Writing a Unix Shell

Grab a goblet of your favorite beverage, this is going to be a long read

Index:

Important!!!

DO NOT PUT THE FORK() STATEMENT INSIDE AN INFINATE LOOP!!. This means no while(1), or equivalent ANYWHERE in your code.

- Do not use while(should_run) from the text
- use a bound for loop instead: e.g.: for (int should_run = 0; should_run < 25; should_run++){

The program should exit on 'exit' command.

1. Overview

It is not the purpose of this assignment to write a fully functional UNIX shell. Rather, the purpose of this programming assignment is to introduce and familiarize ourselves to various concepts useful in 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.

2. Assignment in brief

This project consists of designing a C(or C++) program to serve as a shell interface that accepts user commands and then 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.

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

This shell will support the following operators (Much like the standard 'csh' or 'tcsh' shells in UNIX).

'>' - redirect stdout to file
'<' - redirect stdin from file
'>>' - append stdout to file
'|' - pipe stdout of one command into stdin of next.
'**&&**' - Execute next command only on success.
'||' - Execute next command only on failure.
'**;**' - Execute next command regardless of success or failure.

3. Evaluation and Points Distribution:

In order to evaluate your program, the program should implement to handle a switch '-t'. When used, the program does not print the prompt to the console i.e. disable all print statements when the –t argument is given.

The test scripts are used to test the program. The test can be executed by issuing the command:

```
osh -t < testscript.txt
```

The test file contains 9 test scripts, and the expected answer scripts. The table describes the test case, and the points awarded for each test case.

Test File	**Answer Script**	**Test Case Description**	**Points***
9.malformed.txt	ea9.txt	Identify malformed commands	10
1.singleCommand.txt	ea1.txt	Simple command with arguments (no operators)	10
2.simpleRedir.txt	ea2.txt	Single command with single redirector (input or output <, >, >>)	10
3.moreRedir.txt	ea3.txt	Single command with multiple redirector (input and output <, >, >>)	5
4.logicalConditional.txt	ea4.txt	Two command connected by a single logical operator (&&, ||, ;)	10
5.singlePipe.txt	ea5.txt	Two command connected by a single pipe	10

6.moreLogical.txt	ea6.txt	Multiple commands connected by multiple logical operator (&&, ||, ;)	10
7.morePipes.txt	ea7.txt	Multiple commands connected by multiple pipes	10
8.simplePipeAndLogical.txt	ea8.txt	Multiple commands connected by pipe and logical operator	10
makefile		The program compiles successfully on command 'make'	10
README file		As described below	5

***Points will be awarded only if the output match completely. (Some leeway is allowed in error messages for malformed commands)**

A sample implementation of the shell named "osh" can be found on Canvas under "source" folder. The test scripts file and answer scripts can be downloaded from Canvas under test scripts folder.

4. Submission:

Use web handin to hand in your assignment. Submit a single zip file, <cse-accountname>_pa1.zip containing the

- source files
- makefile
- README

The name of the executable produced by 'make' command must be 'osh'

Include a README file in with your code. Use the README file to:

- Document instructions to compile, run, and test your program.
- Document any problems you had.
- Include a paragraph on what you learned in doing this programming assignment.

Like any other programming assignment of this course, your program must be verified to run on cse.unl.edu.

Due Date: Monday February 12th at 11:59PM

5. Detailed Discussion and Description

This assignment is based on programming assignment *Project 1 (part I)* which is described on pg 157 of the text. We will not implement *Part II* of the project in the text. Instead we will extend the shell to support the file redirections, IPC (pipes) and command combinations described above.

This assignment may seem daunting, so to help, it is broken into four phases.

5.1. Phase 1 - Parsing the Input String

The first step a shell should do is to parse the input line and figure out what it means. Mainly,

- identify the name of program
- supply the correct arguments to the program
- check if input or output needs to be redirected
- check if the output/input redirected from/to a file or another program
- check if there are any logical operator, which control the execution of programs
- repeat the same step for each command in the input

In order to achieve this, we need to

- correctly parsing the input line, by separating each token
- identify separate commands, and arguments for each command from tokens
- building a data structure to represent the parsed data
- save additional information (as to how it interacts with the next/previous command). This could be saving file handles, logical operator etc.

With this data structure, we should be able traverse the list with ease. Easily retrieve arguments, or skip to next commands etc.

We extend this phase to identify malformed commands. Since we are able to traverse the command structure, we should be able to analyze which command, operator combinations make sense, and which don't. Mainly identify if

- we have a null command. Eg. osh > cat | | file
- if there is no file after a redirector symbol. eg. osh > echo >

Examples for malformed commands are shown in 9.malformed.txt test file.

5.2. Phase 2 - Forking a child process and executing a command

After building the data structure, start by executing simple commands (and ignore the operators for time being). Then extend the program to handle file redirection. This would involve

- extracting a simple command from the data structure (executable name and arguments)
- creating a new process for the executable mentioned in the command
- supplying correct arguments to the newly created process

- wait for the process to complete, collect exit code
- output the result to console (as of now, later we'll modify to handle redirector)

We will use the following api

- fork()
- exec()*
- wait()

*You will have to decide on the appropriate exec function to use among the choices. You do not have to search for the executable. The exec function will do it for you. Choose wisely.

Now is the tricky part. Next modify the program to handle redirection operators. By default a program writes the output generated to stdout. stdout points to console by default. In order to write to a file, you need to overwrite stdout to point to file handle. In order to do this, we modify the program as follows,

- when we read the command, check for the input/output redirector
- if so, get the file name
- open the file with appropriate option (new file vs. append)
- after the new process is created, and before doing exec(), overwrite the stdin/stdout

In order to overwrite the stdin/stdout, we use the following api

- dup2()

Here is a nice little article about dup2.

