CIS 5480 Project 1
penn-shell: We Sell Seashells

Midway upon the journey of our life
I found myself within a forest dark,
For the straightforward path had been lost

Dante Alighieri, *The Divine Comedy*

Prof. Boon Thau Loo

MILESTONE: Feb. 8 @ 10pm
DUE: Feb. 20 @ 10pm

Directions

You **must** work in groups of 2 for this project. A team consisting of at least one undergraduate can work in groups of 3. You may use code from Project 0, **but only code you or your partner wrote**. If you are caught using code from other groups or any sources from public code repositories, your entire group will receive ZERO for this assignment, and will be sent to the Office of Student Conduct where there will be additional sanctions imposed by the university.

Overview

In this assignment you will implement a shell named *penn-shell* that is a simplified version of a fully featured shell like `bash`. *penn-shell* will have foreground and background processes; standard input and output redirections; pipelines; and job control. As in the previous project, you will **only** use the functions specified in this project handout.

We will publish the Makefile used by the autograder. In your C source code, you should use the `PROMPT` macro defined in the Makefile as you did in Project 0.

1 Specification

As described in Project 0, an interactive shell is a simple loop that prompts, executes, and waits. In this assignment, you will implement a shell with four additional features: standard input/output redirections, pipelines, background/foreground processing, and job control. Also, the shell can run as a non-interactive shell without supporting background processing and job control. You may reuse any code from the previous assignment, as long as **one of your team members wrote it.**

1.1 Notable differences from Project 0

There are some notable differences between *penn-shell* and *penn-shredder*:

- There is no longer a time limit on program execution. In fact, you may **not** use `alarm(2)` at all.
- *penn-shell* can read a script file at invocation through redirected standard input.
- You can use **`getline(3)`** in addition to `read(2)`.
- You should use **`execvp(3)`** instead of `execve(2)`.

1.2 Non-interactive mode of *penn-shell*

penn-shell can run in either *interactive* or *non-interactive* mode. Like *penn-shredder*, *penn-shell* runs in interactive mode by default: it prompts the user for input. However, it can also run in non-interactive mode by redirecting the standard input at invocation:

```
# cat script.txt
echo hello world
pwd
# ./penn-shell < script.txt
hello world
/root/cis3800/22fa-project-1-<username>
#
```

By default, the standard input of *penn-shell* refers to the terminal. In the above non-interactive mode, it's redirected to the `script.txt` file. You can use `isatty(3)` to check if the standard input is redirected.

1.3 Standard input/output redirections

By default, standard input and output refer to the terminal. However, they can be redirected to read from and write to files, respectively.

1.3.1 Standard input redirection

In the following example, the `wc` command reads from the `script.txt` file via the standard input redirection operator < and prints the number of lines in the file:

```
penn-shell> wc -l < script.txt
2
penn-shell>
```

If script.txt can't be opened successfully, penn-shell should throw an error without executing the command.

1.3.2 Standard output redirection

In the following example, the `pwd` command outputs to the `out.txt` file via the standard output redirection operator >:

```
penn-shell> pwd > out.txt
penn-shell> cat out.txt
/root/cis3800/22fa-project-1-<username>
penn-shell>
```

If out.txt doesn't exist, it will be created. Otherwise, it will be truncated to 0 size.

The >> operator is identical to >, except that the standard output is **appended** to the file if it already exists:

```
penn-shell> echo foo > log.txt
penn-shell> echo bar >> log.txt
penn-shell> cat log.txt
foo
bar
penn-shell>
```

1.3.3 Combining redirections

The standard input/output redirections can be combined. For example, the `cat` command below effectively copies `log.txt` to a new file `log2.txt`.

```
penn-shell> cat < log.txt > log2.txt
penn-shell> cmp log.txt log2.txt
penn-shell>
```

