Lab 1 - Command Shell

This assignment will give you a chance to warm up your C programming skills, see what the UNIX operating system offers in terms of system calls, and also let you implement one of the most important system applications: a command shell. Your command shell must be able to run programs on Linux platforms, and must support basic IO redirection and piping between commands.

Note that this assignment doesn't require you to use any system calls within Pintos; it is a completely stand-alone assignment. Pintos is limited on system calls and shell programs, so we use Linux for this Assignment.

Schedule

The labs have been divided into two slots, one slot for groups A1-27 and the other slot for groups B1-27. You should attend the labs during the time slot assigned to your group as per the schedule. **Your group will remain same for both Lab 1 and Lab 2**, so choose the group as per the time slots that suits you best. You are free to choose your lab group partner for the course. You **must** be in a group of 2 to work on and submit the lab assignments.

Getting Started

- Go through the Lab 1 Introduction slides.
- Join a project group inorder to have access to the **Lab 1 Preparation Test.**
- Pass the **Lab 1 Preparation Test** to gain access to the Lab 1 Assignment page. This is because you need to be familiar with some basic prerequisites before you can efficiently work on Lab 1. The **Useful Unix Functions** section below can come in handy to answer the test.
- The material for Lab 1 is available in Lab 1 Assignment: Shell Programming.
- Enjoy!

Submission

Before you submitting your assignment, you must fill-in the design document in your source tree under the name `pintos/src/shell/DESIGNDOC`. We recommend that you read the design document template before you start working on the project.

Make sure that your code has proper comments and indentation before you submit or even ask lab assistants for help. Trying to go through a badly written code wastes both your and our time.

Final submission should be an archive of the "shell" directory.

Useful UNIX Functions[1]

This section points out some of the standard functions that you might find really useful for this assignment. You are not required to use all of them; some will be necessary to implement the specified functionality, but others are simply only one option for the implementation.

The `man` Utility

You will need to use the UNIX file API and the UNIX process API for this assignment. However, there are too many functions for us to enumerate and describe all of them. Therefore you must become familiar with the man utility, if you aren't already. Running the command `man` command will display information about that command (called a "man page"), and specifically, `man unix_func` will display the man page for the UNIX function `unix_func()`. So, when you are looking at the UNIX functions needed to implement this assignment, use `man` to access detailed information about them.

The `man` program presents you with a simple page of text about the command or function you are interested in, and you can navigate the text using these commands:

- Down arrow goes forward one line
- Up arrow goes back one line
- "f" or Spacebar goes forward a page
- "b" goes back a page
- "G" goes to the end of the man page
- "g" goes to the start of the man page
- "q" exits `man`

One problem with `man` is that there are often commands and functions with the same name; the UNIX command "`open`" and the UNIX file API function "`open()`" are an example of this. To resolve situations like this, `man` collects keywords into groups called "sections"; when `man` is run, the section to use can also be specified as an argument to `man`. For example, all shell commands are in section "1". (You can see this when you run `man`; for example, when you run "`man ls`" you will see the text `LS(1)` at the top of the man page.) Standard UNIX APIs are usually in section 2, and standard C APIs are usually in section 3.

So, if you run "`man open`", you will see the documentation for the `open` command from section 1. However, if you run "`man 2 open`", you will see the description of the `open()` API call, along with what header file to include when you use it, and so forth.

You can often even look at some of the libraries of functions by using the name of the header file. For example, "`man string`" (or "`man 3 string`") will show you the functions available in `string.h`, and "`man stdio`" will show you the functions available in `stdio.h`.

Console I/O Functions

You can use `printf()` and `scanf()` (declared in `stdio.h`) for your input and output, although it is probably better to use `fgets()` to receive the command from the user. **Do not use `gets()`, ever!!!** You should always use `fgets(stdio, ...)` instead of `gets()` since it will allow you to specify the buffer length. Using `gets()` virtually guarantees that your program will contain buffer overflow exploits.

String Manipulation Functions

The C standard API includes many string manipulation functions for you to use in parsing commands. These functions are declared in the header file `string.h`. You can either use these functions, or you can analyze and process command strings directly.

`strchr()`
: Looks for a character in a string.

`strcmp()`
: Compares one string to another string.

`strcpy()`
: Copies a string into an existing buffer; does not perform allocation. Consider using `strlcpy()()` for safety.

`strdup()`
: Makes a copy of a string into a newly heap-allocated chunk of memory, which must later be `free()`d.

`strlen()`
: Returns the length of a string.

`strstr()`
: Looks for a substring in another string.

Process Management Functions

The `unistd.h` header file includes standard process management functions like forking a process and waiting for a process to terminate.

`getlogin()`
: Reports the username of the user that owns the process. This is useful for the command prompt.

`getcwd()`

Reports the current working directory of a process. This is also useful for the command prompt.

`chdir()`

Changes the current working directory of the process that calls it.

`fork()`

Forks the calling process into a parent and a child process.

`wait()`

Waits for a child process to terminate, and returns the status of the terminated process. Note that a process can only wait for its own children; it cannot wait e.g. for grandchildren or for other processes. This constrains how command-shells must start child processes for piped commands.

`execve()`
`execvp()`

The `execve()` function loads and runs a new program into the current process. However, this function doesn't search the path for the program, so you always have to specify the absolute path to the program to be run. However, there are a number of wrappers to the `execve()` function. One of these is `execlp()`, and it examines the path to find the program to run, if the command doesn't include an absolute path.

Be careful to read the man page on `execvp()` so that you satisfy all requirements of the argument array. (Note that once you have prepared your argument array, your call will be something like `execvp(argv[0], argv)`.)

Filesystem and Pipe Functions

`open()`

Opens a file, possibly creating and/or truncating it when it is opened, depending on the mode argument. If you use `open()` to create a file, you can specify 0 for the file-creation flags.

`creat()`

Creates a file (although why not use `open()` instead?).

`close()`

Closes a file descriptor.

`dup()`
`dup2()`

These functions allow a file descriptor to be duplicated. `dup2()` will be the useful function to you, since it allows you to specify the number of the new file descriptor to duplicate into. It is useful for both piping and redirection.

`pipe()`

Creates a pipe, and then returns the two file descriptors that can be used to interact with the pipe. This function can be used to pipe the output of one process into the input of another process:

1. The parent process creates a new pipe using `pipe()`
2. The parent process `fork()`s off the child process. Of course, this means that the parent and the child each have their own pair of read/write file-descriptors to the same pipe object.
3. The parent process closes the read-end of the pipe (since it will be outputting to the pipe), and the child process closes the write-end of the pipe (since it will be reading from the pipe).
4. The parent process uses `dup2()` to set the write-end of the pipe to be its standard output, and then closes the original write-end (to avoid leaking file descriptors).
5. Similarly, the child process uses `dup2()` to set the read-end of the pipe to be its standard input, and then closes the original read-end (to avoid leaking file descriptors).

[1] http://courses.cms.caltech.edu/cs124/pintos_2.html#SEC27