Homework-1:

Submission Deadline: 27th January, 3PM

Part-1: Shell

This assignment will make you more familiar with the Unix system call interface and the shell by implementing several features in a small shell.

Download the shell, and look it over. The 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.

Assuming you have gcc, you can compile the skeleton shell as follows:

```
$ gcc sh.c
```

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

```
$ ./a.out < t.sh
or,
$ ./a.out will give you an interactive shell
```

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

1.A. Executing simple commands

Implement simple commands, such as:

```
$ ls
```

Coding-1: The parser already builds an `execcmd` for you, so the only code you

have to write is for the ' ' case in `runcmd`. You might find it useful to look at the manual page for exec; 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 `OS$` so that you don't get confused with your computer's shell. Now type to your shell:

```
OS$ ls
```

Your shell should print an error message (unless there is a program named `ls` in your working directory). Now type to your shell:

```
OS$ /bin/ls
```

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

Coding-2: Change the 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.

Optional: If you are ambitious you can implement support for a `PATH` variable.

1.B. I/O redirection

Coding-3: Implement I/O redirection commands so that you can run:

```
echo "This is OS course" > 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

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).

1.C. Implement pipes

Coding-4: 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".)

Now you should be able the following command correctly:

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

Make sure you use the right absolute pathnames for the programs.

Submission

Submission is to be done on the moodle at the appropriate link. Just upload your sh.c file, no other file needs to be uploaded with this.

Make sure that your code compiles and works without any special switch being used with gcc. e.g. `gcc sh.c` should not give any error.

Note

You completed the design of a basic Unix shell. Supporting any of the below features will require changes to be done in the parser as well as the `runcmd` function. We leave this to the interest of students for further exploration.

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

Part-2: Syscall

This homework requires downloading the xv6 learning OS.

```
$  wget www.cse.iitd.ac.in/~kedia/os/xv6.tar.gz
$  tar -zxvf xv6.tar.gz
```

```
$  cd xv6
```

Booting xv6:

Similar to our JOS, you need to type the below commands to boot xv6:

```
$  make
$  make qemu
```

Coding-1: System call tracing

Your first task is to modify the xv6 kernel to print out a line for each system call invocation. It is enough to print the name of the system call and the return value; you don't need to print the system call arguments.

When you're done, you should see output like this when booting xv6:

```
...
fork -> 2
exec -> 0
open -> 3
close -> 0
$write -> 1
 write -> 1
```

That's init forking and execing sh, sh making sure only two file descriptors are open, and sh writing the $ prompt.

Hint: modify the syscall() function in syscall.c.

Coding-2: Date system call

Your second task is to add a new system call to xv6. The main point of the exercise is for you to see some of the different pieces of the system call machinery. Your new system call will get the current time and return it to the user program. You may want to use the helper function, `cmostime()` (defined in `lapic.c`), to read the real time clock. `date.h` contains the definition of the `struct rtcdate` struct, which you will provide as an argument to `cmostime()` as a pointer.

You should create a user-level program that calls your new date system call; here's some source you should put in `date.c`:

```
#include "types.h"
#include "user.h"
#include "date.h"

int
```

```
main(int argc, char *argv[])
{
  struct rtcdate r;

  if (date(&r)) {
    printf(2, "date failed\n");
    exit();
  }

  // your code to print the time in any format you like...
  /* example output format should be like this:
     Year: 2016
         Month: 1 or January
         Date: 26
         Hour: 15
         Minute: 12
         Second: 11

  */


  exit();
}
```

In order to make your new `date` program available to run from the xv6 shell, add `_date` to the `UPROGS` definition in `Makefile`.

Your strategy for making a date system call should be to clone all of the pieces of code that are specific to some existing system call, for example the "uptime" system call. You should grep for uptime in all the source files, using `grep -n uptime *.[chS]`.

When you're done, typing `date` to an xv6 shell prompt should print the current time.

Coding-3: dup2() system call.

Read Linux man pages for description on dup2 system call. xv6 doesn't implement the dup2 system call by default and you are required to implement dup2 as a part of this assignment.

The function prototype for dup2 is as below:

```
int dup2(int oldfd, int newfd);
```

The difference from the native dup call is that instead of allocating the first available file descriptor, the user can specify through newfd which exact file descriptor should be used.

Hint: Look into the sys_dup function in the file syscall.c

Similar to date syscall, you can check the functioning of dup2 system call by creating your own user program and making it available to the xv6 shell.

Submission

Submission is to be done on the moodle at the appropriate link. Just bundle the modified files as per below instructions.

Follow the below instructions

```
make clean
tar -zcvf xv6.tar.gz * --exclude .git
base64 xv6.tar.gz > xv6.tar.gz.b64
```

Upload the file named xv6.tar.gz.b64