two integers.
- The system call places two integers into this array. These integers are the file descriptors of the first two available locations in the open file table.
    - pip[0] - the read end of the pipe - is a file descriptor used to read from the pipe
    - pip[1] - the write end of the pipe - is a file descriptor used to write to the pipe

Signal Processing

You are directed to yolinux tutorial for signal handling:

http://www.yolinux.com/TUTORIALS/C++Signals.html

Submission Requirements

- Submit your source code and makefile if necessary.

Grading Rubrics

Marking Criteria (100 pts total)

- Clear prompt ID 2 pts
- Performance of internal commands and aliases (40 points)

    vii. **myprocess –**
    viii. **allprocesses**
    ix. **`chgd <directory>`** -
    x. **`clr`** - clear the screen
    xi. **`dir <directory>`** -
    xii. **`environ`** - list all the environment strings
    xiii. **`quit`** - quit the shell
    xiv. **`help`** - display the user manual.

- File redirection **`repeat`** (8 points)
- Forking process (8 points)
- Correct use of pipes(8 points)
- Correct use of signal handling. (8 points)
- Correct implementation and display of history file (8 points)
- External command functionality (5 marks)
- Readability, suitability & maintainability of source code and instructions on compiling and executing (makefile, if needed) 12 marks)