Assignment 1 - Writing your own Shell

CSC343/CSC443 Introduction to Operating Systems

September 7, 2017

This assignment must be completed by both CSC343 and CSC443 students.

The due date for this assignment is: September 20, 2017.

1 Preparation

You need to have a UNIX or GNU/Linux environment (for the rest of this handout the word UNIX will be a stand-in for UNIX and GNU/Linux) setup in order to complete the assignment. You have the option of:

1. Using your own Linux Box.
2. Using a UNIX or GNU/Linux VM installed on a different OS.
3. Using a UNIX or GNU/Linux VM installed on the cloud.
4. Any other UNIX or GNU/Linux environment you can think of, provided it is checked with me first.

2 Introduction

In this assignment you will be writing your own UNIX shell. A shell is typically used to allow users to run other programs in a friendly environment, often offering features such as command history and job control. Shells are also interpreters, running programs written using the shell's language (shell scripts).

Your shell will have the same basic functionality as the shells you are used to working in (e.g. `csh`, `tcsh`, `bash`), meaning it will allow the user to type in the name of an executable with arguments and then execute it. The shell will also provide a few built-in commands, such as `cd`, as well as some basic features such as file redirection. You don't need to implement anything not described on this page (e.g. history, job control, tab completion).

The purpose of this assignment is to introduce you to the C language, especially C string parsing and system calls. Therefore, we are restricting the library functions you are allowed to use in order to force you to call system calls directly.

3 Assignment

Your job is fairly simple: your shell must display a prompt and wait until the user types in a line of input. It must then do some simple text parsing on the input and take the appropriate action. For example, some input is passed on to built-in shell commands, while other inputs specify external programs to be executed by your shell.

Additionally, the command line may contain some special characters which will correspond to file redirection. The shell must set up the appropriate files to deal with this. As you know, users are far from perfect; your shell should have good error-checking.

3.1 The File System

Crucial to understanding how your shell will work is a working knowledge of the UNIX Virtual File System model.

In the VFS model, there is a root file system denoted as `/`, and zero or more mounted file systems which reside at mount points, like `/dev`. All file systems expose an internal structure of directories and files. Within the root file system there might be subdirectories such as `/bin`, `/home`, `/home/joeuser`, and `/home/joeuser/src`. There are also files within these directories, like `sh.c` and `README`. Mounted file systems behave just like root file systems, except that names of files within the file system are prefixed with the mount point.

The effect of all this is to abstract the particular way of accessing a file (the on-disk structure) from the fact that a file exists. In fact, some file systems might have no on-disk structure at all, and simply provide names that behave like files for other purposes. For instance, files in `/proc` are not really stored anywhere, they simply provide a file interface to kernel data structures.

3.2 Files, File Descriptors, Terminal I/O

To explain how files are represented in UNIX, consider the `open()` system call, which opens a file:

```
int open(const char *path, int oflag, mode_t mode);
```

For this system call, `path` refers to the relative (starting from the current directory) or absolute (starting from the root directory) path to the file to open, `oflag` is a combination of access modes and status flags, and `mode` gives the default permissions for the file if it must be created. The integer returned by `open()` is known as a file descriptor, which is sometimes abbreviated as "fd". File descriptors are simply indices into an in-kernel listing of the files available to the current process. Instead of making the kernel-level `file` structs available to user processes, files must be accessed through system calls like `read()` and `write()` which take file descriptors as arguments.

Each process inherits three standard file descriptors from its parent: input, output, and error. These are assigned file descriptors 0, 1, and 2, respectively.

The normal UNIX shell (e.g. `bash`) from which you begin your shell will set up these three file descriptors originally so that your shell is able to print to the terminal when it calls `write(1, "foo\n", 4)`. If you are having trouble remembering the standard file descriptor numbers, you can use the `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` macros defined in `unistd.h`.

A common question is how to wait for the user to input some text. You should note that `read()` will block until the user enters a new line or `Ctrl-D`.

3.3 Executing a Program

Executing a program in UNIX takes several steps. The `fork()` system call creates a new child process which is an exact replica of the parent, and begins execution at the point where the call to `fork()` returns. `fork()` returns 0 to the child process, and the child's process id (abbreviated `pid`) to the parent.

