Home

Homework 1: OS interface and shell

This assignment will make you more familiar with the Unix system call interface and the shell by implementing several simple programs and several features in a small shell, which we will refer to as the 238P shell. You can do this assignment on any operating system that supports the Unix API (Linux andromeda-XX.ics.uci.edu machines, your laptop that runs Linux or Linux VM, and even MacOS, etc.). Submit your programs and the shell through Canvas (see instructions at the bottom of this page).

First, you have to read the Chapter 0 of the xv6 book.

Part 1: Simple UNIX programs

Download the main.c, and look it over. This is a skeleton for a simple UNIX program.

To compile `main.c`, you need a C compiler, such as gcc. On andromeda-XX.ics.uci.edu (Openlab machines), you can compile the skeleton as follows:

```
$ gcc main.c
```

which produces an `a.out` file, which you can run:

```
$ ./a.out
```

Alternatively you can pass an additional option to gcc to give a more meaningful name to the compiled binary, like

```
$gcc sh.c -o foobar238p
```

Here gcc will compile your program as `foobar238P`. In the rest of this part of the assignment you will convert `main.c` into several simple UNIX programs.

File copy command (cp238p)

Use the main.c template as a starting point for a simple file copy command that you should implement. First copy the `main.c` into `main-cp238p.c` (you will need to use `main.c` for other programs later, so lets keep it around).

The file copy command should take two arguments: names of the input and the output files and copy the input file into the output file. Here is an example invocation which copies `main.c` into `main-out.c` (assuming you call your executable `cp238p`).

```
cp238p main.c main-out.c
```

You should use `read()` and `write()` system calls to read the input file and write the output. Since `cp238p` takes command line arguments you should change the definition of the `main()` function to allow passing of command line arguements like:

```
int main(int argc, char *argv[])
```

If you have never worked with command line arguments in C here is a link that might be useful: Arguments to main. You can also take a look at a couple of user-level programs that take command line arguments from the xv6 source tree: rm.c, ls.c, wc.c.

Note: You might find it useful to look at the manual page for `read()`, `write()`, and other system calls. For example, type

```
 $man read
```

and read about the `read` system call. Here the manual says that you should include

```
#include <unistd.h>
```

in your program to be able to use it, and the system call can be called as a function with the following signature

```
 ssize_t read(int fd, void *buf, size_t count);
```

The manual describes the meaning of the arguments for the system call, return value, and possible return codes. Finally, it lists several related system calls that might be helpful.

Note that when the manual list a function like `open(2)` it means that it's described in the 2nd section of the manual and to get to the specific section you have to invoke man with an additional argument like this:

```
man 2 open
```

It's a good idea to read the man entry on man itself, i.e.,

```
man man
```

. Some useful commands are `-k` to search the manual for the string matching a query:

```
man -k open
```

Note, that here there are multiple entries for the `open()` system call and default invocation of

```
man open
```

will return an entry for the `openvt` command, and not file open command.

Simple I/O redirection (lsy238p)

Use the `main.c` template again as a starting point for another simple program that starts `ls` command but redirects its output into a `y` file. I.e., your program should do an equivalent of this shell command

```
ls > y
```

Internally your program should start `ls`, but before doing this it should arrange that output of the ls is redirected into a file. Note, you don't have to implement `ls` itself, just start the one that is already there in the system with the `exec()` system call.

First copy the `main.c` into `main-lsy238p.c` (again you will need to use `main.c` for other programs later).

Here is how an example invocation of your program should look (assuming you call your executable `lsy238p`).

```
lsy238p
```

You should use

```
exec()
```

system call to start ls and use other system calls required to implement redirection, e.g., `close()`, `open()`.

Simple pipes (pipe238p)

Use the main.c template again as a starting point for another simple program that starts `ls` command but redirects its output into the `grep "main"` program, which itself redirects its output to the `wc`. I.e., your program should internally start three programs connected into with pipes that produce output equivalent of this shell command

```
ls |grep "main" |wc
```

Internally your program should start three new programs: `ls`, `grep "main"`, and `wc` and connect them with pipes.

Copy the `main.c` into `main-pipe238p.c`. Here is an example invocation of your program (assuming you call your executable `pipe238p`).

```
pipe238p
```

You should use `exec()` and `fork()` system calls to create programs, `pipe()` system call to create pipes, and other system calls required for connecting pipes, e.g., `close()`, `dup()`.

Part 2: Building a shell

Now you are ready to integrate the basic skills that you've gained in the first part of the assignment into a more general program that implements I/O redirection, the shell. If you are not familiar with what a shell does, do the Unix hands-on from 6.033 class at MIT (this is optional and will not be graded in 238P).

