Lab 1b: Unix Programming: Implementing a Shell
COMPSCI 210: Introduction to Operating Systems
Due Date: Friday, 5 October 2012 at 11:59pm.

1 Introduction

Unix [5] embraces the philosophy:

Write programs that do one thing and do it well.
Write programs to work together.

The idea of a "shell" command interpreter is central to the Unix programming environment. As a command interpreter, a shell executes commands that you enter in response to its prompt in a terminal window, which is called the *controlling terminal*. Most commands to the shell simply name a program for the shell to execute in a child process, along with any arguments for the program or linkages of its input/output to pipes or files. Thus you can also use a shell as a scripting language that combines subprograms to perform more complex functions.

A shell can also read a list of commands from a file called a *shell script*. A shell script is just another kind of program, written in a programming language that is interpreted by the shell. Modern shells support programming language features including variables and control structures like looping and conditional execution. Thus shell scripting is akin to programming in modern interpreted languages such as Perl, Python, and Ruby. The shell provides an interactive environment in which a user may compose new command programs and execute them immediately, without a need to compile them in advance.

2 The Devil Shell: `dsh`

In this lab you use Unix system calls to implement a basic shell. We call it a *devil shell*, abbreviated as `dsh`.

A `dsh` supports basic shell features: it spawns child processes, directs its children to execute external programs named in commands, passes arguments into programs, redirects standard input/output for child processes, chains processes using pipes, and monitors the progress of its children.

The `dsh` also groups processes for job control. For our purposes, a *job* is a group of one or more commands that the shell executes together and treats as a unit. When the shell is being used interactively (i.e., there is a controlling terminal), then at most one job is in the "foreground" at any given time. If a job is in the *foreground* then any input on the controlling terminal is directed to the foreground job, rather than to the shell itself, and the shell waits for the foreground job to quit (exit) or pause (stop). Jobs in the *background* execute concurrently with the shell, while the shell accepts and processes new commands rather than waiting for a background job to complete. (We might also say that background jobs run "asynchronously" or "in parallel with" the shell. These terms have the same meaning.)

The `dsh` takes many shortcuts to make this lab simpler, even simplistic. For example, although `dsh` can run as a script interpreter, it does not support variables or control structures. The shell ignores common environment variables like `$PATH` and `$HOME`. This ignorance makes `dsh` cumbersome to use: any part of a command string that names a file or directory must be a fully qualified pathname relative to the root directory (if the pathname starts with "`/`"), or relative to the shell's current directory (if it does not). And you must enter the full name: `dsh` does not support auto-completion.

Also, `dsh` avoids the use of Unix *signals*. The Unix signal mechanism was an early abstraction for event handling. It was botched in its initial conception and then reworked in piecemeal fashion over decades, resulting in multiple versions and incompatibilities. Signals are central to Unix, but we are glossing over them in this course. If you find yourself trying to understand signals, then please talk to us first. We recommend the CS:APP book if you want to know how real shells use signals.

The lack of support for signals means that `dsh` will not notice if one of its background jobs stops or exits. Instead, the shell has a built-in command called `jobs` to check and report the status of each of its children.

Although we try to ignore them (literally), signals are intricately wound into the user experience of any shell. If there is a foreground job, you can cause it to exit by typing `ctrl-c` on the controlling terminal. You can cause it to pause (stop) by typing `ctrl-z` on the controlling terminal. These special keychords cause the kernel (tty driver) to send a signal to any processes bound to the controlling terminal (i.e., the foreground job). The default behavior for these signals is to kill or stop the receiving process. If there is no foreground job, then the `dsh` itself has control of the terminal: the signals are directed to `dsh` which may cause it to exit or stop.

The shell itself also exits if it reads an *end-of-file* (EOF) on its input. The shell reads an EOF when it reaches the end of a command script, or if you type `ctrl-d` to it on the controlling terminal. EOF is not a signal, but just a marker for the end of a file or stream.

3 Inputs

The shell reads command lines from its standard input and interprets them. It continues until it is killed or there are no more commands (EOF). We provide a command line parser to save you the work of writing one yourself, or at least to give an example of how to do it.

The shell prints a prompt of your choosing (e.g., `dsh-277$`) before reading each input line. This prompt should include the `dsh` processID.

If an input line starts with the special character "`#`", then the line is a *comment*: the entire line is ignored. Empty lines are also ignored. Any other line is a *command line*. If a command line contains a "`#`" character, then the remainder of the line is a comment and is ignored: the "`#`" and any succeeding characters are not part of the command line.

