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 ordinarily 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. If the human-computer interface is to be a graphical window interface, the window manager handles the shell tasks.

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. Linux machines usually include the machine name. For example, ACC.domanm2@aspen:. 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.

I/O 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.**

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

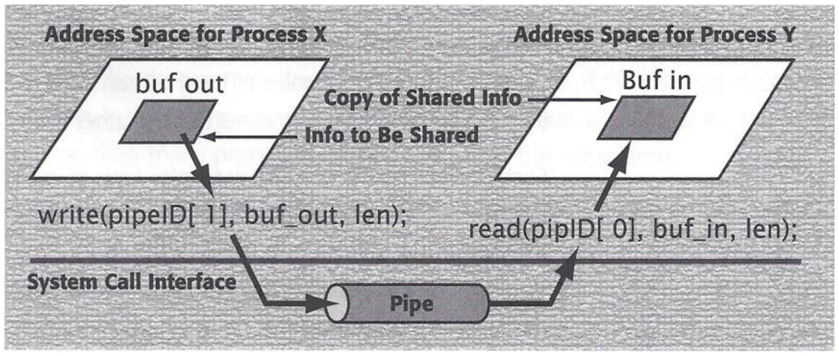
It is possible to have a series of processes arranged in a a pipeline, with a pipe between each pair of processes in the series.

Implementation: A pipe can be implemented as a 10k buffer in main memory with 2 pointers, one for the FROM process and one for TO process

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

Shell Pipes

The *pipe* is an IPC mechanism in uniprocessor Linux and other versions of UNIX. By default, a pipe employs asynchronous send and blocking receive operations. Optionally, the blocking receive operation may be changed to be a non-blocking receive (see Section 2.1.5 for details on invoking non-blocking read operations). Pipes are FIFO (first-in/first out) buffers designed with an API that resembles as closely as possible the file I/O interface. A pipe may contain a system-defined maximum number of bytes at any given time, usually 4KB. 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.


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

The kernel creates the pipe as a kemel FIFO data structure with two file identifiers. In this example code, pipeID[O] is a file pointer (an index into the process's open file table) to the read end of the pipe and pipeID[l] is file pointer to the write end of the pipe.
For two or more processes to use pipes for IPC (interprocess communication), a common 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.

For example, suppose that a parent creates a pipe. It then can create a child and communicate with it by using a code fragment such as the following.

```
pipe(pipeID);
if(forkO = = 0) { /* The child process * /

  read(pipeID[O], childBuf, len);
  /* Process the message in childBuf * /
} else { /* The parent process * /

  /* Send a message to the chi Id * /
  write(pipeID[l], msgToChild, len);
```

Pipes enable processes to copy information from one address space to another by using the UNIX file model. The pipe read and write ends can be used in most system calls in the same way as a file descriptor.

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. A suggestion is that it contain the pathname of the current directory.
2. 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 build directly into the shell itself. You should support the following built-in commands:

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

   ```
   getpid()
   ```

   i. **allprocesses** Return all current processes using the system function **ps**

   ```
   system("ps")
   ```

   ii. `cd <directory>` - change the current default directory to `<directory>`.

   If the `<directory>` argument is not present, report the current directory. If the directory does not exist an appropriate error should be reported. This command should also change the `PWD` environment variable.

   iii. `clr` - clear the screen

   system("clear")..

   iv. `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.

   v. `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. 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) printf("%s\n", *env++);
      }
   ```

   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
```

vi. `quit` - quit the shell
Quit from the program with a zero return value. Use the standard **exit** function

vii. `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.

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

Problem Statement 2: File Redirection

Create a command called repeat that works like an echo command

`repeat <string>` - use the echo command to repeat the string listed.

`Repeat <string> redirect`: list the contents of directory `<directory>`
If the user types a > after the string, then redirect the string to the file specified.

Problem Statement 3: Fork and Wait and Pipes

External Commands: All other command line input is interpreted as program invocation which should be done by the shell `fork`ing and `exec`ing the programs as its own child processes.

