Lab 2: Implementing a Shell
COMPSCI 310: Introduction to Operating Systems

1 Shells are cool

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*.

```
$ echo Hello
Hello
```

The inputs to the shell, referred to as *commands* are composed of program executables(viz., *echo*) or built-in functions. Whenever you type in a command, in response to the shell's prompt(the dollar($) sign in the example above), 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.

```
$ echo Hello
Hello
# (NOTE: This is a shell comment, just like comments in Java, C, Python, ...)
# echo is a program that simply ``echoes'' on the terminal whatever it
   receives as input.
$ echo Hello | wc
1       1       6
# wc is a command that outputs the number of words, lines and bytes in its
   input; try ``man wc'' to learn more.
```

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.

```
# dumping the contents of a simple shell script
$ cat loop.sh
for i in `seq 1 5`; do
  echo " - $i - "
done
# cat dumps of the contents of its input on the terminal.

# calling the shell ``sh'' and using it to run the shell script
```

```
$ sh /tmp/loop5.sh
- 1 -
- 2 -
- 3 -
- 4 -
- 5 -
```

The shell also provides an interactive environment in which a user may compose new programs(sequence of commands) and execute them immediately, without a need to compile them in advance.

```
# the same shell script executed right from the terminal
# (not stored in a file!)
$ for i in `seq 1 5`; do echo " - $i - "; done
- 1 -
- 2 -
- 3 -
- 4 -
- 5 -
```

There are different kinds of shells(after all they are just programs which can be easily extended to support different kinds of features), and you can find out which one you are running by *inspecting* the *environment variable* as follows.

```
$ echo $SHELL
/bin/zsh
```

And, if you think shell is a boring program, please download the "fish" shell from `http://fishshell.com` and try it.

2 The Devil Shell: `dsh`

For this lab you will use Unix system calls to implement a simple shell - *devil shell (dsh)*. `dsh` supports a few cool 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 (optionally) monitors the progress of its children.

`dsh` prints a prompt of your choosing(get creative!) before reading each input line. As part of this lab, you will change the prompt to include the process identifier.

```
$ dsh
# starting devil shell from a terminal...

dsh-2400$
# devil shell started; waiting for inputs at the prompt.
# Observe that the PID of dsh - 2400 is part of the prompt
```

`dsh` reads command lines from its standard input and interprets one line at a time. We provide a simple command line parser(in parse.c[1]) to save you the work of writing one yourself. `dsh` exits when the user issues the built-in command "quit", which you will implement. You can also exit `dsh` by pressing "ctrl-d" at the terminal. "ctrl-d" sends an *end-of-file* marker to signal the program that there is no more input to process (the reason why it makes sense for `dsh` to quit). The parser we provide already detects the EOF marker and indicates if it was received as input; you merely have to handle this case and route to the control to your "quit" implementation.

```
$ dsh
# starting devil shell from a terminal...
```

[1]Please do not waste time looking into the parser until you finish the assignment.

```
dsh-2353$
# Observe that the pid at the prompt keeps changing.

dsh-2353$ quit
$
# dsh terminated and control reverts back to the original terminal.
```

The command-line input to `dsh` can contain one or more commands, separated by the special characters - "`;`" and "`|`". The supplied command-line parser supports four special characters "`;`", "`|`", "`<`" and "`>`". The special characters - "`<`" and "`>`", are shell directives: "`<`" redirects the standard input of a child process to read from a file, and "`>`" redirects the standard output of a child process to write to a file.

We define the syntax of input that `dsh` accepts using a Backus Normal Form(BNF)[2] notation, as follows, to make the shell syntax easy to understand.

Listing 1: BNF for Shell jobs

$\langle$*command-line*$\rangle$ ::= $\langle$*job*$\rangle$ ($\langle$*semicolon*$\rangle$ $\langle$*job*$\rangle$)* [$\langle$*semicolon*$\rangle$]
$\langle$*job*$\rangle$ ::= $\langle$*command*$\rangle$ ($\langle$*pipe*$\rangle$ $\langle$*command*$\rangle$)*
$\langle$*command*$\rangle$ ::= $\langle$*cmd-name*$\rangle$ [$\langle$*arguments*$\rangle$] [$\langle$*shell-directives*$\rangle$]
$\langle$*shell-directives*$\rangle$ ::= [$\langle$*less-than*$\rangle$ $\langle$*file-name*$\rangle$] [$\langle$*greater-than*$\rangle$ $\langle$*file-name*$\rangle$]
$\langle$*arguments*$\rangle$ ::= $\langle$*argument*$\rangle$ ($\langle$*space*$\rangle$ $\langle$*argument*$\rangle$)* $\langle$*argument*$\rangle$ ::= ($\langle$*alpha-numeric*$\rangle$)+

