Project 2
penn-sh: 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*

Directions

This is an **individual** assignment. You may use code you wrote for Project 1, but never use any code from other sources. If you are caught using code from previous semesters, or any sources from public code repositories, your 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

As described in Project 1, a shell is a simple loop that prompts, executes, and waits. In this assignment, you will implement a shell named *penn-sh*. It will have 2 additional features: redirection and two-stage pipeline. You may reuse any code from the previous assignment, as long as **you wrote it.** You will **only** use the functions specified in this project handout.

Advice

This is a large and complex assignment. It will take you more than a coffee-fueled night to complete, so start early and work regularly. Modular programming will be your best friend in this project! Work in parts and divide-and-conquer. Do not try to complete everything at once; get one thing working before moving onto the next. Divide the work up such that each part gracefully integrates with the other. Additionally, going over the man pages in detail will save you a lot of debug time. Read it carefully and write sample programs to learn how each of the system calls functions under a variety of situations. Experimenting with small simple programs will cement your understanding of the system calls and how they are used. Also using version control can help save you from potential catastrophes.

Moreover, you are highly encouraged to utilize Ed Discussion to discuss the project with classmates and TAs. Before asking the question, please search in Ed Discussion to see if there are any similar questions asked before. If you plan to ask about your code/implementation, please consider filling in the following template to make the communication efficient. We are here to help you and make sure you get the most out of the course!

- GitHub URL:
- Description of Question:
- Expected Behavior:
- Observed Behavior:

1 Difference from Project 1

Before getting into the gritty details, here are some upfront difference from the previous assignment that you should keep in mind.

1.1 No time-limitation argument

First, there is no longer a time-limit on program execution. In fact, you will **not** use the `alarm(2)` system call at all. Thus, *penn-sh* should run without any argument as follows:

```
./penn-sh
```

1.2 Command Line Input

Second, your shell must handle program command line arguments. We have provided a command line parser for you, `token-shell.tar`. In the tar-ball you'll find a parser, `tokenizer.c`, a header file, `tokenizer.h`, and a sample program, `token-shell.c`. The tokenizer functions similar to `strtok(3)`, and you may use it freely. See the provided example program for more details on its functionality. **While the tokenizer calls functions unspecified in the specification, you may not use them elsewhere in your shell.**

1.3 Execution

In the previous assignment, you were only allowed to use `execve(2)`, but in this assignment, you will need to more easily access the programs on the system. As such, we will allow you to use `execvp(3)`. `execvp(3)` differs from `execve(2)` in two ways. First, `execvp(3)` will use the `PATH` environment variable to locate the program to be executed. That means the user of your shell will no longer need to provide a full path to the executable. Instead, the name of the executable command is enough. Second, `execvp(3)` does not provide a means to set the environment. As a result of the first difference, instead of using

```
penn-shredder# /bin/cat
```

Now you can type in "cat" to run the `cat` command.

```
penn-sh> cat
```

1.4 Terminal Signals

The shell's behavior of terminal signals remains **unchanged** from project 1: it **should never exit** due to the delivery of `SIGINT` (generated by `Ctrl-C`) from the terminal. It should only exit due to the delivery of `EOF` (generated by `Ctrl-D`). For child processes started from penn-sh, they should respond to `Ctrl-C` by following their default behavior on SIGINT. Notice that in this project we will implement pipeline, which means there may be 2 child processes running concurrently. In this case, both of them should respond to the terminal signals with their default behavior. For example:

```
penn-sh> ^C
penn-sh> sleep 100 | sleep 200
^C
penn-sh>
```

2 Project Milestones and Submission

2.1 Part 2A: Redirection

Each program has three standard file descriptors: standard output, input, and error. Normally, standard output and error are written to the terminal and standard input is read from terminal. One shell feature is the ability to redirect the standard file descriptors to (or from) files. A user requests a redirection by employing < and > symbols. Their usage is best demonstrated by an example. Consider this command line input:

```
penn-sh> head -c 1024 /dev/urandom > random.txt
```

The `head` program will read the first 1024 bytes from `/dev/urandom` which would normally be written to standard output. However, the > symbol indicates to redirect standard output to the file `random.txt`, and a file named `random.txt` is created with the random 1024 bytes written to it.

In a similar way, standard input can be redirected:

```
penn-sh> cat < /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family  : 6
(...)
```

It is also possible to combine redirections. The command below copies `/proc/cpuinfo` to a file in the current working directory named `cpuinfo`:

```
penn-sh> cat < /proc/cpuinfo > cpuinfo
penn-sh>
```

The order of the redirections symbols does not matter.