1.4 Pipelines

Pipes are another form of redirections. Instead of redirecting to a file, a pipe connects the standard output of one process to the standard input of another. This is best demonstrated via an example:

```
penn-shell> head -c 128 /dev/urandom | base64
```

The `head` command will read the specified number of random bytes from `/dev/urandom`, and its output is *piped* to the standard input of the `base64` program. In this assignment **you are required to implement a multi-process pipeline**, *i.e.*, one process can pipe output to the input of another process, whose output can be an input to another and so on as seen in many full-featured shells. A good example of a multi-stage pipeline would be:

```
penn-shell> echo brains > log
penn-shell> cat log | grep brains | wc -l
1
penn-shell>
```

For the above pipeline, the output of the `cat` (i.e. the content in `log`), is sent to the `grep` command, whose output is piped to the `wc` utility. As a result, the above command would print out the number of lines in the `log` file that contain the word `brains`.

Note: Implementing a multi-process pipeline isn't trivial if not planned out properly. We recommend that you think through the flow of the execution (e.g. draw a diagram of how each part of the pipe should be executed).

1.4.1 `pipe(2)` system call

A pipe is a special unidirectional file descriptor with a single read end and a single write end. To generate a pipe, you will employ the `pipe(2)` system call which will return two file descriptors, one for reading and one for writing. You will then `dup2(2)` each end to the standard input and output of the respective processes in the pipeline. Be sure to close the file descriptors that are unused, or else your pipe will not work.

1.4.2 Jobs and process group isolation (interactive shell only)

When executing a pipeline, *penn-shell* will execute multiple processes in parallel, forking for each command in the pipeline. All processes are part of the same *job*; a single command line execution. If penn-shell runs as an interactive shell, each job **must be in its own process group** that is different from the shell's process group and other job's process groups. (Even a job containing a single process must be in its own process group.) To set a process group, you will use the `setpgid(2)` system call, and you should refer to the manual for more details.

Isolating jobs via process groups can affect terminal signaling. Before, all jobs were members of the same process group, the shell's process group, but now they will be in separate groups. You may need to block certain signals during critical sections. See the `signal.h` and `signal(7)` manuals and *APUE*[1] for the reasons signals are delivered and which system calls are re-entrant and which are not. If a non re-entrant (or "unsafe") function is used in contexts where it can be interrupted, the behavior of the program is undefined. This can cause many headaches and bugs, so be sure to consider what is critical and what is not.

1.5 Foreground and background processes (interactive shell only)

A key feature of *penn-shell* is the ability to start a job in either the foreground or the background. In the previous assignment, all jobs executed in the foreground. The shell executed the job and waited until it completed before prompting the user for more input. Alternatively, a job can be started in the background, and the shell will immediately prompt the user for the next command following executing the background job.

To start a job in the background, the user appends the `&` token to the last command of a pipeline. Here is an example:

```
penn-shell> sleep 1&
Running: sleep 1
penn-shell> echo Hello World
Hello World
Finished: sleep 1
penn-shell>
```

`sleep` is started in the background, and the shell drops back to a prompt allowing the user to execute `echo` prior to the completion of `sleep`. The user was notified of the status of the background process, and this requirement is described in more detail in Section 1.6.3.

Additionally, a job can be started in the foreground and later moved to the background (and vice versa). This is done through a combination of terminal control (Section 1.5.2) and job control builtin commands (Section 1.6.1).

1.5.1 Polling background processes

With multiple jobs executing at the same time, it is infeasible to simply call `wait(2)` and expect everything to work. For example, if there is a background job that has completed while a foreground job is running, `wait()` will return due to the completion of the background job but not the foreground one. Moreover, *penn-shell* must wait on **all** completed jobs so that they do not become zombies[2]. Worse, a job may create two background processes, each of which must be waited on.