The semicolon(;) is used for separating a sequence of commands, executed from left to right, sequentially. Unlike regular shells that halt on an error, `dsh` is so cool that it simply ignores the error or failure of a command and moves on to execute the next. The fact that the command failed is recorded in a log file. As part of the lab, you will handle the case where a user input can contain a sequence of commands delimited by a semicolon, and execute the commands sequentially in the order specified.

```
dsh-10437$ echo Hello; echo World
10445(Launched): echo Hello
Hello
10446(Launched):  echo World
World

# Observe the order of execution.

dsh-10437$ echos Hello; echo World
10453(Launched): echos Hello
10454(Launched):  echo World
World

# Note that the first command failed, because it is neither a built-in nor
#  a recognized program that the shell can launch.
# Although, \dsh does not halt on encountering such errors, it records them
#  in a log file, named dsh.log by default.

# quit \dsh and examine the log file using the program 'cat'
$ cat dsh.log
8356(Launched): echos Hello
execvp: No such file or directory
8357(Launched):  echo World

# Note the error reported after attempting to launch 'echos'
```

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

[2]BNF: `http://en.wikipedia.org/wiki/BackusNaur_Form`

command is malformed. You will ensure that dsh supports this cool feature and allows a user to use pipe and compose commands together to perform sophisticated tasks.

```
dsh-22698$ echo Hello | wc -c
22699(Launched): echo Hello | wc -c
6
```

In addition, the last non-blank character of a job may be an "&". The meaning of "&" is simply that any child processes for the job execute in the background. If a job has no "&" then it runs in the foreground. Implementation of background jobs in dsh is optional.

Each command (of a job) is a sequence of words (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-ins (discussed later) are implemented directly within dsh .

Although limited, dsh supports *input/output redirection* using the special characters "<" and ">". Note that these two special characters are treated as directives to the shell to modify the standard input and/or standard output of the command. The directives always follow a command whose standard input and/or output are to be rerouted.

- When the special character "<" is used along with a file name, dsh (like any regular shell)arranges for the command to read its standard input(stdin) from the specified file.
- When the special character ">" is used along with a file name, dsh arranges for the command to write its standard-output to the specified file.

```
dsh-22698$ echo Hello > hello.txt
22838(Launched): echo Hello > hello.txt
dsh-22698$ wc < hello.txt
22841(Launched): wc < hello.txt
1 1 6
```

2.1 dsh built-ins

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 match, 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 will invoke the exit() system call to terminate the shell.

dsh has the following additional built-in commands, which you will implement:

- 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 its leader (the first process in the job).
- bg. Continue a named job in the background. The job is given by a single argument: the job number. Implementation of support for bg is optional.
- fg. Continue a named job, if specified, in the foreground. If no arguments are specified it continues the last job stopped(or suspended), by the user, in dsh . To suspend a job, a user can press "ctrl+z" in the terminal; you should ensure that this happens properly in dsh .
- cd. Change the shell's current directory. The target is given by a single argument: a pathname.

2.2 Launching external programs

If a command is not recognized as a built-in, then it's assumed to be an external program to execute. If the command is not an external program either, then the error should be logged(discussed in the next section). The program executes in a child process (to `dsh`), grouped with any other child processes that are part of the same job. To implement this part, you will understand and use a few basic Unix system calls(discussed later in section 3) to launch and manage child processes.

2.3 `dsh` is a lot cooler!