The command below is equivalent to the previous one:

```
penn-sh> cat > cpuinfo < /proc/cpuinfo
penn-sh>
```

If the standard output redirection file doesn't exist, it should be created. Otherwise, it should be truncated and the new contents should be written to it from the very beginning. For example,

```
penn-sh> echo Hello > output
penn-sh> cat output
Hello
penn-sh> echo Hi > output
penn-sh> cat output
Hi
penn-sh>
```

Alternatively, if the standard input file does not exist, an error should be reported. Check the *2.1.2 Redirection Errors* for more detail.

You are not required to implement the append functionality (>>) or the redirection of `STDERR`.

2.1.1 The `dup2(2)` and `open(2)` system calls

To perform redirection, you will use the `dup2(2)` and `open(2)` system calls. The `dup2(2)` system call duplicates a file descriptor onto another, and the `open(2)` system call will open a new file, returning a file descriptor. Here is a simple example of their usage (without error checking). More details can be found in the function man pages.

```
new_stdout = open("new_stdout", O_WRONLY | O_TRUNC | O_CREAT, 0644);
dup2(new_stdout,STDOUT_FILENO);
write(STDOUT_FILENO, "Helloooo, World!!!!",9);
```

This code snippet will open a file named `new_stdout` with the mode "write only", "truncate it if it's not empty", "create it if it doesn't exist". If it is a new file, the mask is set to 0644 (that's octcal), which gives the user permissions to read and write that file. Next, the new file descriptor is duped onto the current standard output file descriptor, and all subsequent writes to standard output will actually be written to `new_stdout` instead.

2.1.2 Redirection Errors

If a user provides invalid redirections, your shell should report errors. This could be because there are contradictory redirections of the same standard file descriptor, or because an open failed. Regardless of the error, your shell **should gracefully handle and report user input errors**.

To be clear, for this command:

```
penn-sh> cat > out < in
```

If currently there is no "in" file in the directory, `cat` will fail to open it. Thus, it will print error information:

```
penn-sh> cat > out < in
Invalid standard input redirect: No such file or directory
penn-sh>
```

Note that the "No Such file or directory" error message above was printed using `perror(3)`.

However, If we had created the "in" file, then the above a valid redirection:

```
penn-sh> touch in
penn-sh> cat > out < in
penn-sh>
```

Whereas bash in Linux handles both multiple input redirections and multiple output redirections, for our project, they are considered as an incorrect input. For example,

```
penn-sh> ls > out1 > out2
Invalid: Multiple standard output redirects
penn-sh> cat < in1 < in2
Invalid: Multiple standard input redirects or redirect in invalid location
penn-sh>
```

Specifically, a valid input can only have a maximum of one input and one output redirection. However, be sure to gracefully handle invalid inputs; don't allow the shell to crash or perform undefined behavior. For the error message, you must include the **"invalid"** keyword to pass the auto-grader.

2.1.3 Submission

Develop your code in the Git repository we assigned to you in project 1. However, create a brand new directory called **project2a**, and put all your code there. When you are done, check in and push your code to the repository. You should only check in source files, Makefile but not binary or object files.
Separately, create a tar-ball of your submission. This can be done by going to the directory above project 1, and then typing in the Linux command "`tar zcvf project2a.tar.gz project2a`". Upload this tar-ball onto the Canvas submission site for project 2. The submission of this tar ball will trigger execution of an auto-grader. We encourage you to start the project early so that you can validate your output against our auto-grader.

2.2 Part 2B: Two-Stage Pipeline

Pipes are another form of redirection, but instead of redirecting to a file, a pipe connects the standard output of one program to the standard input of another. Again, this is best demonstrated via example.

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

Like before, the `head` program will read the specified number of random bytes from `/dev/urandom`, but instead of redirecting the output to a file, it is *piped* to the `base64` program. In this assignment **you are required to implement a two-stage pipeline**, *i.e.*, one process can pipe output to the input of another process. A good example of a two-stage pipeline would be:

```
penn-sh> echo I have one brain > test
penn-sh> cat test | grep brain
I have one brain
penn-sh>
```

In the above example, we use `echo` to create the file `test` and write the sentence "I have one brain" to it. For the two-level pipeline command, the output of the `cat` (i.e. the content in "test"), is sent to the `grep` command, which prints out the sentence that contains the word "brain".
Notice that when we pipeline 2 commands, they actually run concurrently. The reason why it looks like the `cat` command runs first and then `grep` command begin running, is because `grep` is waiting for the input generated by `cat`. If we take the following command:

```
penn-sh> sleep 5 | sleep 5
penn-sh>
```

It should only sleep for 5 seconds instead of 10 seconds.