Download the 238P shell, and look it over. The 238P shell contains two main parts: parsing shell commands and implementing them. The parser recognizes only simple shell commands such as the following:

```
ls > y
cat < y | sort | uniq | wc > y1
cat y1
rm y1
ls |  sort | uniq | wc
rm y
```

Cut and paste these commands into a file `t.sh`

To compile `sh.c`, you need a C compiler, such as gcc. On andromeda-XX.ics.uci.edu (Openlab machines), you can compile the skeleton shell as follows:

```
$ gcc sh.c
```

which produces an `a.out` file, which you can run:

```
$ ./a.out < t.sh
```

This execution will print error messages because you have not implemented several features. In the rest of this assignment you will implement those features.

Alternatively you can pass an additional option to gcc to give a more meaningful name to the compiled binary, like

```
$gcc sh.c -o sh238P
```

Here gcc will compile your shell as `sh238P`.

Executing simple commands

Now, you're ready to work on the homework itself. First, extend your shell to implement simple commands, such as executing external programs, for example ls:

```
$ ls
```

Here you tell the shell to execute ls.

In the sh.c, the parser already builds an `execcmd` for you, so the only code you have to write is for the ' ' case in `runcmd`. **At a high level you should understand a typical UNIX interface that we've discussed in class (the functions to cone processes, i.e., fork(), executing new processes, i.e., exec(), working with file descriptors (close(), dup(), open(), wait(), etc.). Combine these functions to implement various shell features.**

You might find it useful to look at the manual page for exec, for example, type

```
$man 3 exec
```

and read about `execv`. Print an error message when exec fails.

To test your program, compile and run the resulting a.out:

```
$./a.out
```

This prints a prompt and waits for input. `sh.c` prints as prompt `238P$` so that you don't get confused with your computer's shell. Now type the following in your shell:

```
238P$ ls
```

Your shell may print an error message (unless there is a program named `ls` in your working directory or you are using a version of `exec` that searches `PATH`, i.e., execlp(),

execvp(), or execvpe()). Now type the following:

```
238P$ /bin/ls
```

This should execute the program `/bin/ls`, which should print out the file names in your working directory. You can stop the 238P shell by typing ctrl-d, which should put you back in your computer's shell.

You may want to change the 238P shell to always try `/bin`, if the program doesn't exist in the current working directory, so that below you don't have to type "/bin" for each program, or (which is better) use one of the exec functions that search the `PATH` variable.

I/O redirection

Implement I/O redirection commands so that you can run:

```
echo "238P is cool" > x.txt
cat < x.txt
```

The parser already recognizes ">" and "<", and builds a `redircmd` for you, so your job is just filling out the missing code in `runcmd` for those symbols. You might find the man pages for open and close useful.

Note that the `mode` field in `redircmd` contains access modes (e.g., `O_RDONLY`), which you should pass in the `flags` argument to `open`; see `parseredirs` for the mode values that the shell is using and the manual page for `open` for the `flags` argument.

Make sure you print an error message if one of the system calls you are using fails.

Make sure your implementation runs correctly with the above test input. A common error is to forget to specify the permission with which the file must be created (i.e., the 3rd argument to open).

Implement pipes

Implement pipes so that you can run command pipelines such as:

```
$ ls | sort | uniq | wc
```

The parser already recognizes "|", and builds a `pipecmd` for you, so the only code you must write is for the '|' case in `runcmd`. You might find the man pages for pipe, fork, close, and dup useful.

Test that you can run the above pipeline. The `sort` program may be in the directory `/usr/bin/` and in that case you can type the absolute pathname `/usr/bin/sort` to run sort. (In your computer's shell you can type `which sort` to find out which directory in the shell's search path has an executable named "sort".)

From one of the andromeda machines you should be able to run the following command correctly (here a.out is your 238P shell):

```
$ a.out < t.sh
```

Don't forget to submit your solution through Canvas Canvas HW1 OS Interface and Shell (as a collection of source files "main-cp238p.c", "main-lsy238p.c", "main-pipe238p.c", and "sh.c"). If you decide to submit a challenge exercise submit an extra file "sh-extra.c", and a shell script "extra.sh" that contains an example extra command that your shell can handle as a single tar or zip archive. Please write us a comment at the top of "sh-extra.c" explaining which extra features you decided to handle.

Challenge exercises (extra 21%, 7% each)

You can add **any** feature of your choice to your shell. But, you may want to consider the following as a start:

- Implement lists of commands, separated by ";"
- Implement sub shells by implementing "(" and ")"
- Implement running commands in the background by supporting "&" and "wait"

All of these require making changes to the parser and the `runcmd` function.

Updated: October, 2018