We have a few cool features specific to `dsh`:

- *Automagically compile and execute C programs*: `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
# listing directory contents

dsh-287$ hello.c
A devil shell

# automagically compiles the program and executes it

dsh-287$ /bin/ls
hello.c devil

# the compiled binary (from hello.c) is named devil
```

- *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). When you print the command line, use `fflush` to ensure that the print completes immediately.

- *Error handling*: The shell prints an error message beginning with "`Error:`" to its `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 of to the terminal. The job status changes are also 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.

3 Notes on System Calls

This section provides some notes on system calls you will need for this lab.

3.1 Fork

We provide a procedure called `spawn_job` that uses the `fork` system call to create a child process for a job. The `spawn_job` routine also attends to a few other details. Each job has exactly one *process group* containing all processes in the job, and each process group has exactly one process that is designated as its *leader*. If a job has only one process then that process is the leader of the job's process group.

Process groups are useful for signal delivery to all the processes in the group. The kernel distributes signals generated from the terminal (ctrl-c, ctrl-z) to all processes in a *foreground* process group. The purpose of the process group is to allow the shell to control which processes receive such signals. Note that `dsh` need not and should not declare signal handlers for `ctrl-c` or `ctrl-z`. The lack of support for signals means that `dsh` will not notice if one of its background jobs(optional implementation) stops or exits. Instead, the shell has a built-in command called `jobs` to check and report the status of each of its children.

At any given time, at most one process group, which is the foreground process group controls access to the terminal, *tty*, for its input and output. The shell sets the foreground process group when starting a new set of processes; by convention the new process group number is the same as the process ID of one of the members of the 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 `spawn_job` 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 using *tcsetpgrp*.

3.2 Exec*

The `exec_()` family of system calls (e.g., `execvp`) 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.

3.3 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.

3.4 I/O 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 using `creat()`, `open()` , `read()`, and `write()` system calls.

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:

```
/ close(stdout) and 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`.

3.5 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 command line contains a symbol `|`, the processes are chained by a pipeline.

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.

The descriptors in the child are often duplicated onto standard input or output. The child can then exec() another program, which inherits the standard streams. `dup2()` is useful to duplicate the child descriptors to stdin/stdout. For example, consider:

```
int fd[2];
pid_t pid;
pipe(fd);

switch (pid = fork()) {
case 0: /* child */
dup2(fd[0], STDIN_FILENO); /* close stdin (0); duplicate input end of the pipe
    to stdin */
execve(...);
....
}
```

where `dup2()` closes stdin and duplicates the input end of the pipe to stdin. The call to `exec_()` will overlay the child’s text segment (code) with new executable and inherits standard streams from its parent–which actually inherits the input end of the pipe as its standard input! Now, anything that the original parent process sends to the pipe, goes into the newly exec’ed child process.

4 Getting started

The source code is available at `http://www.cs.duke.edu/courses/fall13/compsci310/projects/lab2/`. Copy the parser source code files into a directory, cd into that directory, and type "make".

4.1 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 input and output redirection.
8. Add code to display the status of the background/foreground jobs, when "jobs" is issued. Any background job which is completed earlier but have not been notified (since dsh need not support asynchronous notification), needs to be reported by "jobs".
9. Add code for logging and error handling.
10. Add support for pipes using the `pipe()` and `dup2()` system call.
11. Add support for .c files auto-compilation and execution.
12. Finish up all of the details
13. (Optional) 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.
14. (Optional) Add job control features - implement foreground (fg) and background (bg) commands.
15. Test, test, and test
16. Go *devil*s{hell}. Celebrate!

5 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 built-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 `dsh` 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 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.

```
# starting dsh
$ ./dsh-example

# Observe the pid of dsh in the prompt.
dsh-23822$

# listing current directory contents
dsh-23822$ ls
23823(Launched): ls
dsh.c  dsh-example  dsh.h  dsh.log  Makefile

# The above job consists of one command ‘‘ls’’ which is an external program.
# \dsh spawns a child process and launches this external program in the child.
# The programs runs, prints its output and quits.
# This finished job can be viewed using the jobs (built-in) command.

# checking the status of the finished ‘‘ls’’ job
dsh-23822$ jobs
23823(Completed): ls

# completed jobs are only displayed once.

dsh-23822$ jobs

# now, jobs shows nothing!

# launching multiple jobs using a semicolon
dsh-23822$ ls; ps
```

```
23840(Launched): ls
dsh.c  dsh-example  dsh.h  dsh.log  Makefile
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

# note that ``ps'' ran after ``ls''

# Note that semicolon is used as a delimiter to specify multiple jobs
dsh-23822$ jobs
23840 (Completed): ls
23841(Completed): ps

# Observe that the output above shows two jobs (issued in the same line)

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