Introduction to Developing a Shell Program

In this project, you will learn how to develop and implement a Linux/Unix shell. This will give you the opportunity to learn how child processes are created to perform large-grained work and how the parent process can follow up on a child process's work.

INTRODUCTION

A *shell,* or *command line interpreter* program, is a mechanism with 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 window-type 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 windows interface, then the software that implements the window manager subsumes the shell tasks that are the focus of this exercise. Thus, the character-oriented shell assumes a screen display with a fixed number of lines (usually 25) and a fixed number of characters (usually 80) per line.

Once the shell has initialized its data structures and is ready to start work, it clears the 25-line display and prints a prompt in the first few character positions on the first line. Linux systems are usually configured to include the machine name as part of the prompt.

For example, a Linux machine is named cis-linux1.cis.temple.edu, so the shell prints, as its prompt string:
cis-linux1>
or
bash>
depending on which shell I am using. The shell then waits for the user to type a command line in response to the prompt. The command line could be a string such as:
cis-linux1> ls –al

terminated with an **<ENTER>** or return character (in Linux, this character is represented internally by the NEWLINE character, **'\n'**). When the user enters a command line, the shell's job is to cause the OS to execute the command embedded in the command line.

Every shell has its own language syntax and semantics. In the standard Linux shell, *bash*, a command line has the form:

command [arg1] [arg2] ... [argN]

in which the first word is the command or program to be executed and the remaining words are arguments expected by that command. The number of arguments depends on which command is being executed. For example, the directory listing command may have no arguments-simply by the user's typing "**ls**" or it may have arguments prefixed by the negative "**-**" character, as in "**ls –al"**, where "**a**" and "**l**" are arguments. The command determines the syntax for the arguments, such as which of the arguments may be grouped (as for the "**a**" and "**l**" in the "**ls**" command), which arguments must be preceded by a "**-**" character, and whether the position of the argument is important.

Other commands use a different argument-passing syntax. For example, a g++ compiler command might look like:
cis-linux1> g++ -g -o deviation -S main.cpp inout.cpp –lmath
in which the arguments **g, o deviation, S, main.cpp, inout.cpp,** and **lmath** are all passed to the C++ compiler, **g++.**

The shell relies on an important convention to accomplish its task: The command for the command line is usually the name of a file that contains an executable program, for example, "**ls**" and "**g++**" (files stored in **/bin** on most UNIX-style machines). In a few cases, the command is not a filename but rather a command that is implemented within the shell. For example, "**cd**" (change directory) is usually implemented within the shell itself rather than in a

file in **/bin.** Because the vast majority of the commands are implemented in files, you can think of the command as actually being a filename in some directory on the machine. This means that the shell's job is to:

1. find the file
2. prepare the list of parameters for the command,
3. cause the command to be executed using the parameters.

Many shell programs are used with UNIX variants, including the original Bourne shell (**sh**), the C shell **(csh)** with its additional features over **sh**, the Korn shell, and the standard Linux shell **(bash).** All have followed a similar set of rules for command line syntax, though each has a superset of features.

Basic UNIX-Style Shell Operation

The Bourne shell is described in Ritchie and Thompson's original UNIX paper [Ritchie and Thompson, 1974]. As described in the previous subsection, the shell should accept a command line from the user, parse the command line, and then invoke the OS to run the specified command with the specified arguments. This command line is a request to execute the (command) program, i.e., the specified file that contains a program, including programs that the user wrote. Thus, a programmer can write an ordinary C program, compile it, and have the shell execute it just like it was a UNIX command.

For example, suppose that you write a C++ program in a file named **main.cpp** and then compile and execute it with shell commands such as:

cis-linux1> g++ -c –I. –o main.o main.cpp
cis-linux1> g++ -o main.o main
cis-linux1> ./main

For the first command line, the shell will find the **g++** command (the C++ compiler) in the **/bin** directory and then, when the **g++** command is executed, pass it the string **main.cpp.** The C++ compiler will translate the C++ program that is stored in main.cpp and write the resulting object file named **main.o** in the current directory.
The next command links the object file into an executable.
The third command is simply the name of the file to be executed, **main,** without any parameters. The shell finds the **main** file in the current directory (specified by ./) and then executes it.

Consider the following steps that a shell must take to accomplish its job.

