CS167 Programming Assignment 1: Shell

Assignment Out:	Sep. 5, 2007
Helpsession:	Sep. 11, 2007 (8:00 pm, Motorola Room, CIT 165)
Assignment Due:	Sep. 17, 2007 (11:59 pm)

1 Introduction

In this assignment you will be writing a real live 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., C shell (`csh/tcsh`) or `bash`) — meaning it will allow the user to type in the name of an executable, with arguments, and then execute this program. 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 in this handout (eg. history, job control, tab completion).

Your shell will be written and tested under Linux.

To run the TA shell demo, run `/course/cs167/demo/shell/shell`.

2 The 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.*

2.1 The Filesystem

Crucial to understanding how your shell will work is a working knowledge of the UNIX VFS (Virtual Filesystem) model.

In the VFS model, there is a root filesystem denoted as "`/`", and zero or more mounted filesystems which reside at mount points, like "`/dev`". All filesystems expose an internal structure of directories and files: within the root filesystem 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 filesystems behave just like root filesystems except that names of files within the filesystem 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 filesystems might have no on-disk structure at all, and simply provide names that behave like files for other purposes. For instance, objects in "`/dev`" are not real files; they simply provide a file-like interface.

2.2 Files, File Descriptors, Terminal I/O

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

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

`path` is an absolute (starting with '`/`') or relative pathname of the file; `oflag` is a combination of the access modes and status flags described in the `open(2)` man page; `mode` is the permission used if the file is created.

`open` returns an integer which is a *file descriptor* (sometimes referred to in shorthand as an "fd"). File descriptors are references to the open file in user mode. Instead of having access to the kernel-level `file` struct, users make system calls (like `read`) that take file descriptors.

Each process is initially "given" (set up by its parent) three standard file descriptors for input, output, and error: file descriptors 0, 1, and 2, respectively. You will start your shell from your regular UNIX shell (eg. `tcsh`), which will set these first three file descriptors, using code similar to this:

```
if (!fork())
{
        /* now in the child process */
        close(0);
        close(1);
        close(2);
        open("/dev/tty", O_RDONLY);
        open("/dev/tty", O_WRONLY);
        open("/dev/tty", O_WRONLY);

        execve("/bin/sh", 0, 0);
}
```

Since file descriptors are assigned in increasing numerical order, we are assured that fds 0, 1, and 2 are assigned to the corresponding terminal. This in turn means that to write to the terminal, all you need to do is call `write(1, "foo", 3);`. If you are having trouble remembering the file descriptor numbers for the standard streams, you can use `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` (after including `unistd.h`), which are macros for 0, 1, and 2, respectively.

2.3 Executing a program

Executing a program in UNIX takes place in several steps. The `fork(2)` system call creates a new child process which is a replica of the parent, and begins execution at the point that the call to `fork(2)` returns. `fork(2)` returns 0 to the child process, and the child's process id (also known as a `pid`) to the parent.

The `execve(2)` system call begins the execution of a new program and if it succeeds it will not return. It is passed the path of the the program to be executed, an argument list (`argv`), and a list of environment strings (`envp`). `argv` and `envp` are null-terminated arrays of pointers to strings (character arrays); the command

```
/bin/echo Hello world!
```

would have an `argv` that could be implemented 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 strings of the form "`variable=value`". Processes typically access these through the `libc` function `getenv(3)`; type `/usr/bin/env` in a shell to list all the environment variables set in your shell (and passed to `/usr/bin/env` via `execve(2)`). You are not required to keep track of environment variables for this assignment, (just pass `NULL` for `envp`[1]) but you may implement setting and unsetting them for extra credit.

Here is an example of forking and executing a new process:

```
if (!fork())
{
        /* now in the child process */
        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 {
```

[1]This may break some programs. Another possibility is to have your `main` function take not only the standard `argc` and `argv`, but also a third argument, `char *envp[]` which will get the environment of the program which executes your shell (probably the real shell you use in your terminal). You can then pass this directly along to `execve(2)`.

```
                fprintf(stderr, "sh: execution of %s failed: \%s\n",
                        argv[0], strerror(errno));
                exit(1);
        }
}
/* continue parent process execution */
```

2.4 Built-In Shell Commands

In addition to supporting the spawning of external programs, your shell will support a few internal (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 that your shell should provide:

- `cd` *dir*: change the current working directory.
- `ln` *src dest*: makes a hard link to a file.
- `rm` *file*: remove directory entry.
- `exit`: quit the shell.

Note that we are only looking for the default behavior on these commands; you don't need to implement `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 may print out an error message if the user enters a malformed command.