As a final step, you can add a loop to execute each command in the input line as separate command (ignore the functionality of logical operators and pipe)

5.3. Phase 3, Logical operators

Handling logical operators is super easy. The execution of the command is determined by the exit status of the previous command. In the previous section, we implemented the logic to collect the exit status. Let us break it down further,

- check if the previous command has any logical operator
- if it does, get the exit status of the previous command
- determine if you need to execute current command or skip it
- if you need to skip it, then set the exit status of the current command to be the same as previous command

The above logic, is sufficient to handle chains of logical operators.

5.4. Phase 4, Pipes

This is the hard part, where you have to slay the dragon and rescue the Prince/Princess. 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.

If you remember, a pipe works similar to a redirector. But unlike a redirector, where the input/output is a file, pipe connects two commands. So what does this mean? One way to think about it is that it is a concatenation of both input and output operator. Consider the example,

```
osh> ls | cat
```

we can equivalently write this as,

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

If you examine closely, there are two steps

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

But creating a file is expensive if its purpose is to just act as a temporary buffer. Rather if we can hold this buffer in memory, it would be much faster. This is where we use one of the IPC mechanisms mentioned in the text, pipes.

`pipe()`

pipe is a unidirectional data channel, used for IPC. Please go through the presentation to understand more about pipes. The presentation can be found on canvas under help files folder with filename pipes.ppt .

Thus handling pipe changes to a 2 step procedure (assuming that you have created a pipe)

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

This is done 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 next command by pipe, create a pipe, store pipe handles at a temp location, connect the stdout of current process to pipe

- If the current command is connected to previous command by pipe, then connect the stdin of current command to the pipe (which was created earlier)

Once these connections are made, rest of the procedure remains unchanged. At a very high level, your procedure looks like

```
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
        }

        // rest of code
}
```

NOTE: As you can see, in case of pipe, we do not wait for 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.

Finally, we will look at handling commands with more than one pipe. It could be two pipes, or many more.

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

Well, the good news is, if you implemented the previously described algorithm properly, you don't need to do anything more. It should just work.

6. Implementation Notes

This section contains some notes, and pseudocode to help with the implementation. We will present it using the same approach as section 5.

6.1. Phase 1 - Parsing the Input String

There are two ways to implement this phase.

- Use parse.cpp

- Implement your own parser

parse.cpp

This is a helper file that does the tedious steps of parsing the input line, and puts it in a doubly linked list. This is the easiest and quickest way to implement phase 1. Additional details of what the file does, and how to use it can be found on canvas in "man_parse.txt" help files folder, and the source file on canvas in "parse.cpp" in source files folder. It is not required to understand the implementation of parse.cpp, you only need to understand how to use the parser. Hence it is not necessary to read the parser. **The parse.cpp does not check for malformed commands.** You will have to make sure your code checks for malformed commands.

Alternatively, the TA Yutaka has provided his own version of the parser available on github: https://github.com/ytsutano/osh-parser . Feel free to use this instead of parse.cpp

Implement your own parser

If you wish to implement without any help from scratch, you can do so. This is probably hardest of the two approaches.

6.2. Phase 2 - Forking a child process and executing a command

In this phase, fork() and exec() the commands with their respective arguments, ignoring file redirects and pipes (treat pipes as ";").

Create a child process to launch the executable. As covered in class you must use fork to exec a new program, else the shell will be replaced with the exec command and cease to exist. Shown below is pseudo code to accomplish this. Note the four rules when using fork.

1. Fork is not placed in a while(1) loop.
2. We trap and exit if the fork fails.
3. We always have an exit in the child (NOT Shown in Fig 3.9 of the text).
4. Have a wait in parent (unless we have multiple exec's - i.e. pipes).

You have a choice of six exec functions, execl(), execlp(), execle(), execv(), execv(), execvp(), execve(). You are restricted by two facts, first you do not know ahead of time how many arguments you will have and second our shell does not maintain a hash table of executables and their locations. We would instead like the exec function to find the executable for us (Very unshell like). What all this means is that only one of the above exec function will work.

Pseudocode

```
/* Rule 1: Dont forget this should not go inside a while(1)
loop anywhere in your program */

pit_t cpid = fork();
if (cpid < 0) {    /* Rule 2: Exit if Fork failed */
        fprintf(stderr, "Fork Failed \n");
        exit(1);
}

else if (cpid == 0) {

        execlp(....);
        fprintf(stderr, "Exec Failed \n");
        exit(1)  /*Rule 3: Always exit in child */
}

else {

        int status;
        wait(&status);     /*    Rule 4: Wait for child unless
                                  we need to launch another exec */
        printf("Child caught bla bla bla\n");
}
```

6.3 Phase 4, Pipes

Processes reading from pipes will hang (not receive an EOF) until ALL write file descriptors to the pipe have been closed.

To understand how creating the pipes in the parent and forking two children will require careful closing of various file descriptors for pipes to work. To minimize open write pipe descriptors, follow the recommendations below.

- The pipes need to be kept open till the fork() 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.

You can access a sample program on cse by executing osh

7. Additional Resources

Presentation on Pipes: pipes.ppt on Canvas under help files folder

Presentation on assignment (from a previous semester): programmingassignment1.pptx on canvas under help files folder

Beej's guide: http://beej.us/guide/bgipc/

PA0: A document prepared by the previous TA "PA0.docx" is provided on Canvas under help files folder. This helps you with revising the pre-requisites for this class. Although it is not required to read this document it will help you if you need guidance on some of the basics required for PA1.