[1] W. Richard Stevens and Stephen A. Rago. Advanced Programming in the UNIX Environment (2/e or 3/e). Addison-Wesley Professional. 2005

[2] mmmm ... brains!

penn-shell periodically polls for state changes of all background processes. Such polling consumes CPU cycles, so **you should poll only once for each shell read** and pass `-1` as the pid argument to `waitpid(2)`, e.g., `waitpid(-1, &status, WNOHANG | WUNTRACED)`.

1.5.2 Terminal control

In order to support background and foreground jobs, *penn-shell* is now responsible for delegating the terminal control. This becomes particularly important when jobs are isolated via process groups. For example, if a process group that is not in control of the terminal attempts to read from it, a `SIGTTIN` signal is delivered and the default action is to stop the process group.

This concept is best demonstrated with the `cat` program. When run with no arguments, `cat` will read from the standard input and write to the standard output. If `cat` is executed in the foreground, then it should be in control of the terminal. For example:

```
penn-shell> cat > file.txt
Hello World
^D
penn-shell>
```

Here, the user used `cat` as a mini-text editor by redirecting input to the file `file.txt`. However, if this were executed in the background:

```
penn-shell> cat > pennshell&
Running: cat > pennshell
penn-shell>
Stopped: cat > pennshell
penn-shell>
```

The `cat` command would be stopped because it attempted to read from the standard input when it was not in control of the terminal. A `SIGTTIN` would be generated and its default action is to stop the process.

Although *penn-shell* should not change the disposition of `SIGTTOU`, it must stop background jobs that try to read from the standard input while allowing the *foreground* job to read from the standard input.

To do this, you will use the library function `tcsetpgrp(3)`. We will provide a toy program (tc_example.tar) that demonstrates how to use it.

Another consequence of having multiple process groups is handling the delivery of terminal signals. There are two that you should pay particular attention to: `SIGINT`, the signal generated by `Ctrl-C`; and `SIGTSTP`, the signal generated by `Ctrl-Z`.

penn-shell is responsible for properly relaying those signals to the foreground job. If you have proper process group isolation and terminal-control setup, then this should happen by default. In any event, *penn-shell* **should never stop or exit** due to the delivery of `SIGTSTP` or `SIGINT`. You should set up appropriate signal handlers to relay the signal to the right process group.

A sample of correct behaviors for terminal signals are as follows:

```
penn-shell> ^C
penn-shell> ^Z
penn-shell> sleep 100
^C
penn-shell> sleep 200
^Z
Stopped: sleep 200
penn-shell>
```

1.6 Job control (interactive shell only)

penn-shell supports basic job control.

1.6.1 Job control builtins

A shell builtin is a built-in command that runs as a function in the same process as the shell. That is, the builtin is not forked as a child process. *penn-shell* has three builtins to support job control: `bg`, `fg`, and `jobs`.

`penn-shell` has a notion of **the current job**. If there are stopped jobs, the current job is the most recently stopped one. Otherwise, it is the most recently created background job.

- `bg`

  Syntax: `bg [job_id]`

  `bg` resumes in the background the stopped job identified by `job_id`, which defaults to that of **the current job**. If the job is already running or no such job exists, `bg` throws an error.

- `fg`

  Syntax: `fg [job_id]`

  `fg` brings to the foreground the background job identified by `job_id`, which defaults to that of **the current job**. If the background job is stopped, `fg` resumes it before bringing it to the foreground. If the job does not exist, `fg` throws an error.

- `jobs`

  Syntax: `jobs`

  `jobs` prints to the standard error all background jobs, sorted by `job_id` in ascending order. If there are no background jobs, it simply returns without throwing an error. (See the example below for the format.)