2.5 UNIX system calls for built-ins

To implement these commands, you will need to understand the functionality of several Linux system calls; you can (and should!) read the man pages in the lab. [2] It is highly recommended that you examine the man pages for these syscalls before beginning to implement the shell built-ins.

```
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] Type `man 2` *syscall* to read the man pages for system calls.

2.6 File redirection

From the `sh(1)` man page:

A command's input and output may be redirected using a special notation interpreted by the shell.[3] 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.

- `< word` Use file `word` as standard input (file descriptor 0).
- `> word` Use file `word` as standard output (file descriptor 1). If the file does not exist, it is created; otherwise, it is truncated to zero length.[4]
- `>> word` Use file word as standard output. If the file exists, output is appended to it[5]; otherwise, the file is created.

You must code your parser to support file redirection and concatenation, 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, that is also an error (ie. it is illegal to redirect standard input twice, it's perfectly legal to redirect input and output). To understand the details of file redirection, it will be helpful to experiment with redirection in your favorite UNIX shell[6] (whose error messages you may borrow for your own shell).

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:

- 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 redirect token that is not followed by a file name) at this point.
- Remove all traces of redirection from the command line (i.e. replace the relevant characters with blanks).
- Split up the line into words. The first word will be the command, and each subsequent word will be an argument to the command.

[3] You don't have to do redirection on builtin commands (not that it would be very useful to redirect the output of `ln` anyway...)

[4] See the description of the `O_CREAT` and `O_TRUNC` flags in the `open(2)` man page.

[5] See the description of the `O_APPEND` flag in the `open(2)` man page.

[6] In Linux's `csh` and `tcsh`, you can set the `noclobber` flag, causing redirecting to a file which exists to be an error. You may choose to implement this behavior or the truncation behavior described above, just tell us which you did in your README.

Most symbols and words are separated by one or more spaces or tabs. Redirection characters may be separated from arguments by zero or more 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.

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 of these errors (as well as many others):

`/bin/cat < foo < gub`	ERROR - can't have 2 input redirects on one line.
`/bin/cat <`	ERROR - no redirection file specified.
`> gub`	ERROR - no command specified. Make sure file `gub` is not overwritten.

Your shell also has to correctly handle bizarre yet correct input:

`< bar /bin/cat`	OK - redirection can appear anywhere in the input.
`[TAB]/bin/ls <[TAB] foo`	OK - any amount of whitespace is acceptable.
`/bin/gub -p1 -p2 foobar`	OK - make sure parameters are parsed correctly.
`cat>bar<README`	OK - no whitespace between redirection symbols is acceptable.

4 Use of external functions

You should use the `read(2)` and `write(2)` system calls to read and write from file descriptors 0, 1, and 2. Please don't use C++ iostreams *cin, cout* or *cerr* or C stdio (`fopen`, `fread`, etc); part of the purpose of this assignment is to learn about the system calls you'll be implementing later on in the semester.

You may use almost all the library calls defined in `string.h` to perform string or buffer manipulation in your shell code. You may also use any syscalls (functions with section 2 manpages) your little heart desires. Do not use floating point numbers (don't ask). If you have any questions about functions that you are able to use, please post on the newsgroup.

In order to avoid confusion, here is a list of the external functions allowed. Note that you can't use `strtok()`; part of the assignment is to practice C string manipulation.

```
open                    printf (and variants)       isalnum
close                   str(n)cpy                   isalpha
chdir                   str(n)cat                   iscntrl
link                    str(n)cmp                   isdigit
unlink                  str(r)chr                   islower
read                    str(c)spn                   isgraph
write                   strpbrk                     isprint
fork                    strstr                      ispunct
execve                  strlen                      isspace
wait                    strerror                    isupper
exit                    memcpy                      isxdigit
```

`malloc`	`memmove`	`tolower`
`free`	`memcmp`	`toupper`
`assert`	`memchr`	
`perror`	`memset`	

Another important aspect of parsing the command line is knowing how to handle Ctrl-D. When the 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 Code Exchange

To get started, copy the `/course/cs167/asgn/shell` directory and start hacking `sh.c`. To run your program, type `./sh`. Note the '`.`' before '`sh`'. To debug your program, type `gdb sh`, and at the `(gdb)` prompt, type `run`.

You need to write a simple README, documenting any bugs you have in your code, any extra features you added, and anything else you think we should know about your shell.

You should hand in your shell by running "`make clean; /course/cs167/bin/cs167_handin shell`" from the directory containing your code.

6 Extra credit

Here are some suggestions for extra credit:

- `PATH`, either as an environment variable, or just as a hard coded search path.
- `rm -r`
- environment variables (`setenv`, `printenv`, and passing the environment to processes via `execve`
- `pwd`, though you must use the `getdents(2)` system call (not `pwd`) for it to count. This is a bit tricky, but quite interesting. Yes, you must use `getdents`, not even `readdir`.