Lab 2: Unix Programming: Implementing a Shell
COMPSCI 210: Introduction to Operating Systems

1 Introduction

Unix [3] 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*. The inputs to the shell are program executable or simply commands, and when you type in the input (in response to its prompt), the shell spawns a child process that executes that command along with any arguments passed for that command. The shell also links the input/output from terminal to other commands in the pipeline or to standard 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 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. You can find out which shell you are running by typing "echo $SHELL" at a prompt.

2 The Devil Shell: `dsh`

For 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 later in the handout.

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*.

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.

3.1 Built-in commands

The shell executes built-in commands in its own context, without spawning a child process or a job. Whenever an input is supplied, the shell checks for its list of built-in commands—and if the command does not matches, the shell spawns a child process for the command to execute. For example, the command `quit` terminates a shell. `quit` is a built-in command which should internally invoke `quit` will invoke the `exit()` system call to terminate the shell.

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 is the same as process group number, which is typically the process id of the parent process group. You can support a logical index job ids starting from 0 as in real shell, but it is optional for this lab. To implement logical job index, you can associate a job number j within a integer range $0 < j < 20$. 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.

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.

3.2.1 Fork

Fork system call creates a copy of itself in a separate address space. The original process that calls `fork()` system call is called a parent process and the newly created process is called a child process.

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 provided with the code 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.

3.2.2 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.

3.2.3 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 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) {
          ...
}
```

3.2.4 Continue job

To continue a stopped job, the combination of kill system call and SIGCONT signal comes handy.

```
/* Sends SIGCONT signal to wake up the blocked job */
void continue_job(job_t *j) {
     if(kill(-j->pgid, SIGCONT) < 0)
          perror("kill(SIGCONT)");
}
```

3.3 Input/Output redirection

Instead of reading and writing from stdin and stdout, one can choose to read and write from a file. The shell supports I/O redirection using the special characters "<" for input and ">" for output respectively.

Redirection is relatively easy to implement: just use close() on stdout and then open() on a file.

With file descriptor, you can perform read and write to a file. If you have only used fopen(), fread(), and fwrite() for reading and writing to a file. Unfortunately, these functions work on FILE* , which is more of a C library support; the file descriptors are hidden.

To work on a file descriptor, you should use open() , read(), and write() system calls. These functions perform their work by using file descriptors.

You may use the dup2() system call that 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);
```

Whenever there is a input/output redirection, the parser sets the mystdin and mystdout members of the job structure to the predefined MACRO descriptors INPUT_FD and OUTPUT_FD. The parser also correspondingly sets the file labels to ifile and ofile.

3.4 Pipelines

A pipeline is a sequence 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.`

```
$./dsh
dsh-287$ /bin/cat inFile
this is an input file
dsh-287$ /bin/cat inFile | /bin/wc #Asynchronously starts two
   process: cat and wc; pipes the output of cat to wc
  1 5   22  #executes wordcount program on inFile; result 1 line;
      5 words; 22 characters
```

Pipes can be implemented using the `pipe()` and `dup2()` system calls. A more generic pipeline can be of the form:

```
p1 < inFile | p2 | p3 | ....  | pn > outFile
```

where `inFile` and `outFile` are input and output files for redirection. Implementing multiple pipelines can be non-trivial and you have more than one design choice. The key is to set the process group ids correctly and make sure the process group leader does not become inactive until all the subsequent processes in the pipeline are set to the same process group. If you choose to wait until each process is completed (using explicit `wait_()` call), then the process group leader (which is the first process in the pipeline) will become inactive and hence, your calls to subsequent `setpgid()` will fail. So: you have to keep the process group leader active and also make sure to perform the wait in the right order.

4 Additional features

We have some additional features specific to the `dsh`:

- *Compile and execute C programs by default*: `dsh` compiles any command with extension ".c" to produce an executable "devil", which is executed immediately following the compilation. For example, a program (hello.c) with a simple print statement "A devil shell" can be compiled and executed by `dsh` when "hello.c" is issued on the command line.

  ```
  $./dsh
  dsh-287$ /bin/ls
  hello.c
  dsh-287$ hello.c
  A devil shell
  dsh-287$ /bin/ls
  hello.c devil #executable is written to devil file
  ```

- *Batch mode*: The shell can be run in a batch mode instead of interactive mode by supplying a file as input. We test your shells in batch mode for large part and so you should support batch mode execution.

  In interactive mode, you display a prompt and the user of the shell will type in one or more commands at the prompt. In batch mode, your shell is started by specifying a batch file on its command line; the batch file contains the same list of commands as you would have typed in the interactive mode.

  In batch mode, you should not display a prompt. You should print each line you read from the batch file back to the user before executing it; this will help you when you debug your shells (and us when we test your programs). To print the command line, do not use printf because printf will buffer the string in the C library and will not work as expected when you perform automated testing. To print the command line, use `write(STDOUT_FILENO, ...)` this way:

  ```
  write(STDOUT_FILENO, cmdline, strlen(cmdline));
  ```

- *Error handling*: The shell prints an error message beginning with "`Error:`" to `stderr` 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 (dsh.log), instead to the terminal. The job status changes also be logged to the file dsh.log.