Upon receiving a command line, the shell interprets the line as a sequence of one or more commands, separated by the special characters "`;`" or "`|`". If a command is followed by a "`;`" then the shell completes its processing of the command before moving to the next command in the sequence (the successor), if there is one. The special character "`|`" indicates a pipeline: if a command is followed by a "`|`" then the shell arranges for its standard output (stdout) to pass through a pipe to the standard input (stdin) of its successor. The command and its successor are grouped in the same job. If there is no successor then the command is malformed.

Each command is a sequence of substrings (tokens) separated by blank space. The first token names the command: it is either the name of a built-in command or the name of an external program (an executable file) to run. The built-in commands are discussed below.

In addition, the last non-blank character of a command may be an "`&`". The meaning of "`&`" is simply that the command executes in the background, if it is an external program. If the command is not an external program, i.e., it is a built-in command, then any "`&`" is ignored. If a command has no "`&`" then it runs in the foreground.

The remaining tokens in a command specify arguments for the command. The `dsh` has limited support for *input/output redirection*:

- If an argument begins with the special character "`<`" then the shell arranges for the command to read its stdin from a file: the rest of that argument is the name of the input file.
- If an argument begins with the special character "`>`" then the shell arranges for the command to write its stdout to a file: the rest of that argument is the name of the output file.

All other arguments for a command are *argument strings*. Argument strings may contain spaces if they use quote marks, and they may contain special characters if escaped with "
": the syntax rules for an argument string are standard for other Unix shells.

The shell passes the argument strings to the command in the order in which they appear on the command line. The command interprets its argument strings in a command-specific fashion. External programs receive these arguments through an array of strings called the *argv array*. The *argv* array has *argc* elements, where $argc \geq 1$. The first string in *argv* is the name of the command. The remaining strings in *argv* are the argument strings as they appear on the command line.

Here is an example:

```
$./dsh  #This is a comment; executing dsh shell from a default
   shell
dsh$ /bin/ls -altr  #command ls with arguments -altr
-rw-r--r-- 1 cps210 cps210  9621 2012-01-02 20:52 dsh.c
drwxr-xr-x 5 cps210 cps210  4096 2012-09-08 20:46 ..
-rw-r--r-- 1 cps210 cps210  4304 2012-09-08 23:17 cmdList
drwxr-xr-x 5 cps210 cps210  4096 2012-09-10 09:04 .
drwx------ 2 cps210 cps210  4096 2012-09-10 15:49 test
-rwxr-xr-x 1 cps210 cps210  7901 2012-09-10 22:01 dsh
dsh$ <ctrl-d> #prompt returns to the dsh shell after execution; <
   ctrl-d>
exits the shell

$cat cmdList    #cmdList is a file with a list of commands
/bin/ls -ltr
/bin/echo "hello world"

$./dsh < cmdList  #processing the batch of commands from a file
-rw-r--r-- 1 cps210 cps210  9621 2012-01-02 20:52 dsh.c
drwxr-xr-x 5 cps210 cps210  4096 2012-09-08 20:46 ..
-rw-r--r-- 1 cps210 cps210  4304 2012-09-08 23:17 cmdList
drwxr-xr-x 5 cps210 cps210  4096 2012-09-10 09:04 .
drwx------ 2 cps210 cps210  4096 2012-09-10 15:49 test
-rwxr-xr-x 1 cps210 cps210  7901 2012-09-10 22:01 dsh
hello world
$     #prompt returns to the built-in shell
```

3.1 Built-in commands

The shell executes built-in commands in its own context, without spawning a child process or job. The devil shell has the following built-in commands:

- `jobs`. Output the command strings and status for all jobs in a numbered sequence. Each job has a *job number*, which does not change during the job's lifetime. A job number j is an integer with $0 < j < 20$. Each new job is assigned the lowest job number that is not currently in use. If `jobs` detects that a job has exited, then it outputs any exit status for the job, frees the job number, and deletes any shell state relating to the job.
- `fg`. Continue a named job in the foreground. The job is given by a single argument: the job number.
- `bg`. Continue a named job in the background. The job is given by a single argument: the job number.
- `cd`. Change the shell's current directory. The target is given by a single argument: a pathname.