1. Print a prompt.
   A default prompt string is available, sometimes hardcoded into the shell, for example the single character string **%, #,** or **>.** When the shell is started, it can look up the name of the machine on which it is running and prepend this string to the standard prompt character, for example a prompt string such as **cis-linux1>.** The shell also can be designed to print the current directory as part of the prompt, meaning that each time that the user types **cd** to change to a different directory, the prompt string is redefined. Once the prompt string is determined, the shell prints it to **stdout** whenever it is ready to accept a command line.
2. Get the command line.
   To get a command line, the shell performs a blocking keyboard input operation so that the process that executes the shell will be asleep until the user types a command line in response to the prompt. Once the user types the command line (and terminates it with a NEWLINE ('**\n**') character), the command line string is returned to the shell program.
3. Parse the command.
   The syntax for the command line is trivial. The parser begins at the left side of the command line and scans until it sees a whitespace character (such as space, tab, or NEWLINE). The first word is the command name, and subsequent words are the parameters.
4. Find the file.
   The shell provides a set of *environment variables* for each user[1]. These variables are first defined in the user's login file (for the bash shell this is **/home/<username>/.bashrc**) but they can be modified at any time by using the **set** command. The **PATH** environment variable (whose value can be viewed by typing

[1] Bash User Manual: http://www.gnu.org/software/bash/manual/bashref.html
Bash Guide for Beginners: http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html

"**echo $PATH"** at the bash shell) is an ordered list of absolute pathnames specifying where the shell should search for command files. If the **login** file has a line such as:
set path=(.:/bin:/usr/bin) for c shell (for bash, check out this link http://www.fnal.gov/docs/UNIX/unix_at_fermilab/htmldoc/rev1997/uatf-62.html)
then the shell will first look in the current directory (since the first full pathname is "."), then in **/bin**, and finally in **/usr/bin**. If no file with the same name as the command can be found in any of the specified directories, then the shell notifies the user that it is unable to find the command.

5. Prepare the parameters. The shell simply passes the parameters to the command as the argv array of pointers to strings. (on the top of stack)
6. Execute the command.
   The shell must execute the executable program in the specified file. UNIX shells have always been designed to protect the original process from crashing when it executes a program. That is, since a command can be *any* executable file, then the process that is executing the shell must protect itself in case the executable file contains a fatal error. Somehow, the shell wants to launch the executable so that even if the executable contains a fatal error (which destroys the process executing it), then the shell will remain unharmed.(by fork())

The Bourne shell uses multiple processes to accomplish this by using the UNIX-style system calls **fork(), execvp(),** and **wait().**

fork()
The **fork()** system call *creates a* new process that is a copy of the calling process, except that it has its own copy of the memory, its own process ID (with the correct relationships to other processes), and its own pointers to shared kernel entities such as file descriptors. After fork() has been called, *two* processes will execute the next statement after the fork() in their own address spaces: they are the parent and the child. If the call succeeds, then in the parent process fork() returns the process ID of the newly created child process and in the child process, fork() returns a zero value.

execvp()
The **execvp()** system call *changes* the program that a process is currently executing. It has the form:
execvp(char* path, char* argv[]);
The **path** argument is the pathname of a file that contains the new program to be executed. The **argv[]** array is a list of parameter strings. When a process encounters the **execvp()** system call, the next instruction it executes will be the one at the entry point of the new executable file. Thus the kernel performs a considerable amount of work in this system call. It must:

- find the new executable file,
- load the file into the address space currently being used by the calling process (overwriting and discarding the previous program),
- set the argv array and environment variables for the new program execution, and start the process executing at the new program's entry point.

Various versions of **execvp()** are available at the system call interface, differing in the way that parameters are specified (for example, some use a full pathname for the executable file and others do not).

wait()
The **wait()** system call is used by a process to block itself until the kernel signals the process to execute again, for example because one of its child processes has terminated. When the **wait()** call returns as a result of a child process's terminating, the status of the terminated child is returned as a parameter to the calling process.

Here is the code skeleton that a shell might use to execute a command when these three system calls are used:

```
// Child
if (fork() == 0)
{
execvp(fullpathname, argv);
}
// Parent
```

```
else
{
int status=0;
wait(&status);
cout << "Child exited with status of " << status << endl;
}
```

Putting a Process in the Background

In the normal paradigm for executing a command, the parent process creates a child process, starts it executing the command, and then waits until the child process terminates. If the "and" ("**&**") operator is used to terminate the command line, then the shell is expected to create the child process and start it executing on the designated command but not have the parent wait for the child to terminate. That is, the parent and the child will execute *concurrently.* While the child executes the command, the parent prints another prompt to **stdout** and waits for the user to enter another command line. If the user starts several commands, each terminated by an "**&**", and each takes a relatively long time to execute, then many processes can be running at the same time.