  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/spring13/compsci210/projects/lab2/`. 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 material from OSTEP [1] on process creation and execution. Bryant and O'Hallaron can be a handy reference [2].
2. Read the man pages for `fork(), exec_(), wait_(), dup2(), open(), read(), write(), and exit()`.
3. Write small programs to experiment with these system calls.
4. Read man pages for `tcsetpgrp()` and `setpgid()`.
5. Read the code we provided for `tcsetpgrp()` 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 `pipe()` and `dup2()` system call.
13. Add support for .c files compilation and execution.
14. Finish up all of the details
15. Test, test, and test
16. Go *devil*s{hell}. Celebrate!

6 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 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) : 10 points
- Input/Output redirection and bulit-in command `cd` : 10 points
- Pipes : 25 points
- Process groups, job control, and additional requirements specific to `dsh` shell (.c files compilation and execution, logging, and error handling): 40 points
- README: : 15 points

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Process API, Excerpts from Operating Systems: Three Easy Pieces.* `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf`.

[2] 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/spring13/compsci210/internal/controlflow.pdf`.

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

A The devil shell by example

Here we provide a walk-through of the *devil shell* by issuing some sample jobs. You can issue all the commands yourself on the sample shell, `dsh-example`, available in the repository.

To log the errors, you can use `dup2()` system call.

Note: The extra line after each job execution is added here for the readability and need not be present in the actual `dsh` output.

```
$ ls #My working directory for lab1b
dsh.c  dsh.h  Makefile

# Run make to produce the executable dsh
$ make
gcc -I. -Wall -DNDEBUG -o dsh dsh.c

#Move the executable to dsh-example; dsh-example is provided to
   you for playing around
$ mv dsh dsh-example
$ ./dsh-example #Starting the shell
#The process id (pid) for the shell is displayed by the prompt
dsh-23822$ ls
23823(Launched): ls
dsh.c  dsh-example  dsh.h  dsh.log  Makefile

dsh-23822$ sleep 50 #Issuing a sleep
23824(Launched): sleep 50
^Z    #Stopped the job by issuing ctrl-z

dsh-23822$ jobs #Checking the status of the jobs
23823(Completed): ls #The completed jobs are displayed only for
   the first time
23824(Stopped): sleep 50

dsh-23822$ jobs #Checking the status of the jobs again
23824(Stopped): sleep 50 #The completed jobs disappear


dsh-23822$ bg 23824 #Resuming my sleep job and putting it to
   background

dsh-23822$ jobs #Prompt immediately returns
23824(Running): sleep 50 #The status will be running

dsh-23822$ fg 23824 #Resuming my sleep job
^Z    #Stopping again the job by issuing ctrl-z

dsh-23822$ fg 23824 #Resuming my sleep job
^C    #Terminating the job by issuing ctrl-c

dsh-23822$ jobs #Checking the status again
23824(Completed): sleep 50 #The status for sleep is now completed
    since we terminated the job by issuing ctrl-c

dsh-23822$ sleep 10 & #Launching a job in the background
23825(Launched): sleep 10

#Run the jobs within 10 seconds after issuing ``sleep 10''
dsh-23822$ jobs
23825(Running): sleep 10 #Status of the sleep is running (in the
   background)

#Now, let the job complete by waiting for 10 seconds. No message
   will be displayed on the shell until we explicity request for
   it. Wait for 11 seconds and issue ``jobs''
dsh-23822$ jobs
23825(Completed): sleep 10 #Now the status show my sleep is
   completed

#Example of multiple jobs separated by the symbol ``;''
dsh-23822$ ls; ps
23840(Launched): ls
dsh.c  dsh-example  dsh.h  dsh.log  Makefile
#The second job is launched sequentially after the first job is
   completed
23841(Launched): ps
```

```
  PID TTY          TIME CMD
 23841 pts/13   00:00:00 ps
17690 pts/13   00:00:00 tcsh
17705 pts/13   00:00:00 bash
23822 pts/13   00:00:00 dsh-example

dsh-23822$ jobs #Issue jobs to see the status
23840 (Completed): ls #The status shows as two seperate jobs were
    run: ``ls'' followed by ``ps''
23841(Completed): ps

dsh-23822$ ls | wc -l #Example with pipes
23843(Launched): ls | wc -l
5

dsh-23822$     #quit the dsh by issuing ctrl-d


#Back to our launching shell; dsh.log will contain the log info
   written to stderr
$ls
dsh.c  dsh-example  dsh.h  dsh.log  Makefile

$ cat dsh.log #examining the log file
23823(Launched): ls
23824(Launched): sleep 50
23824: Stopped by signal 127 #Singal number for ctrl-z; can be
   obtained by issuing WTERMSIG(process->status)
23823(Completed): ls
23824(Stopped): sleep 50
23824(Running): sleep 50
23824: Stopped by signal 127
23824: Terminated by signal 2 #Signal number for ctrl-c
23824(Completed): sleep 50
23825(Running): sleep 10
23825(Completed): sleep 10
23840(Launched): ls
23841(Launched): ps
23840(Completed): ls
23841(Completed): ps
23843(Launched): ls | wc -l #The completion for this job is not
   shown in the log as we did not issue ``jobs'' command
```