For example, the `jobs` command shows the state of all processes in the background.

```
$./dsh
dsh$ /bin/sleep 30 & #executed by spawning a child process
dsh$ /bin/sleep 5 & #executed by spawning another child process
dsh$ jobs      #executed in the current context of the dsh shell
[1]-  Running                 sleep 30 &
[2]+  Running                 sleep 5 &
dsh$ jobs      #After 5 seconds
[1]-  Running                 sleep 30 &
[2]+  Done                    sleep 5
dsh$
```

3.2 Managing processes

If a command is not a built-in, then its first token names an external program to execute. The program executes in a child process, grouped with any other child processes that are part of the same job. The essence of this lab is to use the basic Unix system calls to manage processes and their execution.

3.3 Fork

We provide a procedure called `spawn` that uses the `fork` system call to create a child process for a job. The `spawn` routine also attends to a few other details.

Each job has exactly one *process group* containing all processes in the job. Each process group has exactly one process that is its *leader*. If a job has only one process then that process is the leader of the job's process group.

A job has multiple processes if it is a pipeline, i.e., a sequence of commands linked by pipes. In that case the first process in the sequence is the leader of the process group.

If a job runs in the foreground, then its process group is bound to the controlling terminal. If a process group is bound to the controlling terminal, then it can read keyboard input and generate output

to the terminal. The kernel also directs any signals generated from the keyboard (`ctrl-c` and `ctrl-z`) to all processes in the group.

If a job running in the background attempts to perform any I/O on the terminal, then the kernel sends a signal whose default action is to pause (stop) the process.

The `spawn` routine shows how to use *fork* and other Unix system calls to set the process group of a process and bind a process group to the controlling terminal.

```
/* Spawning a process with job control. fg is true if the
 * newly-created process is to be placed in the foreground.
 * (This implicitly puts the calling process in the background,
 * so watch out for tty I/O after doing this.) pgrp is -1 to
 * create a new job, in which case the returned pid is also the
 * pgrp of the new job.  Else pgrp specifies an existing job's
 * pgrp: this feature is used to start the second or
 * subsequent processes in a pipeline.
 */

pid_t spawn_job(pid_t pgrp, bool fg) {

      int ctty = -1;
      pid_t pid;

      switch (pid = fork()) {

        case -1: /* fork failure */
          return pid;

        case 0: /* child */
         /* establish a new process group, and put the child in
          * foreground if requested
          */
          if (pgrp < 0)
            pgrp = getpid();

          if (!setpgid(0,pgrp))
            if(fg) // If success and fg is set
                 tcsetpgrp(ctty, pgrp); // assign the terminal

          return 0;

        default: /* parent */
          /* establish child process group here to avoid race
             conditions. */
          if (pgrp < 0)
                  pgrp = pid;
          setpgid(pid, pgrp);

          return pid;
      }
}
```

3.4 Exec*

The child process resulting from a `fork` is a clone of the parent process. In particular, the child process runs the parent program (e.g., `dsh`), initially with all of its active data structures as they appeared in the parent at the time of the `fork`. The parent and child each have a (logical) copy of these data structures. They may change their copies indendently: neither will see the other's changes.

The `exec_()` family of system calls (e.g., `execve`) enables the calling process to execute an external program that resides in an executable file. An `exec_()` system call never returns. Instead, it transfers control to the main procedure of the named program, running within the calling process. All data structures and other state relating to the previous program running in the process—the calling program—are destroyed.

3.5 Wait*

The `wait_()` family of system calls (e.g., `waitpid`) allows a parent process to query the status of a child process or wait for a child process to change state. You will use it to implement the `jobs` built-in command, and also to wait for foreground processes to exit or pause (stop), or change state in some other way reported by `waitpid`. The `WNOHANG` option turns `waitpid` into a query: it returns the current status of a child process without actually waiting for child status to change. The `WUNTRACED` option waits until the child exits or stops.

```
/* Reaping all the child processes using the WNOHANG option.
 * The waitpid call will be non-blocking and the parent
 * can continue to do useful work.
 */
while (waitpid(-1, NULL, WNOHANG) > 0) {
          ...
}
```

3.6 Input/Output redirection

The shell supports I/O redirection using the "<" and ">" special characters as described above. Here is an example:

```
$./dsh
dsh$ /bin/echo "redirecting output" > outFile
dsh$ /bin/cat outFile
redirecting output

dsh$ /bin/echo "this is an input file" > inFile
dsh$ /bin/cat < inFile
this is an input file

dsh$ /bin/cat < inFile > outFile #prints the prompt immediately
dsh$ /bin/cat outFile
this is an input file
```