Create a command called 'hiMom'. This will result in a child process being forked. 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.

Attacking the Problem

Create your own shell

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

1. Print a prompt
2. Get 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.
4. Find the file to execute
5. Prepare the arguments to send to the command (argc, argv arguments).
6. Execute the command

File Redirection

ONE WAY: 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 _dup() or _dup2() system calls
- cout rdbuf() function to redirect the cout buffer)

ANOTHER WAY: In repeat command… you can just write to the file with standard IO

Pipe System Call

- 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.
- One process can write to this "virtual file" or pipe and another related process can read from it.
- 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 finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe. Recall that the open system call allocates only one position in the open file table.

Syntax in a C/C++ program:

```
#include <unistd.h>
int pip[2];
(void) pipe(pip);
```

With error checking:

```
#include <unistd.h>
int pip[2];
int result;
result = pipe(pip);
if (result == -1)
{
        perror("pipe");
        exit(1);
}
```

Programming notes:

- The pipe() system call is passed a pointer to the beginning of an array of two integers.
- It appears in C/C++ as if pipe() is passed the name of the array without any brackets.
- 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

The buffer size for the pipe is fixed. Once it is full all further writes are blocked. Pipes work on a first in first out basis.

Brief Example

- no error checking, no closing of pipe

```
#include <unistd.h>

int main()
{
       int pid, pip[2];
       char instring[20];

       pipe(pip);

       pid = fork();
       if (pid == 0)   /* child : sends message to parent*/
       {
              write(pip[1], "Hi Mom!", 7); /* write to the pipe */
       }
       else   /* parent : receives message from child */
       {
              read(pip[0], instring, 7); /* read from the pipe */
       }
 }
```

Submission Requirements

- Submit your source code (Shell script file) and makefile if necessary.
- Submit a lab report. This includes (from : http://www.columbia.edu/cu/biology/faculty/mowshowitz/howto_guide/lab_report.html)

It is extremely important that you understand the need for, and format of, a good report. Scientific work of any sort is useless unless its results can be communicated to others. Over the years a particular format, or general outline, has evolved for the preparation of scientific reports. It is this format which you should get accustomed to using.

First of all, a report should have a title. In addition, a scientific paper generally has five sections:

1) Introduction: Include a statement of the problem to be investigated, why the work was carried out, history and theoretical background of the problem, a brief statement of the general method of approach to the problem, and expected results.
    a. Terms (if necessary)
        i. Shell
        ii. Environment variables
    b. A brief overview of the purpose and description of the lab or programming assignment. It should be similar to a readme file: The description should contain enough detail for a beginner to UNIX to use the shell. For example, you should explain the concepts of the program environment.
2) Methods and materials: This section tells the reader how and with what the work was done.
    a. For this assignment the method would a list of the directories where you found the information and your code.
3) Results: While this is usually the meat of the report, for us, it is the output of your code. Please explain the output. Don't just cut and pasts the screen. I want to know that you understand what is being presented.
4) Discussion and conclusions: For this lab, this section is a bit contrived. But you can talk about why this information is useful.
5) References: if used.

Submission Items

Firstly, your source code will be assessed and marked. Commenting is definitely required in your source code. The user manual can be presented in a format of your choice (within the limitations of being displayable by a simple Text Editor). Again, the manual should contain enough detail for a beginner to UNIX to use the shell. For example, you should explain the concepts of i/o redirection, the program environment and pipes.

Grading Rubrics

Marking Criteria (100 marks total)

- Submission of required files only (5 marks)
- Warning free compilation and linking of executable with proper name (5 marks)
- Performance of internal commands and aliases (30 marks)
- External command functionality (10 marks)
- Readability, suitability & maintainability of source code and instructions on compiling and executing (makefile, if needed) (20 marks)
- User manual (30 marks)
    - Description of lab and concepts (20 marks)
    - Overall layout and display of understanding (10 marks)