The following example illustrates how these builtins work together:

```
penn-shell> jobs
penn-shell> sleep 100
^Z
Stopped: sleep 100
penn-shell> sleep 200 &
Running: sleep 200
penn-shell> jobs
[1] sleep 100 (stopped)
[2] sleep 200 (running)
penn-shell> bg
Running: sleep 100
penn-shell> jobs
[1] sleep 100 (running)
[2] sleep 200 (running)
penn-shell> fg
sleep 200
^C
penn-shell> jobs
[1] sleep 100 (running)
penn-shell>
```

Below is an example of running `fg` and `bg` with a job id:

```
penn-shell> sleep 100 &
Running: sleep 100
penn-shell> sleep 200 &
Running: sleep 200
penn-shell> jobs
[1] sleep 100 (running)
[2] sleep 200 (running)
penn-shell> fg 1
sleep 100
^Z
Stopped: sleep 100
penn-shell> jobs
[1] sleep 100 (stopped)
[2] sleep 200 (running)
penn-shell> bg 1
Running: sleep 100
penn-shell> jobs
[1] sleep 100 (running)
[2] sleep 200 (running)
penn-shell>
```

A builtin is meant to run as a single command, and standard input/output redirections and the trailing `&` (if any) will be ignored. If the builtin is the first command of a multi-process pipeline, the rest of the pipeline should be ignored.

1.6.2 Job queue

penn-shell must maintain a queue of background jobs, identified by job id. **You must implement the queue with a linked list from scratch.**

Job ids start at 1 and must be unique. A job id must remain constant for the duration of the job. The job id for a new background job is always that of the most recently created background job plus 1.

1.6.3 Reporting job statuses

penn-shell must notify the user when a background job changes state. However, it should hold on to the notifications and print them right before prompting the user. The notifications should be printed to the standard error. The format for reporting job statuses is as follows:

- Stopping the foreground job: `Stopped: <pipeline>`
- Starting a job, or continuing a stopped job, in the background: `Running: <pipeline>`
- Continuing a stopped job in the foreground: `Restarting: <pipeline>`
- Bringing a running background job to the foreground: `<pipeline>`
- When a background job is completed: `Finished: <pipeline>`

where `<pipeline>` is a job without the trailing `&` if any. For example:

```
penn-shell> sleep 1 &
Running: sleep 1
penn-shell> echo hello world
hello world
Finished: sleep 1
penn-shell> sleep 100
^Z
Stopped: sleep 100
penn-shell> bg
Running: sleep 100
penn-shell> fg
sleep 100
^C
penn-shell>
```

1.7 Code design

This is a large, complex assignment. *You should organize your code in a reasonable way.* It is not reasonable to have all your code in one file. Take the time to break your code into modules. You must place all .c and .h files in the top directory where the Makefile will be located, for the autograder Makefile to be able to locate it.

1.8 Arguments to *penn-shell*

penn-shell accepts an optional option `--async`, which is needed only for the additional credit. Also, it can optionally read from redirected standard input. (See Section 1.2.)

1.9 Ctrl-C, Ctrl-D, and Ctrl-Z behaviors

If *penn-shell* runs as an interactive shell and has control of the terminal, Ctrl-C and Ctrl-D have the same behaviors as in *penn-shredder*, and Ctrl-Z has the same behavior as Ctrl-C. If *penn-shell* runs as a non-interactive shell, Ctrl-C terminates it and Ctrl-Z stops it.

1.10 Terminal signals

An interactive *penn-shell* **should never exit or stop** because of the following terminal signals: `SIGTTIN`, `SIGTTOU`, `SIGINT`, `SIGQUIT`, and `SIGTSTP`.

1.11 Parsing

This assignment is not about writing a parser. As such, we have provided a full parser so that you can focus on the other parts of *penn-shell* more relevant to an OS course. (See the Ed discussions announcement.)

