Project Shell

Due: October 31, 2012

1 Introduction

In this 2-part 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 read commands from the user and execute those commands. The shell will also provide a few built-in commands, such as `cd`, as well as some basic features such as file redirection, pipes, and eventually job control (but that will come later).

The first part of the assignment serves as an exercise in C string parsing and an introduction to *system calls*.

2 Assignment

Your task 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 and pipes. 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.

2.1 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/jcarberr*, and */home/jcarberr/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.

2.2 Files, File Descriptors, Terminal I/O

- `int open(const char *path, int oflag, mode_t mode)`
- `int close(int fd)`
- `ssize_t read(int fd, void *buf, size_t count)`
- `ssize_t write(int fd, const void *buf, size_t count)`

You have previously read from and written to files using the `FILE` struct and functions such as `fopen()` and `fclose()`. This struct and these functions provide a high-level abstraction for how file input and output actually works, obscuring lower-level notions such as file descriptors and system calls. In this assignment, you will be performing input and output using exclusively file descriptors and system calls.

2.2.1 File Descriptors

At a lower level, file input and output is performed using *file descriptors*. A file descriptor is simply an integer which the operating system maps to a file location. The kernel maintains a list of file descriptors and their file mappings for each process. Consequently, processes do not directly access files using `FILE` structs but rather access files through the kernel first by using file descriptors and low-level system calls.

Processes inherit open files and their corresponding file descriptors from their parent process. As a result, proceses started from within a normal UNIX shell inherit three open files: `stdin`, `stdout`, and `stderr`, which are assigned file descriptors 0, 1, and 2[1] respectively. Since *your* shell will be run from within the system's built-in shell, it inherits these file descriptors; processes executed within your shell will then also inherit them. As a result, whenever your shell or any process executed within it writes characters to file descriptor 0 (the descriptor corresponding to `stdin`), those characters will appear in the terminal window.

2.2.2 `open()`

The `open()` system call opens a new file, located at the relative (starting from the process working directory) or absolute (starting from the root directory, /) path `path`, and returns a new file descriptor which maps to that file.

The other arguments for this system call are bit vectors which indicate how the file should be opened. In particular, `oflag` indicates both status flags and access modes, allowing the user to determine the behavior of the new file descriptor. `mode` is used to determine the default permissions of the file if it must be created.

File descriptors are opened lowest-first; that is, `open()` returns the lowest-numbered file descriptor currently not open for the calling process. On an error, `open()` returns -1.

[1] The header file *unistd.h* defines macros `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` which correspond to those file descriptors. This is useful for making code more readable.

2.2.3 `close()`

`int close(int fd)`

`close()` closes an open file descriptor, which allows it to be reopened and reused later in the life of the calling process. If no other file descriptors of the calling process map to the same file, any system resources associated with that file are freed. `close()` returns 0 on success and -1 on error.

2.2.4 `read()`

`ssize_t read(int fd, void *buf, size_t count)`

`read()` reads up to `count` bytes from the given file descriptor into the buffer pointed to by `buf`. It returns the number of characters read, and advances the file position by that many bytes, or returns -1 on an error.

`read()` blocks waiting for input; that is, if there is no data waiting to be read when the system call is invoked, it will wait until there is before returning.

2.2.5 `write()`

`ssize_t write(int fd, const void *buf, size_t count)`

`write()` writes up to `count` bytes from the buffer pointed to by `buf` to the given file descriptor. It returns the number of bytes successfully written, or -1 on an error.

2.3 Executing a Program

- `int fork()`
- `int execve(const char *filename, char *const argv[], char *const envp[])`

When a UNIX process executes another program, the process replaces itself with the new program entirely. As a result, in order to continue running, your shell must defer the task of executing another program to another process.

Since your shell knows nothing of the behavior of any process but its own, this must be done using the system call `fork()`, which creates a new "child" process which is an exact replica of the "parent" (the process which executed the system call). This child process 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. This allows the parent and child processes to branch off and execute different tasks, which happens to be exactly the behavior needed here.

To actually execute a process, use the system call `execve()`, which begins the execution of a new program. Since doing so replaces the entire process image with that of the new program, this function never returns if it is successful. `execve()` takes as arguments `filename`, which is the path to the program to be executed, a null-terminated[2] argument vector `argv`, and a null-terminated environment vector `envp` (which you need not worry about for this project). `filename` and `argv[0]` should be the same. As an example, 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;
```

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

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

    /* we won't get here unless execv 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 */
```

Your shell should wait for the executed command to finish before displaying a new prompt and reading further input. To do this, you can use the `wait()` system call, which you can read about by entering `man 2 wait` into a terminal.

