CSCE 451/851 Programming Assignment 2
Writing a Unix Shell

Interprocess Communication via Pipes

IMPORTANT PRELIMINARIES!!!

- **DO NOT PUT THE FORK() STATEMENT INSIDE AN INFINITE LOOP!!**
  - This means no `while(1),` or equivalent **ANYWHERE** in your code.
  - use a bounded `for` loop instead, for example
    `for (int should_run = 0; should_run < 25; should_run++)`
  - You may safely assume that files are not infinite, and therefore, you can read lines of a file until the file ends. For example
    `while(getLine(file, inputString) != false)`
  - If we find an infinite loop this in submitted code you'll get a 0.
- The program should exit on `exit` command.
- Finally, you **may not** use the `system()` function.

---

1 Overview of Project (PA1 and PA2)

This project consists of writing a C (or C++) program to serve as a shell interface that accepts user commands and executes each command in a separate process. A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command: `cat prog.c.` (This command displays the file prog.c on the terminal using the Unix `cat` command.)

```
osh> cat prog.c
```

The above is an example of a simple command (i.e., it does not contain any operators). We will extend this program to execute more complex commands which contains one or more simple command connected together by an operator such as:

```
osh> cat < code.c
osh> cat < code.c > out
osh> ps | cat > out
```

Your shell will support the following operators (much like the standard `csh` or `bash` shells in Unix).

- `>` - Redirect stdout to a file
- `<` - Redirect stdin from a file
- `>>` - Append stdout to a file
- `&&` - Execute next command only on success.
- `||` - Execute next command only on failure.
- `;` - Execute next command regardless of success or failure.
- `|` - Pipe stdout of one command into stdin of next.

The functionality defined here for the above operators are basic, and may slightly deviate from the functionality defined by the standard shells available on Unix. But it is not the purpose of this project to write a fully functional Unix shell. Rather, it is to introduce and familiarize ourselves with various concepts useful in understanding operating systems and systems programming such file I/O, processes control, and inter-process communication. As such, we will only code parts of the shell that help introduce these topics.

PA1 This project is broken up into two programming assignments. In PA1 you will add support for:

- Input parsing
- `fork` and `exec` to create new processes
- All logical operators listed above except `|` (pipes)

PA2 In PA2, you will build upon PA1 and add support for:

- Interprocess communication via pipes

2 Submission

Use web handin to hand in your assignment. Submit a single zip file, `<UNL username>_<pa#>.zip` (e.g., `jdoe2_pa1.zip`) containing **only**:

- source files
- `makefile` with targets:
    - `all`: this must compile your `osh` binary and must be the first target in your `makefile`.

Executing `make` in your project directory should produce an executable named `osh`. Remember to verify that your code compiles and runs on the CSE servers.

3 PA2: Evaluation and Points Distribution

The PA2 zip contains 4 test scripts, alongside the corresponding expected output files. The table describes the test case, and the points awarded for each test case.

File	Answer Script	Test Case Description	Points
6.singlePipe.txt	ea6.txt	Two command connected by a single pipe	25
7.moreLogical.txt	ea7.txt	Multiple commands connected by multiple logical operator (&&, ||, ;)	15
8.morePipes.txt	ea8.txt	Multiple commands connected by multiple pipes	25
9.simplePipeAndLogical.txt	ea9.txt	Multiple commands connected by pipe and logical operator	25
makefile		The program compiles successfully on command 'make'	10
Total			**100**

The test scripts can be used to test your program and are the test scripts we use for grading. A fully functional executable, `osh`, is included in the distribution zip file for this project. You can use it to see what your outputs should look like.

3.1 Critical "Gotchas" for grading

- **Your code MUST compile and run on the CSE server**. If it doesn't, you'll get a 0
- Ensure that the output binary produced by makefile is `osh`.
- Points will be awarded only if your output matches the expected output **exactly**. Please conform to the error strings as specified in the expected output for Malformed commands.
- **You MUST implement the '-t' option**. This was described in the PA1 handout. If you don't, you'll get a 0.
- Although you need to implement the '-t' option, you still must have interactive shell capability. We will test that it functions as an interactive shell.
- The Yutaka Tsutano parser has an infinite loop in it, debug information in the output, as well as extra blanks in the output:
    - We will dock you points for having an infinite loop in your code, so take that out

- Your shell needs to work interactively as a shell without extraneous debug information... that is only printing command outputs and `osh>`
- Your output must match exactly... so no extraneous blank lines or spaces in the output

3.2 Grading Procedure

On our end, after running `make` to compile your program, we will run:

```
./osh -t < testscripts/testscript.txt > & tmp ; diff tmp testscripts/ea.txt ;
```

for each test script in the testscripts directory. Note that the "`> &`" redirects stderr to stdout in csh. If there are no differences you get all the points, otherwise we take points off.

