Lab Exercise: A Shell Program

Write a C/C++ program that will act as a command line interpreter (or shell) for the Linux kernel. Your shell program should use the same style for running programs as the UNIX sh command. In particular, when the user types a line such as

`identifier [identifier [identifier]]`

your shell should parse the command line to build the argv data structure in the form expected in any C/UNIX main program. It should then search the directory system in the order specified by the PATH environment variable for a file with the same name as the first identifier (which may be a file name or a full pathname for a file). When the file is found, execute it with the optional parameter list specified by the other identifiers, just as sh does. Use the execv form of execve (that is, you are required to search the PATH variable to find the directory in which the command appears).

BACKGROUND

A command line interpreter, or shell, program is a mechanism with which each interactive user can issue commands to the OS and by which the OS can respond directly to the user. Whenever a user has successfully logged into the computer (a process will have been assigned to the user when they begin the login procedure), the OS causes the user process to execute a shell. The OS does not ordinarily have a built-in window interface—instead the OS assumes a simple character-oriented interface in which the user types a string of characters (terminated with the "enter" or "return" key), and in which the OS responds by typing lines of characters back to the screen. If the human-computer interface is to be a graphic window interface, then the software that implements the window manager subsumes the shell tasks on which we focus in 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.

Shell Behavior

In an interactive system, a shell program is executed by the login process after it has authenticated the user. Once the shell has initialized its data structures and is ready to start work, it clears the 25-line display, then 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. My Linux machine is named kiowa.cs.colorado.edu, so the shell that I use prints

`kiowa$`
as its prompt string. (My BSD workstation uses the C shell, so its prompt is "pawnee"). 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

    kiowa$ ls -al

terminated with an enter or return character (in Linux, the character is represented internally by the \n 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 argument_1 argument_2 ...

where the command to be executed is the first word in the command line 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 can be used with no arguments—simply by typing "ls", or it may have arguments prefaced by the "-" character, as in "ls -a1" where "a" and "l" are arguments. Other commands use their own syntax for accepting an argument; for example, a C compiler command might look like

    kiowa$ cc -g -o deviation -S main.c inout.c -lm

where the arguments "g", "o", "deviation", "S", "main.c", "inout.c", and "lm" are all being passed as parameters to the C compiler, "cc". That is, the command determines which of the arguments may be grouped (like the "a" and "l" in the ls command), which arguments must be preceded by a "-" symbol, whether or not the position of the argument is important, and so on.

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 cc are the names of files (stored in /bin on most UNIX-style machines). In a few cases, the command is not a file name, but is actually a command that is implemented within the shell; for example cd ("change directory") is usually implemented within the shell rather than in a file. Since the vast majority of the commands are implemented in files, think of the command as actually being a file name in some directory on the machine. This means that the shell's job is to find the file, to prepare the list of parameters for the command, then to cause the command to be executed using the parameters.

There is a long line of shell programs used with UNIX variants, including the original Bourne shell (sh), the C shell (csh) with its additional features over sh, the Korn shell, and so on, to the standard Linux shell (bash—meaning Bourne-Again shell). All these shells have followed a similar set of rules for command line syntax, though each has its own special features. The cmd.exe shell for Windows NT uses its own similar, but distinct, command language.
Basic UNIX-style Shell Design

A shell could use many different strategies to execute a user's computation. The basic approach used in modern shells is to create a new process (or thread) to execute any new computation. For example, if a user decides to compile a program, the process interacting with the user creates a new child process. The first process then directs the new child process to execute the compiler program. This same technique can be used by the initial process (the OS) when it decides to service a new interactive user in a timesharing environment. That is, when the user attempts to establish an interactive session, the OS treats this as a new computation. It awakens a previously created process for the login port or creates a new process to handle the interaction with the user.

This idea of creating a new process to execute a computation may seem like overkill, but it has a very important characteristic. When the original process decides to execute a new computation, it protects itself from any fatal errors that might arise during that execution. If it did not use a child process to execute the command, a chain of fatal errors could cause the initial process to fail, thus crashing the entire machine.

The UNIX paradigm for executing commands is illustrated in Figure 2.13. Here, the shell has prompted the user with the `%` character and the user has typed "`grep first f3`". This command means the shell should create a child process and cause it to execute the `grep` string search program with parameters `first` and `f3`. (The semantics of `grep` are that the first string is to be interpreted as a search pattern and the second string is a filename.)

The Bourne shell is described in Ritchie and Thompson's original UNIX paper [Ritchie and Thompson, 1973]. The Bourne shell and others accept a command line from the user, parse the command line, then invoke the OS to run the specified command with the specified arguments. When a user passes a command line to the shell, it is interpreted as a request to execute a program in the specified file—even if the file contains a program that the user wrote. That is, a programmer can write an ordinary C program, compile it, then have the shell execute it just like it was a normal
UNIX command. For example, you could write a C program in a file named main.c, then compile and execute it with shell commands like

```
kiowa$ cc main.c
kiowa$ a.out
```

The shell finds the `cc` command (the C compiler) in the `/bin` directory, then passes it the string "main.c" when it creates a child process to execute the `cc` program. The C compiler, by default, translates the C program that is stored in `main.c`, then writes the resulting executable program into a file named `a.out` in the current directory. In the second command, the command line is just the name of the file to be executed, `a.out` (without any parameters). The shell finds the `a.out` file in the current directory, then executes it.

Consider the detailed steps that a shell must take to accomplish its job:

- Printing a prompt. There is a default prompt string, sometimes hardcoded into the shell, e.g., the single character string "#", "%", "$" or other. When the shell is started, it can look up the name of the machine on which it is running, and prepend this string name to the standard prompt character, for example giving a prompt string such as "kiowa$". The shell can also be designed to print the current directory as part of the prompt, meaning that each time the user employs `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.