To implement I/O redirection you may use the `dup2()` system call. `dup2()` duplicates an open file descriptor onto the named file descriptor. For example, to redirect all the standard error output `stderr (2)` to a standard output `stdout (1)`, simply invoke `dup2()` as follows:

```
/* Redirect stderr to stdout */
  dup2(2, 1);
```

3.7 Pipelines

A pipeline is a set of processes chained by their standard streams, so that the output of each process (`stdout`) feeds directly as input (`stdin`) to the next one. If an argument contains a symbol `|`, the command-line contains a pipeline. The example below shows the contents of the file, produced by the output of a `cat` command, are fed directly as an input to the to `wc` command, which then produces the output to `stdout.`

```
$./dsh
dsh$ /bin/cat inFile
this is an input file
dsh$ /bin/cat inFile | /bin/wc
  1 5   22  #executes wordcount program on inFile; result 1 line;
      5 words; 22 characters
```

Pipes can be implemented using the `dup2()` system call, which was explained earlier.

4 Additional features

The shell prints an error message beginning with "`Error:`" if an input line is malformed, or on any error condition. Your shell may generate other outputs as well, as you choose.

You can use the `perror()` library function to print informative messages for errors returned by system calls. `perror()` writes a string on the standard error output: the prefix is a string that you specify ("`Error:` ..."), followed by a standard canned summary of the last error encountered during a previous system call or library function. Check also the man pages of `errno`.

Logging: The `dsh` *devilishly* causes all child processes to log their errors to a file, instead of to the terminal.

When the shell detects that a child process has exited, it should output a string reporting the event and the exit status of the child. In general, the processes of a job will exit together. When the shell detects that a job has exited, the shell frees its job number and discards any record of the job.

5 Getting started

The source code is available at `http://www.cs.duke.edu/courses/fall12/compsci210/projects/lab1b/`. Copy the parser source code files into a directory, cd into that directory, and type "make".

5.1 Suggested plan for implementation

1. Read this handout. Read the Bryant and O'Hallaron section on process creation and execution [4, 1].
2. Read the man pages for `fork(), exec_(), wait_(), dup2(), and exit()`.
3. Write small programs to experiment with these system calls.
4. Read man pages for `tcsetgrp()` and `setpgid()`.
5. Read the code we provided for `tcsetgrp()` and `setpgid()` and combine it with earlier programs you have written.

6. Using the parser we gave, start writing single commands.
7. Add support for running programs in the background, but do not worry about printing the message when a background job terminates (asynchronous notification). Add the jobs command while you are doing this—it may prove helpful for debugging.
8. Add input and output redirection.
9. Add code to print a message when a background job terminates.
10. Add code for logging.
11. Add job control features - implement foreground (fg) and background (bg) commands.
12. Add support for pipes using the `dup2()` system call.
13. Finish up all of the details
14. Test, test, and test
15. Go *devil*s{hell}. Celebrate!

6 Roadblocks and hints

You are encouraged to post your questions or issues to piazza [3]. The instructor or the TAs will post tips that may point you and others in the right direction.

You need to use `SIGCONT` to move a job from a background to a foreground process group. The GNU libc [1] pages on implementing a shell can be handy.

7 What to submit

Submit a single source file named as `dsh.c` along with a README file describing your implementation details, the results, the amount of time you spent on the lab, and also include your name along with the collaborators involved.

You will submit your code through the website at [2]. You should choose the course label as "compsci210" and then the lab label as "lab1b". You can submit the files multiple times before the deadline and we will treat the latest submission as the final submission for grading.

The grading is done on the scale of 100 as per below:

- Basic command support (process creation and execution, and error handling) : 10 points
- Input/Output redirection and bulit-in command `cd` : 20 points
- Pipes : 20 points
- Process groups, job control, and logging : 40 points
- Readability : 5 points
- README: : 5 points

References

[1] Gnu notes on implementing a shell. `http://www.gnu.org/software/libc/manual/html_node/Implementing-a-Shell.html#Implementing-a-Shell`.

[2] Lab submission page. `https://www.cs.duke.edu/csed/websubmit`.

[3] Piazza discussion page. `https://piazza.com/class#fall2012/cps210`.

[4] Randal E. Bryant and David R. O'Hallaron. *Exceptional Control Flow, Excerpts from Chapter 8 from Computer Systems: A Programmer's Perspective, 2/E (CS:APP2e).* `http://www.cs.duke.edu/courses/fall12/compsci210/internal/controlflow.pdf`.

[5] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.