The `execve()` system call begins the execution of a new program, and if it succeeds it will never return. This is because once the execution of a new program begins, the entire memory space of the process which calls `execve()` will have been overwritten with the initial contents of the newly-started process. It is passed the `path` of the program to be executed, an argument vector `argv`, and a list of environment strings `envp`. `argv` and `envp` are always null-terminated arrays of pointers to strings (character arrays). The shell command "`/bin/echo Hello world!`" would have an `argv` that looks like this:

```
char *argv[4];
argv[0] = "/bin/echo";
argv[1] = "Hello";
argv[2] = "world!"
argv[3] = NULL;
```

The environment strings `envp` are a set of string of the form `"variable=value"`. Processes typically access these through the `libc` function `getenv`. You can run the command "`/usr/bin/env`" in a shell to list all the environment variables set in your shell. You are not required to keep track of environment variables for this assignment (just pass `NULL` for `envp`), but you may implement setting and un-setting them for extra credit.

Here is an example of forking and executing `ls`:

```
if (!fork()) {
    /* now in child process */
    char *argv[] = { "ls", NULL};
    char *envp[] = { NULL };
    execve(argv[0], argv, envp);

    /* we won't get here unless execve failed */
    if (errno == ENOENT) {
        fprintf(stderr, "sh: command not found: %s\n", argv
            [0]);
        exit(1);
    } else {
        fprintf(stderr, "sh: execution of %s failed: %s\n",
             argv[0], strerror(errno));
        exit(1);
```

```
    }
}
/* parent process continues to run code out here */
```

Note that your shell needs to wait for the executed command to finish before displaying a new prompt (unless you want to support background jobs). Look at the man page for the `wait()` system call for more information about how to do this.

3.4 Built-In Shell Commands

In addition to supporting the spawning of external programs, your shell will support a few built-in commands. When a built-in command is input, your shell should make the necessary system calls to handle the request and return control back to the user. The following is a list of the built-in commands your shell should provide.

- `cd` *`dir`*: Changes the current working directory.
- `ln` *`src dest`*: Makes a hard link to a file.
- `rm` *`file`*: Remove something from a directory.
- `exit`: Quit the shell.

Note that we are only looking for the basic behavior of these commands. You do not need to implement flags to these commands such as `rm -r` or `ln -s`. You also do not need to support multiple arguments to `rm` or multiple commands on a single line. Your shell should print out a descriptive error message if the user enters a malformed command.

3.5 UNIX System Calls for Built-In Functions

To implement the built-in commands, you will need to understand the functionality of several UNIX system calls. You can read the manual for these commands by running the shell command "`man 2 <syscall>`". It is highly recommended that you read all the man pages for these syscalls before even starting to implement built-in commands.

```
int open(const char *path, int oflag, mode_t mode);
int close(int fd);
int chdir(const char *path);
int link(const char *existing, const char *new);
int unlink(const char *path);
```

3.6 Prompt Format

While the contents of your shell's prompt are up to you, we ask that you implement a particular feature in order to make your shell easier to grade. Specifically,

you should surround the statement that prints your prompt with the C preprocessor directives `#ifndef NO_PROMPT` and `#endif`, which will cause the compiler to skip over the statement that prints your prompt when the `NO_PROMPT` macro is defined. For example, if you print your prompt with the statement `write(STDOUT_FILENO, "$ \n", 3);`, you would replace it with the following:

```
#ifndef NO_PROMPT
if (write(STDOUT_FILENO, "$ \n", 3) < 0) {
    /* handle a write error */
}
#endif
```

You can test this by running `make noprompt`, which will compile your code with the `NO_PROMPT` macro defined. Make sure you don't define the macro in your source file! Our grading scripts will fail on shells that don't implement this feature, so make sure you include it.

4 File Redirection

File redirection allows your shell to feed input to a user program from a file and direct its output into another file. Here is a summary from the `sh` man page.

> A command's input and output may be redirected using a special notation interpreted by the shell. (You do not need to support redirection for built-in commands.) The following may appear anywhere in a simple-command or may precede or follow a command and are not passed on as arguments to the invoked command.
>
> - "< *path*" - Use file *path* as standard input (file descriptor 0).
> - "> *path*" - Use file *path* as standard output (file descriptor 1). if the file does not exist, it is created; otherwise, it is truncated to zero length. (See the description of the `O_CREAT` and `O_TRUNC` flags in the `open(2)` man page.)
> - ">> *path*" - Use file *path* as standard output. If the file exists, output is appended to it; otherwise the file is created. (See the description of the `O_APPEND` flag in the `open(2)` man page.)