- Getting the command line. To get a command line, the shell performs a blocking read operation so that the process that executes the shell will be blocked until the user types a command line in response to the prompt. When the command has been provided by the user (and terminated with a `NEWLINE` character), the command line string is returned to the shell.

- Parsing 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 white space character (such as space, tab, or the end of the line). The first such word is treated as the command name, and subsequent words are treated as parameter strings.

- Finding the file. The shell provides a set of environment variables for each user—this variable is first defined in the user’s `.login` file, though it can be modified at any time with the `set` command. The `PATH` environment variable is an ordered list of absolute pathnames that specifies where the shell should search for command files. If the `.login` file has a line such as

```
set path=(./:bin:/usr/bin)
```

the shell will first look in the current directory (since the first pathname is "." for the current directory), then in `/bin`, and finally in `/usr/bin`. If there is no file with the same name as the command (from the command line) in any of the specified directories, the shell responds to the user that it is unable to find the command.
- Preparing the parameters. The shell simply passes the string parameters to the command as the `argv` array of pointers to strings.

- Executing the command. Finally the shell must execute the binary object 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, the process that is executing the shell must protect itself in case the executable file has a fatal error in it. Somewhere, the shell wants to "launch" the executable so that even if the executable contains a fatal error (which destroys the process executing it), the shell will remain unharmed. The Bourne shell uses multiple processes to accomplish what the UNIX-style system calls `fork`, `execve`, and `wait`.

- `fork`. This system call creates a new process which is a copy of the calling process except that it has its own process identification (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—the parent and the child. If the call succeeds in the parent process, `fork()` returns the process identification of the newly created child process, and in the child process, `fork()` returns a zero value.

- `execve`. This system call is used to change the program that the process is currently executing. It has the form

  `execve(char *path, char *argv[], char *envp[])`

  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, and the `envp` array is a list of environment variable strings and values that should be used when the process begins executing the new program. When a process encounters the `execve` system call, the next instruction it executes will be the one at the entry point of the new executable file. This means that the kernel performs a considerable amount of work in this system call: It must find the new executable file, load it 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, then start the process executing at the new program's entry point. There are various versions of `execve` available at the system call interface; they differ in the way parameters are specified (for example, some use a full pathname for the executable file, others do not).

- `wait`. This 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 terminating, the status of the terminated child is returned as a parameter to the calling process.

This discussion provides the bare minimum functionality for a shell. A production shell must also keep track of the current directory, implement several commands, handle background processes (invoked by terminating the command line with a "&")
character rather than a NEWLINE), handling redirection of stdin and stdout, supporting the pipe operation ("|"), providing a scripting language, and so on.

Attacking the Problem

There are several aspects of this lab exercise that might require more discussion. The following sections describe:
- How command names and parameter lists should be managed by the shell.
- How you should go about finding the location of the file that contains the command.
- How to execute the program for the command.
- How to organize your shell program.

Determining the Command Name and the Parameter List

You should recognize the argv name in the execve call from writing C programs in your introductory programming classes. That is, if you write a C program and you want the shell to pass parameters (from the command line) to your program, you declare the function prototype for your main program with a line like

```c
int main(int argc, char *argv[]);
```

The convention is that when your executable program (a.out) is executed, the shell will build an array of strings, argv, with argc entries in it. argv[0] points to a string with the command name ("a.out"), argv[1] points to a string specifying the first parameter, argv[2] points to a string specifying the second parameter, and so on. When your program is executed, it reads the argv array to get the strings, then applies whatever semantics it wants to interpret the strings. For example your program might be run with a command line of the form

```
a.out foo 100
```

so that when your program begins execution it will have argc set to 3, argv[0] will point to the string "a.out", argv[1] will point to "foo", and argv[2] will point to the string "100". Your program can then interpret the first parameter (argv[1]) as, say, a file name, 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 file name and the remaining words as strings.

Refer to the Code Skeleton section for the following discussion. After you have read the command line into a string, commandLine, you will need to parse it to populate the command_t fields (name, argc, and argv). The explanation in the Background section should be sufficient for you to design and implement code to parse the command line so that you have the name of the file containing the command in command->name, and (a pointer to) the array of pointers to parameter strings in command->argv.
Finding the Full Pathname

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

Launching the Command

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

```c
if(fork() == 0) {
    // This is the child
    // Execute in same environment as parent
    execvp(full_pathname, command->argv, 0);
} else {
    // This is the parent - wait for child to terminate
    wait(status);
}
```

Code Skeleton

The Background section provides a general, verbal description of how a shell behaves.
That description can be refined into the skeleton of a specific software solution with
the following pseudocode

```c
struct command_t {
    char *name;
    int argc;
    char *argv[];
    ...
);

int main () {
    ...
    struct command_t *command;
    ...
    // Shell initialization
    ...

    // Main loop
    while(TRUE) {
        // Print the prompt string
        ...
```
// Read the command line and parse it
... 
// Find the full pathname for the file
... 
// Create a process to execute the command
... 
// Parent waits until child finishes executing command
}

// Shell termination
... 
}

- First, implement the basic parts to print the prompt and read the command line. In this first-cut version, just print the command that is to be executed instead of trying to execute it. After you are able to determine the command name, complete the code to parse the command line and build the argv array, again just printing the results to stdout.

- Next, implement the command execution functionality. This means you will implement the fork/exec/wait code. Initially use execvp instead of execv (it is easier to use since you do not have to search the PATH directories).

- After you get the program working with execvp, implement the PATH search capability and switch from using execvp to execv.