When a child process is created and starts executing as its own program, both the child and the parent expect their **stdin** stream to come from the user via the keyboard and for their **stdout** stream to be written to the character terminal display. Notice that if multiple child processes are running concurrently and all expect the keyboard to define their **stdin** stream, then the user will not know which child process will receive data on its **stdin** if data is typed to the keyboard. (do they all receive the same data?) Similarly, if any of the concurrent processes write characters to **stdout**, those characters will be written to the terminal display wherever the cursor happens to be positioned. The kernel makes no provision for giving each child process its own keyboard or terminal (unlike a windows system, which controls the multiplexing and demultiplexing through explicit user actions).

I/O Redirection

A process, when created, has three default file identifiers: **stdin**, **stdout**, and **stderr**. These three file identifiers correspond to the C++ objects **cin**, **cout**, and **cerr**. If the process reads from **stdin** (using **cin**) then the data that it receives will be directed from the keyboard to the **stdin** file descriptor. Similarly, data received from **stdout** (using **cout**) and **stderr** (using **cerr**) 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 filename with a "less than" 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

cis-linux1> we < main.cpp > program.stats

will create a child process to execute the "**we**" command. Before it launches the command, however, it will redirect **stdin** so that it reads the input stream from the file **main.cpp** and redirect **stdout** so that it writes the output stream to the file **program.stats**.

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 process, 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 to *files* rather than to the keyboard and display.

Each process has its own file descriptor table in the kernel. When the process is created, the first entry in this table, by convention, refers to the keyboard (stdin) and the second two refer to the terminal display.

Next, the C++ runtime environment and the kernel manage **stdin**, **stdout**, and **stderr** so that:

```
cin stdin 0 Keyboard
cout stdout 1 Terminal Display
```

cerr stderr 2 Terminal Display

If you want to perform I/O redirection, you need to 'connect stdin to a file' which can be read for input instead of the keyboard, or 'stdout to a file' which can accept output instead of the terminal display.

Let's consider **stdin** specifically. The key to getting I/O redirection to work for stdin is to replace the contents of the file descriptor entry for **stdin**, currently the keyboard, with the file descriptor entry for the file you want to use for input. To accomplish this, you need to access the file descriptors for **stdin** and the file you want to read from. You know how to get the file descriptor for **stdin**, that's just the number "0". But how do you get the file descriptor for the file you want to use for input? Unfortunately, you can't get the file descriptor directly from a C++ **ifstream** object. Instead you have to issue the following Linux system call (we are still using the example command line for the we command):

```
int newstdin = open("main.cpp",O_RDONLY);
```

The second argument to the open() call [O_RDONLY] is an "open this file for reading only" flag. Now newstdin has a file descriptor for the file "main.cpp".
Next you have to replace the contents of the **stdin** file descriptor with the "main.cpp" file descriptor. Use the following sequence of Linux system calls:

```
close(0);
dup(newstdin);
close(newstdin);
```

The close(0) call wipes out the contents of file descriptor table entry 0, which is the table entry for **stdin**. The dup(newstdin) call copies the contents of the newstdin file descriptor table entry to the first empty table entry in the file descriptor table. In this case, that will be entry 0. Now **stdin** will use the file "main.cpp" instead of the keyboard for input. There are now two file descriptors which are linked to "main.cpp" (one from the open() and one from the dup()). The final close(newstdin) cleans things up so that only the **stdin** file descriptor is linked to main.cpp.

A similar set of calls is used if you want to redirect **stdout** to an output file:

```
int newstdout = open("program.stats",O_WRONLY|O_CREAT,S_IRWXU|S_IRWXG|S_IRWXO);
close(1);
dup(newstdout);
close(newstdout);
```

Make sure that when you use this open call you include ALL of the flags listed here. Otherwise the redirection won't work.

Shell Pipes

The *pipe* is a common IPC (InterProcess Communication) mechanism in Linux and other versions of UNIX. By default, a pipe employs asynchronous send and blocking receive operations[2]. Optionally, the blocking receive operation may be changed to be a non-blocking receive. Pipes are FIFO (first-in/first out) buffers designed with an API that resembles as closely as possible a low level file I/O interface. A pipe may contain a system-defined maximum number of bytes at any given time, usually 4KB. 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.