Although you won't get points for these we will dock points for the following:

- Having an infinite loop in your code
- If your program doesn't run as an interactive shell
- If, when running as an interactive shell, you print extraneous output (outside `osh>` and the command output

4 PA2: Detailed Discussion and Description

This is the most challenging part of the shell project. We'll handle pipes in two steps. First we implement the logic to handle a single pipe, then extend it to handle any number of pipes in a command.

Pipes work similar to redirectors. But unlike a redirector, where the input/output is a file, pipe connects two commands. Consider the following command:

```
osh> ls | cat
```

This command will pass stdout from `ls` to stdin of `cat`. We could equivalently write this as,

```
osh> ls > tempfile; cat < tempfile
```

which consists of two steps:

- we overwrite stdout of `ls` to point to tempfile
- we overwrite stdin of `cat` to point to tempfile

But creating a file is expensive if its purpose is just to act as a temporary buffer. If we can hold this buffer in memory, it would be much more efficient. Interprocess communication (IPC) can help us do this. We will use pipes in this PA to accomplish this IPC. There is a system call `pipe()` that can be used:

`pipe()` - http://man7.org/linux/man-pages/man2/pipe.2.html

"pipes" are either bidirectional or unidirectional data channels, used for IPC. For this assignment we assume pipes are **unidirectional** and only pass information in one direction. For this project, using pipes boils down to 3 steps:

- create the pipe
- connect `stdout` of `ls` to one end of the pipe
- connect `stdin` of `cat` process to the other end of the pipe

The timing of doing all this is critical. The process image is copied upon `fork()` so to pass the file descriptors of the pipe along to the child, the pipe needs to be created **before** the fork. Once this is done, connect the pipes at the beginning of your loop (after `fork()` and before calling `exec()`). We can generalize this as follows,

- If the current command is connected to the next command by a pipe:

1. create a pipe
2. store the pipe handles at a temp location
3. connect stdout of the current process to the pipe
- If the current command is connected to the previous command by a pipe:
    1. connect the stdin of the current command to the pipe created earlier

Once these connections are made, the rest of the procedure for exec(), error handling, etc. remains unchanged. At a high level, your procedure should look like this:

```
prevpipe = null;
while(i<25) {
    get command()
    new process = fork()

    if (current command is connected to previous command by pipe) {
        // connect stdin of new process to prevpipe
    }

    if (current command is connected to next command by pipe) {
        prevpipe = pipe();
        // connect stdout of new process to prevpipe
    }

}
```

Remember that you still need to implement this in the context of forking the child processes and managing them. So think carefully about how to connect the pipes amongst the child processes when they're either the left command, right command, or middle command in left_cmd | middle_cmd | right_cmd. You should be able to handle commands with more than one pipe. It could be two pipes, or many more:

```
osh> ls | cat | cat
```

NOTE: In the case of a pipe, we do not wait for the current process to exit before starting the next one. Instead we wait for the last process in the chain of commands connected by pipe. When we create a pipe, a buffer of size PIPE_BUF will be allocated. When this buffer is full, it blocks the write (i.e., the current process writing to the pipe also blocks). So we will need another process at the other end reading from the pipe.

4.1 Tips

Processes reading from pipes will hang (i.e., not receive an EOF) until **all** write file descriptors to the pipe have been closed. Creating the pipes in the parent and forking two children will require careful closing of the file descriptors for pipes to work. To minimize open write pipe descriptors, follow the recommendations below.

- The pipes need to be kept open until the fork() in order to replicate the pipe file descriptors to the children
- After the first child is launched (writes to pipe) the parent should close the write end of the pipe before the second fork
- The first child should close the write end of the pipe after it's duped to stdout.