2 Acceptable system calls and library functions

In this assignment you may use **only** the following system calls and library functions:

- `atoi(3)`
- `clearerr(3)`, `feof(3)`, and `ferror(3)`

- `close(2)`, `dup2(2)`, `open(2)`, and `pipe(2)`
- `execvp(3)`, `fork(2)`, and `waitpid(2)`
- `_exit(2)` and `exit(3)`
- `fprintf(3)`, `fputc(3)`, `fputs(3)`, `fwrite(3)`, `printf(3)`, `putc(3)`, `putchar(3)`, `puts(3)`, and `write(2)`
- `free(3)` and `malloc(3)`
- `getline(3)` and `read(2)`
- `getpgid(2)`, `getpgrp(2)`, and `setpgid(2)`
- `getpid(2)` and `getppid(2)`
- `isatty(3)`
- `kill(2)` and `killpg(2)`
- `strcmp(3)` and `strlen(3)`
- `perror(3)`
- `signal(2)`
- `tcgetpgrp(3)` and `tcsetpgrp(3)`

Using any other library function than those specified above will affect your grade on this assignment. If you use the `system(3)` library function, **you will receive a ZERO on this assignment.**

3 Pitfalls

Do not make the mistake of writing to stdin (file descriptor 0). Students have done this in the past and it becomes a headache to figure out where the output of the program went to. One way to verify if you are not writing to stdout is to redirect the output of the program to a file.

```
# ./penn-shell > output_of_penn_shell
```

4 Memory errors and leaks

You are required to fix all memory errors and leaks (except "still reachable" leaks) in your code.

5 Reusing code

We do allow the reuse of code as long as the code was personally authored by you and must be cited. We do not allow, for example, copying and pasting CIS 240 linked list boilerplate code into the project. Reuse of code not authored by you will result in a 0 for the project.

6 Developing Your Code

As before, you must develop your code using docker – **not ENIAC! Students who have been identified by CETS as developing this project on ENIAC will be penalized. Students who have been identified by CETS as responsible for launching a fork-bomb on ENIAC will receive a ZERO on this assignment.**

7 What to turn in

You must provide each of the following for your submission, *no matter what*. Failure to do so may reflect in your grade.

1. `README` file. In the `README` you will provide
   - Your name and Pennkey
   - A list of submitted source files
   - Overview of work accomplished
   - Description of code and code layout
   - General comments and anything that can help us grade your code
2. Your code. You may think this is a joke, but at least once a year, someone forgets to include it.

8 Submission

This project will be submitted on Gradescope. Place **all** relevant code and documentation into a **simple** directory, i.e. no sub-directory. Then, from the top of this directory, compress your files via the following command:

```
tar cvaf pennkey1-pennkey2-project1.tar.gz *.c *.h README
```

Replace `pennkey1` and `pennkey2` with your Pennkeys. You can then upload the compressed tar file to the Gradescope website.

9 Grading Guidelines

Each team will receive a group grade for this project; however, individual grades may differ. This can occur due to lack of individual effort, or other group dynamics. The team Git repository will be monitored and used to determine individual contribution. In most cases, everyone on a team will receive the same grade. Below is the grading guideline.

- 10% Milestone
- 5% Documentation
- 85% Functionality

Please note that general deductions may occur for a variety of programming errors including memory violations, lack of error checking, etc.

Your programs will be graded with docker and must execute as specified there. You may develop and test on other machines but you should ensure that your program functions properly with docker.

10 Milestone (10%)

In order to ensure that students stay on track and receive early feedback, we require all groups to submit an intermediate solution before the deadline. For the milestone, students are expected to have completed standard input/output redirections and multi-process pipelines (up to and including Section 1.4.1). Do include a README in this submission, including an accounting of the work you have completed thus far.

This milestone is designed to guide your workflow. As such, we will not grade it in significant detail.

11 Attribution

The primary rule to keep you safe from plagiarism/cheating in this project is to attribute in your documentation any outside sources you use. This includes both resources used to help you understand the concepts, and resources containing sample code you incorporated in *penn-shell*. The course text and APUE need only be cited in the latter case. You should also use external code sparingly. Using most of an example from the `pipe(2)` man page is ok; using the `ParseInputAndCreateJobAndHandleSignals()` function from *Foo's Handy Write Your Own Shell Tutorial* is not (both verbatim, and as a closely followed template on structuring your shell).