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 (at least) 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.malformed.txt	ea7.txt	Identify malformed commands	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 and is executable on the CSE servers. 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** (except for blank lines). 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 parser has debug information and extra blanks in the output, so you should adjust that accordingly:
  - Your shell needs to work interactively as a shell without extraneous debug information. . . that is only printing command outputs and `osh>`

- Note that even common programs behave differently on different OSs. That is, the behavior of `ls` on the CSE servers may not exactly match the behavior of `ls` on a Mac. So if you do your development on your local machine be sure you test it regularly on the CSE servers since some output may differ.
- Additionally, some commands have different behavior when outputting to an interactive shell vs. a file. For example, the `ls` command, in its documentation reads:

  > "If standard output is a terminal, the output is in columns (sorted vertically) and control characters are output as question marks; otherwise, the output is listed one per line and control characters are output as-is."

  So, running the commands from the testscripts interactively may produce outputs different than what's in the "ea*.txt" files. But your output should match exactly when run with the `-t` option.

3.2 Grading Procedure

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

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

for each test script in the testscripts directory. Note that the "`> &`" redirects stderr to stdout in csh. This redirection looks different if you're using a bash (i.e., `2>&1`) shell, but for grading we will use csh. If there are no differences you get all the points, otherwise we take points off according to the table above.

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 the first command (e.g., `ls`) to one end of the pipe
- connect `stdin` of the next command (e.g., `cat`) to the other end of the pipe

The challenge in this assignment is to get the timing of the connections correct. The process image is copied upon a call to `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(processing commands) {
    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 | grep foo
```

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 pipes. 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.

5 Tips

5.1 Tip #1

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 `dup()`-ed to stdout.

5.2 Tip #2

The best way I've found to approach this assignment is to start by thinking about it sequentially, with all the loops unrolled, outside of the parser and logical operator support in PA1. In other words, we want to simplify the task as much as possible and focus only on the pipes. Don't worry about reading commands from the command line, just hard code some commands for now. Once we have that working we can incorporate it into what we wrote for PA1.

Start by assuming we have the command sequence:

```
osh> ls | cat
```

To process these commands this I need 2 child processes, one for `ls` and one for `cat`. So write up some code to do this outside of PA1 and the parser:

```
// psuedocode for command "ls | cat"
int main() {
    int fdp[2] // file descriptor array for pipe
    pipe(fdp) // create pipe
    pid = fork() // fork the first child process for "ls"
    if (child) {
        close(fdp[READ_END]); // don't need the read end
        dup2(fdp[WRITE_END], STDOUT_FILENO) // use the write end
        close(fdp[WRITE_END]) // can close the write end now since it's STDOUT
        execvp("ls", args) // now exec the first command
    } else { // now the parent
        // close the write end of the pipe (remember it's a copy) cuz here we need
        // the read end
        close(fdp[WRITE_END])
        // at this point only the read end of the pipe is open in the parent
        pid = fork() // fork another child now for "cat"
        if (child) {
            // use the read end of the pipe for cat
            dup2(fdp[READ_END], STDIN_FILENO)
            close(fdp[READ_END]) // now can close read end since it's STDIN
            execvp("cat", args) // exec cat
            exit(0)
        }
        // finish work in parent by waiting
        wait(&status) // wait for a child
        wait(&status) // wait for other child
    }
}
```

Once you have this working try putting things into a loop in this manner:

```
// psuedocode for command "ls | cat"
int main() {
    int fdp[2] // file descriptor array for pipe
    pipe(fdp) // create pipe
    for(int i=0; i<2; i++) {
        pid = fork() // fork the first child process for "ls"
        if (child) {
            if (i==0) { // if first command "ls"
                // do all the stuff from above for "ls"
            } else { // if second command "cat"
                // do all the stuff from above for "cat"
            }
        } else { // this is the parent
            // do all the stuff for the parent
        }
    }
}
```

You should start seeing a lot of duplicated code that could be simplified and placed more intelligently. You should also start seeing there is a pattern for working with the pipes depending on where the current command is. The above pseudocode can work for commands of the form `osh> command1 | command2`. But what if I have `osh> command1 | command2 | command3`? You now need to start generalizing how to work with the pipes if it's a first command, middle command, or last command.

Let's continue with the example above but add a third command, and rethink the conditions for using the pipes:

```
// psuedocode for command "ls | cat | grep foo"
int main() {
    int fdp[2] // file descriptor array for pipe
    int initial_pipe_in_chain // have to save the first pipe if we have multiple pipes
    pipe(fdp) // create pipe
    for(int i=0; i<3; i++) { // now need 3 child processes
        pid = fork() // fork the first child process for "ls"
        if (child) {
            if (i==0) { // if first command
                // do all the stuff from above for a first command
            } else if (i==1) { // if middle command
                // This is the new case. Here, we have to work with two pipes.
                // Think carefully about how to connect the pipes to the command
                // and which ends to close.
                execvp("cat", args) // execute the middle command
            } else { // if last command
                // do all the stuff from above for a last command
            }
        } else { // this is the parent
            // if there are multiple pipes in the chain we need to save the first one.
            // but really we only need the read end
            initial_pipe_in_chain = fdp[READ_END]
            // do all the remaining stuff for the parent
        }
    }
}
```

At this point, depending on you implementation, hopefully it should be relatively straightforward to incorporate the above code into your existing PA1. There's lots of repeated code you can simplify, after which you can add a few conditions to determine whether you're working with the first command, middle command, or last command.

Incidentally, compilable versions of the first two set of pseudocode presented are in the "helpful_code" folder in the OneDrive class folder.