Information Flow Through a Pipe

A pipe is represented in the kernel by a file descriptor, just like a file. A process that wants to create a pipe calls the kernel with a call of the following form:

```
int thePipe[2];
...
pipe(thePipe);
```

[2] A blocking receive means that the receive cannot complete until a new character is available to read. An asynchronous send means that a write can take place at any time and will be accepted.

The kernel creates the pipe as a kernel FIFO data structure with two file identifiers. In this example code, thePipe[0] is a file pointer (an index into the process's open file table) to the ***read*** end of the pipe and thePipe[1] 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:

```
int pid;
int thePipe[2];
pipe(thePipe);
pid = fork();
// Parent
if(pid > 0)
{
char message="Hello";
char messageLen=6;
write(thePipe[1], message, messageLen);
cout << "Parent sent: " << message << endl;
}
// Child
else if (pid == 0)
{
char message[6];
int messageLen=6;
read(thePipe[0], message, messageLen);
cout << "Child received: " << message << endl;
}
```

Pipes enable processes to copy information from one address space to another (one process 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 (for reading and writing a file). Further, the information written to and read from the pipe is a byte stream. UNIX pipes do not explicitly support messages, though two processes can establish their own protocol and data structures to provide structured messages. Also, library routines are available that can be used with a pipe to communicate via messages.

A process that does not intend to use a pipe end should close the end so that end-of-file (EOF) conditions can be detected. *A named pipe* can be used to allow unrelated processes to communicate with each other. Typically in pipes, the children inherit the pipe ends as open file descriptors. In named pipes, a process obtains a pipe end by using a string that is analogous to a filename but that is associated with a pipe. This allows any set of processes to exchange information by using *a public pipe* whose end names are filenames. When a process uses a named pipe, the pipe is a system-wide resource, potentially accessible by any process.

Just as files must be managed so that they are not inadvertently shared among many processes at one time, named pipes must be managed, by using low level file system commands.

Attacking the Problem

The introduction to this exercise generally describes how a shell behaves. It also implicitly provides a plan of attack, summarized here. This plan describes several debugging versions that you can use to begin the development, then apply this information to the other parts of the problem as required.

1. Organize the shell to initialize variables and then to perform an endless loop until the shell detects an EOF condition. When in the shell and you are reading from stdin, an EOF condition can be:
   - CTRL-D character typed
   - Keyword “exit” typed

   Develop a very simple version that prints the prompt character and then waits for the user to type a command. After it reads the command, it should print the command to stdout.

2. Refine your simple shell so that it *parses* the command typed by the user. The parser should do the following:
   - Convert each distinct string in the command into a C-String
   - Store the number of strings in the command in the integer variable argc
   - Store the C-Strings in an array of character pointers declared like this:

   char* argv[100];

   Set the first location in the array after your C-Strings to NULL. You are going to pass the argv array to the execvp() system call. This system call does NOT take argc as an argument, but the call still needs to detect the end of the argv array. To detect the end of the argv array, execvp looks for the first array entry that has the value of NULL.

3. In the next debug version, use argv[0] to find the executable file. In this version, simply print the filename.
   - Construct a simple version that can find only command files that are in the current directory.
   - Enhance your program so that it can find command files that are specified with an absolute pathname.
   - Enable your program to search directories according to the string that is stored in the shell PATH environment variable. You can access all of the environment variables by adding the following two lines before main() in your program:

   ```
   #include <unistd.h>
   extern char *environ[];
   ```
   The last item in the array is a NULL C-String.

4. Create a child process to execute the command.

```
#include <iostream>
using namespace std; //introduces namespace std
int main( void )
{
string theLine;
int argc;
char* argv[100];
// Initialization stuff
while (true)
{
cout << "MyShell> ";
getline(cin,theLine);
cout << "Command was: " << theLine << endl;
if ((theLine != "exit") && (!cin.eof()))
{
exit(0);
}
// Determine the command name, and construct the parameter list
// argv[0] - command name
// argv[1] - first parameter
// argv[2] - second parameter
// ...
// argv[N] - Nth parameter
// argv[N+1] - NULL to indicate end of parameter list
```

```
// Find the full path name for the file
// Launch the executable file with the specified parameters using
// the execvp command and the argv array.
}
```

Determining the Command Name and the Parameter List

A program might be run from your shell with the command:

```
MyShell> ./main foo 100
```

Before calling execvp() you need to set up the argv array so that, argv[0] will point to the C-String "main", argv[1] will point to "foo", argv[2] will point to the string "100", and argv[3] will be set to NULL to indicate the end of elements in the argv array. Your program then can interpret the first parameter (argv[1]) as, say, a filename, and the second parameter (argv[2]) as, say, an integer record count.