2.2.1 `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 programs in the pipeline. Be sure to close the file descriptor that is unused both in parent and child process. Otherwise your pipe will not work. **The manual page for `pipe(2)` is *very* helpful.**

2.2.2 Clean up zombies

With multiple processes executing at the same time, it is very likely that they terminate at different time. For example,

```
penn-sh> sleep 1 | sleep 10
penn-sh>
```

Here the first process terminates after 1 second, while the second needs 10 seconds. We need to ensure that both processes terminate and have been waited before we print out the next prompt. Otherwise, they will become zombies[1]. In order to make sure processes are terminated or stopped, we need to use the macros: For `WIFEXITED(status)` and `WIFSIGNALED(status)`, we can check if the process is terminated. Refer to the man page of `wait(2)` and `waitpid(2)` for more details.

2.2.3 Pipes and Redirections and Errors

It is possible to mix redirection symbols with pipes, however there are possible invalid inputs. For example:

```
penn-sh> cat /tmp/myfile > new_tmp | head > output
Invalid output redirection
penn-sh>
```

makes no sense, as there are two directives for redirecting standard output of `cat`. However, the command below is allowed because there are no overlapping redirections.

```
penn-sh> cat < pennshell | head > output
penn-sh>
```

2.2.4 Submission

Develop your code in the Git repository we assigned to you in project 1. However, create a brand new directory called **project2b**, and put all your code there. When you are done, check in and push your code to the repository. You should only check in source files and Makefile but not binary or object files.
Separately, create a tar-ball of your submission. This can be done by going to the directory above project 2, and then typing in the Linux command "`tar zcvf project2b.tar.gz project2b`". Upload this tar-ball onto the Canvas submission site for project 2. As before, the submission of this tar ball will trigger execution of an auto-grader.

[1] mmmm... brains!

3 Guidelines

3.1 Error Handling

All system calls that you use will report errors via the return value. As a general rule, if the return value is less than 0, then an error occurred and `errno` is set appropriately. **You must check your error conditions and include the key word "invalid" in the error output**. For example:

```
if(fork() < 0)
{
  perror("invalid fork")
}
```

To expedite the error checking process, we allow you to use `perror(3)` library function. Although you are allowed to use `perror`, it does not imply you should report errors at an extreme verbosity. Instead, try and strike a balance between sparse and verbose reporting.

3.2 Code Organization

Sane code organization is critical for all software. You code should adhere to DRY (Don't Repeat Yourself). If you are writing code that is used in more than one place you should write a function or a macro.
Also you should organize the 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/libraries that you can easily reference and include in your build process.

3.3 Memory Errors

You are required to check your code for memory errors. This is not only a nontrivial task, but also an extremely important one. Memory leaks will cause your computer to slowly run out of memory and may cause programs to crash. Fortunately, there are very nice tools like Sanitizers and `valgrind` that is available to help you. These two tools can help you identify leaks, but you still need to find and fix any bugs that these tools locate. There is no guarantee it will find *all* memory errors in your code, **especially those that rely on user input!**

4 Acceptable Library Functions

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

- `execvp(3)`
- `fork(2)`
- `wait(2)` or `waitpid(2)`
- `read(2)`, `write(2)`, `printf(3)`, `fprintf(3)`, and `sprintf(3)`
- `signal(2)` or `sigaction(2)`
- `kill(2)`
- `exit(2)` or `exit(3)`

- `dup2(2)`
- `pipe(2)`
- `open(2)`
- `close(2)`
- `malloc(3)` or `calloc(3)`
- `free(3)`
- `perror(3)`
- `atoi(3)` and `itoa()`
- `String.h`

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.**

5 Developing Your Code

We highly recommend you use the course-standardized virtual machine given by course staff because all grading will be done on the course virtual machine. Please do not develop on OSX as there are significant enough differences between OSX and Unix variants such as Ubuntu Linux. If you decide to develop on a different machine anyway, you must compile and test your penn-sh on the course-standardized virtual machine to ensure your penn-sh runs as you expect.

This is a large and complex assignment, using arcane and compactly documented APIs. We do not expect you to be able to complete it without relying on some outside references. That said, we do expect you to struggle a lot, which is how you will learn about systems programming, by *doing things yourself.*

Your programs will be graded on the course-standardized virtual machines, and must execute as specified there. Although you may develop and test on your local machine, you should always test that your program functions properly there.