You must code your parser to support file redirection and appending with error checking. For example, if the shell fails to create the file to which output should be redirected, the shell must report this error and abort execution of the specified program. If multiple input or output redirections appear, this is also an error (it is illegal to redirect standard input twice, but it is perfectly legal to redirect both input and output). To understand the details of file redirection, it will be helpful to experiment with redirection in `bash` (some other shells have different redirection conventions).

5 Parsing the Command Line

A significant part of your implementation will most likely be the command line parsing. Redirection symbols may appear anywhere on the command line, and the file name appears as the next word after the redirection symbol. One algorithm for parsing the command line is as follows:

1. Scan through the line for redirection symbols, keeping track of the input and output file names if they exist. Check for errors such as multiple redirection or missing file names (i.e. a redirection token that is not followed by a file name) at this point.

2. Remove all traces of redirection from the command line (i.e. replace the relevant characters with blanks).

3. Split the line in to words. The first word will be the command, and each subsequent word will be an argument to the command.

Most symbols and words are separated by one or more spaces or tabs. Redirection characters may be separated from arguments by spaces or tabs. There need not be spaces or tabs before the first word on the line. Special characters such as control characters should be treated just like alphanumeric characters and should not crash your shell. You do not need to special case quotes (in most shells quotes would group several words into a single argument that contains white space).

Be very careful to check for error conditions at all stages of command line parsing. Since the shell is controlled by a user, it is possible to receive bizarre input. For example, your shell should be able to handle all these errors (as well as many others):

```
$ /bin/cat < foo < gub
ERROR - Can't have two input redirects on one line.
$ /bin/cat <
ERROR - No redirection file specified.
$ > gub
ERROR - No command. Make sure file gub is not overwritten.
$ < bar /bin/cat
OK - Redirection can appear anywhere in the input.
$ [TAB]/bin/ls <[TAB] foo
OK - Any amount of whitespace is acceptable.
$ /bin/bug -p1 -p2 foobar
OK - Make sure parameters are parsed correctly.
$ cat>bar<README
OK - No whitespace around redirection symbols is acceptable.
```

You will not be held responsible if your input buffer is not big enough to handle user input. Use a large buffer length (e.g. 1024) and assume that the user will

not enter more that that many characters. Note that in future assignments you will be responsible for handling similar error cases.

6 Use of Library Functions

You should use the `read()` and `write()` system calls to read and write from file descriptors 0, 1, and 2. Do not use C++ iostreams `cin`, `cout`, or `cerr` or C stdio (`fopen()`, `fread()` etc.). Part of the purpose of this assignment is to learn about system calls you'll be implementing later on in the semester. In order to avoid confusion, here is a list of the non-syscall functions allowed. Note that you can't use `strtok()`; part of the assignment is to practice C string manipulation.

assert	closdir	exit	free
isalnum	isalpha	iscntrl	isdigit
isgraph	islower	isprint	ispunct
isspace	isupper	isxdigit	malloc
memchr	memcmp	memcpy	memmove
memset	opendir	perror	readdir
s(n)printf	str(c)spn	str(n)cat	str(n)cmp
str(n)cpy	str(r)chr	strerror	strlen
strpbrk	strstr	tolower	toupper

Extra Credit (5 percent): Another important aspect of parsing the command line is knowing how to handle `Ctrl-D`. When a user enters some text on the command line followed by `Ctrl-D`, handle it as a newline. If the user does not enter anything but `Ctrl-D`, the shell should exit.

7 Groups and submission

This assignment is to be done individually or in groups of 2. One submission per group is sufficient.

Please include a Project Report which includes the following:

1. Project overview.
2. Assumptions made, if any.
3. Any Design decisions you had to make or issues you faced, in brief.
4. Summary (include any specific new learnings, if possible).
5. Source Code in Appendix.

In addition to submitting your source code and Report to D2L, you will have to demo your running solution to the instructor during Office Hours.

Alternatively, you can submit your source code via Git/GitHub. Remember having a public repo is a way of building a tech/dev portfolio.

I will setup a teleconferencing mechanism for Online students to demo.

8 Exchanging Information

I have setup a Slack instance so people can share information and discuss design and code issues. I encourage you to collaborate with each other but please do not copy blindly.

The link is: depaul OS Slack