The shell would simply treat the first word on the command line as a filename and the remaining words as strings.

Finding the Full Pathname

The user might have provided a full pathname as the command name word or only a relative pathname that is to be bound according to the value of the PATH environment variable. A name that begins with a "/" "./" or "../" is an absolute pathname that can be used to launch the execution. Otherwise, your program must search each directory in the list specified by the PATH environment variable to find the relative pathname.

You can figure out if the file exists in a particular directory by trying to open the file. If the open fails, then the file does not exist.

Launching the Command

The final step in executing the command is to fork a child process to execute the specified command and then to cause the child to execute that command. The following code skeleton will accomplish this.

```
// Child
if (fork() == 0)
{
execvp(<fullpathnameofcommand>, argv);
}
// Parent
else
{
int status=0;
wait(&status);
cout << "Child exited with status of " << status << endl;
}
```

Attacking I/O Redirection and Pipes

1. Modify your shell program so that the user can redirect the **stdin** or **stdout** file descriptors by using the "<" and ">" characters as filename pre-fixes. For example:
   MyShell> ./main > out.txt
   MyShell> cat out.txt
   Hello World!
   MyShell>
   will place the output of the "Hello World" program into the file "out.txt" rather than sending it to the screen (which is **stdout**).

2. Also, allow your user to use the pipe operator, "|", to execute two processes concurrently, with **stdout** from the first process being redirected as the **stdin** for the second process. For example:
   MyShell> cat out.txt
   Hello World!

MyShell> cat out.txt | wc –l
MyShell>

The cat command sends the contents of the “out.txt” file to stdout. The “wc –l” command counts the number of characters, words, and lines typed into stdin. The two commands can be strung together using the pipe operator “|” so that the output of cat (sent to stdout) is connected to the input of wc (received from stdin).

You can design your program so that you only have to handle redirection OR pipes in one command line, but not both.

Don’t worry about mixing the “&”, redirection operators (“<” and “>”), and pipe operator “|” in the same command.
Concentrate on getting the redirection operators to work first. The key to getting these to work is to understand the I/O Redirection discussion in this document.

Watch out for file permission problems with the redirection operators. You need to bone up on the man pages for file() to get these flags correct. The default permissions for the file open for write meant subsequent redirects to the same file didn't work of because permission problems.

```
int newstdin = open(inputFileName.c_str(),O_RDONLY);
int newstdout = open(outputFileName.c_str(),O_WRONLY|O_CREAT,S_IRWXU|S_IRWXG|S_IRWXO);
```

To get command line pipes to work, re-read the Shell Pipes section carefully. Since a pipe is just like a file... there’s no reason why you couldn’t use I/O Redirection and pipes together!

1. Have the parent create a pipe.
2. Spawn a child.
3. Have the parent redirect stdout to the write portion of the pipe.
4. Have the parent execute the first command with excecvp().
5. Have the child redirect stdin to the read portion of the pipe.
6. Have the child execute the second command with execvp().
7. The output from the parent command should flow to the child command via the pipe and i/o redirection you performed on the pipe.

Watch out for (EOF) problems with pipes. I couldn't get the child process to terminate on an eof condition because the parent didn't close the read portion of the pipe. Make sure that for parent and child that you call the system call close() for any portion of a pipe you aren’t using.

General Hints

You should learn to use man pages to get info about important system calls. Sometimes the man page I wanted didn't come up when I typed man <keyword>. For example:
man wait
gave me lots of information about the bash shell, not the system call wait. But I could find out about wait by typing:
man 2 wait
I figured this out by looking at the documentation. If you see something like:
See also wait(2)
It means use the syntax above.
You should figure out how to get the error from a system call. If a system call fails, you can find out WHY it failed in and English friendly way by calling strerror() with the global variable errno. Add this to your code:

```
#include <errno.h>
#include <string.h>
extern int errno;
. . .
make a system call that fails
cout << strerror(errno) << endl;
```

See the document "Shell Project Specifics" for details on the project requirements and submission materials.

Some helpful books available in the Temple Library
C++ Recipes: A Problem-Solution Approach (Safari ebooks)
C++ Multithreading Cookbook (Safari ebooks)
Modern C++ Programming with Test-Driven Development (Safari ebooks)