2.4 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.

[2] An array for which `argv[argc]` is `NULL`, if `argc` is the number of entries in `argv`.

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`, multiple commands on a single line, or shortcut arguments such as `rm *` or `cd ~`. Your shell should print out a descriptive error message if the user enters a malformed command.

2.4.1 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);
```

2.5 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 `printf("$ \n");`, you would replace it with the following:

```
#ifndef NO_PROMPT
printf("$ \n");
#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.

2.6 Input and Output Redirection

Most modern shells allow the user to redirect the input and output of a program, either into a file or through a *pipe* (a form of interprocess communication). For example, bash terminals allow you to send a program input from a file using <, send output from a program to a file using > or >>, and chain the output of a program to the input of another using |. Your shell will be responsible for redirecting the input and output of a program but not for chaining multiple programs together.

2.6.1 Redirecting a File Descriptor

An open file descriptor is associated with a single file, and can't readily be "switched" to another, which poses a problem when trying to redirect input or output, which are already associated with file descriptors. However, there is an elegant solution to this problem: when a file is opened, the kernel returns the smallest file descriptor available, regardless of its traditional association. Thus, if we close file descriptor 1 and then open a file on disk, that file will be assigned file descriptor 1. Then, when our program writes to file descriptor 1, it will be writing to the file we've opened rather than `stdout` (which corresponds to file descriptor 1 by default). Nothing special must be done with the descriptor once it's been closed and re-opened.

The system call `dup2()`[3] encapsulates several of these steps.

```
int dup2(int oldfd, int newfd)
```

This system call points `oldfd` and `newfd` to the same file. It first closes `newfd`, and then duplicates (hence the name of the system call) `oldfd` with `newfd` as the copy. After a successful call to `dup2()`, `oldfd` and `newfd` refer to the same file; since only one of those file descriptors is needed, the other (which should be `oldfd` for `dup2()` to have been meaningful) can be safely closed.

2.6.2 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 will not be 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 does not exist, it is created; otherwise, output is appended to the end of it. (See the description of the `O_APPEND` flag in the `open(2)` man page.)

Your shell should support input and output redirection 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. Additionally, it is illegal to redirect input or output twice (although it is perfectly legal to redirect input and redirect output). You can experiment with I/O redirection in `bash`[4], which should serve as a model for the expected functionality of your shell.

[3] There is also a system call `dup()` and even `dup3()`. Each of these system calls has a similar use with slight differences; here, `dup2()` is the most useful.

[4] `bash` shells are the default shell of the terminals you open while logged in to the department. Other shells exist and may have different conventions regarding I/O redirection, among other things.

3 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 into 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. You shell must allow any number or combination of spaces and tabs when whitespace is required. Redirection characters may be separated from arguments by spaces or tabs, or may be immediately adjacent (touching the next word). There need not be spaces or tabs before the first word on the line, but their presence is not an error. 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 bytes) 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.

4 Use of Library Functions

You should use the `read()` and `write()` system calls to read and write from file descriptors `STDIN_FILENO` (a macro defined as 0), `STDOUT_FILENO` (1), and `STDERR_FILENO` (2), which correspond to the file streams for standard input, standard output, and standard error respectively. Do *not* use the I/O streams `stdin`, `stdout`, or `stderr`, as part of the purpose of this assignment is to learn about and use C system calls.

You may use any syscalls (functions with man pages that can be accessed using the shell command `man 2 <function>`) your heart desires. Do not use floating point numbers. If you have any questions about functions that you are able to use, please email the TAs.

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()`

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.

5 Grading

Your grade for the first part of the shell project will be determined by the following categories, in order of precedence:

- *Functionality*: your shell should produce correct output.
- *Correct use of system calls*: make sure you check the return value of each system call you use and handle errors accordingly. You must abide by the restrictions on library functions imposed in section 4 - you *will* be penalized for using disallowed functions.
- *Error checking*: your shell should perform error checking on its input and display appropriate, informative error messages when any error occurs. Error messages should be written to standard error rather than standard out.
- *Style*: your code will be evaluated for its style.

6 Handing In

To hand in the first part of your shell, run

```
cs033_handin shell_1
```

from your project working directory. You must at a minimum hand in all of your code, the Makefile used to compile your program (if you use one), and a README documenting the structure of your program, any bugs you have in your code, any extra features you added, and how to compile it (if you do not use a Makefile).

If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.