Programming Assignment #2

Due: **4th Nov.** (Thu.), **11:59 PM**

1. Introduction

In this assignment, you will make a *swsh* shell which works under Linux environment. The main goal of this assignment is to make a program which covers the files, processes, signals, and IPC topics learned in the first half of the class.

2. Problem specification

A shell is a program that takes *1) input from a user*, *2) interprets it*, and *3) executes a command*. *swsh* is a shell that implements only a small amount of functionality and can execute specific commands. It supports input / output redirection and pipeline functions.

The following table describes the strings that can be input by the user in a form similar to BNF (Backus-Naur Form). (In the following, we call it <table>)

```
input         ::=   commands
                    commands | input
commands      ::=   command
                    command < [filename]
                    command > [filename]
                    command >> [filename]
                    command < [filename] > [filename]
command       ::=   cmd_type1 [options/arguments]
                    cmd_type2 [options/arguments]
                    cmd_type3 [arguments]
                    cmd_type4
cmd_type1     ::=   { ls, man, grep, sort, awk, bc }
                    path
path          ::=   { pathname with leading "./" }
cmd_type2     ::=   { head, tail, cat, cp }
cmd_type3     ::=   { mv, rm, cd }
cmd_type4     ::=   { pwd, exit }
```

Some expressions used in <table> can be interpreted in their literal meaning. In short, filename means any [filename], and options / arguments is the program's [options and arguments]. Also, some of the symbols used in <table> are the same as those used in bash.

|	Pipeline
<	Input redirection
>	Output redirection
>>	Output redirection (appending)

The function of each symbol is the same as what we learned in the Pipe class.

- Pipeline: Connect the standard output of the preceding process through a **pipe** to the standard input of the later process
- Input redirection: Replace the standard input (fd 0) of the process with the file [filename]
- Output redirection: Replace the standard output (fd 1) of the process with the file [filename]
  - If redirected by >, the file is initialized and rewritten.
  - If redirected by >>, append from the end of the file.

Also, for convenience of implementation, it is assumed that **each keyword of <table> is separated by a space**. If the input is "command < filename", there is a space between the [command] and the '<' characters, and a space between the '<' and [filename].

2.1. Interpreting inputs

The user enters a string of up to 200 bytes. If the user's input is a string that can be derived from the input of <table>, then the input is valid. For example, for the following command:

```
cd dir3
```

It is valid because it can be interpreted in following order.

- `input` ➔ `commands` ➔ `command` ➔ `cmd_type3 [arguments]`
  - `cmd_type3` ➔ `cd`
  - `[arguments]` ➔ `dir3`

The following commands are also valid because they can be derived from input.

```
ls -al /etc | sort -r > ls_sorted.txt
```

However, the following command is not valid because the program less cannot be derived from input.

```
ls -al /etc | less
```

In the case of path in <Table>, this means that the shell will execute the program in the current directory. When executing the *a.out* file in the current directory, use the following command.

```
./a.out
```

It means, if [command] is a string (path) starting with "./", it can be derived from cmd_type1.
(For simplicity, we don't consider absolute paths. Also, don't use '~' which means home directories.)

If the input is valid, the input is evaluated. If it is invalid, the following error message is output to standard error (fd 2).

```
swsh: Command not found
```

2.2. Evaluating inputs

There are many ways to evaluate the input, but if you don't have a specific idea, here's how:.

1) Determine the number of commands (when commands are connected by pipeline, etc.)
   (For each command)
2) Determine if input / output redirection was used
3) Analyzes the command type,
   A. If you implement it yourself, create a child process (if necessary) to call that routine.
   B. If you not implemented it yourself, create a child process to load and run the program.
4) If you create a child process, wait until it terminates

2.2.1. Input/output redirection

When redirection is used as a shell command, it is necessary to check whether the file specified by the user exists or if the file exists even if it is a normal file (not a directory), and whether the user has permission to use the file. However, *swsh* only detects the following cases:

If the file specified by input redirection exists, it is a normal file with read permission. If it does not exist, the following string is displayed.

```
swsh: No such file
```

A file designated as output redirection is assumed as a file that does not exist or has write permission. (You don't need to check for errors)

Pipelines and redirection are conflicting in that they switch standard I / O. Therefore, when a user uses a pipeline, redirection is assumed to be limited. For example, suppose the four commands are piped as follows:

```
A | B | C | D
```

Input redirection can only exist in the first process A and output redirection can only exist in the last process D. (You don't need to check if redirection exists in B and C)

```
A < file1 | B | C | D > file2
```

2.2.2. command

swsh has four types of command to be processed.

- `cmd_type1:` Programs that need to be loaded (fork-exec *) and run
- `cmd_type2:` Program to be loaded or to be implemented
- `cmd_type3:` Program to be implemented (with arguments)
- `cmd_type4:` Program to be implemented (without arguments)

The command corresponding to cmd_type1 extracts options or arguments and creates a child process. Then, loads and executes the binary through the exec family of functions.
The command corresponding to cmd_type2 can be loaded from existing command using exec functions or can be implemented by your own. **If you implement the command, you will get higher score.** In addition to inputting arguments, **program should be developed considering connection with other process through pipe or redirection.**
The commands corresponding to cmd_type3 and cmd_type4 must be implemented directly.

For the commands that need to be implemented, Chapter 3 below defines the function of each command and mentions the system call / library functions required.

2.3. Process group

If *swsh* creates a new child process, make sure that it belongs to the new process group. (Makes the pgid of that child process match its own pid)
If you create more than one process due to the use of pipes, make sure all processes belong to the group of processes you created first. (The pgid of the process must match the pid of the first child process created)

2.4. Reaping child processes

If *swsh* has created a child process, use the wait series system call to identify and remove the child process so that no zombie processes are created.

If a process receives a SIGTSTP signal generated by the Ctrl + Z key during execution, it will stop and halt the execution of the process as usual. At the same time, the parent process receives the SIGCHLD signal and returns true if the parent executes the WIFSTOPPED macro on the status value obtained by calling waitpid as follows: (option WUNTRACED of the waitpid system call)

```
waitpid(-1, &status, WNOHANG | WUNTRACED)
```

($ man 2 waitpid)

If a child process is stopped (when WIFSTOPPPED is true), it kills all associated processes by sending a SIGKILL signal to **the process group** to which the child belongs.

2.5. Signals

swsh does not terminate when it receives SIGINT and SIGTSTP signals.

3. Programs

The program to be implemented performs the same tasks as the existing program but is a simplified version for easy implementation.

3.1. head

SYNOPSYS

```
head [OPTION] file
```

DESCRIPTION

`file` Print 10 lines from the top of the file to standard output.

`-n K`

Print K lines instead of 10 lines.

`file` It assumed that file always exist.

3.2. tail

SYNOPSYS

```
tail [OPTION] file
```

DESCRIPTION

`file` Print 10 lines to standard output from the bottom of the file.

`-n K`

Print K lines instead of 10 lines.

`file` It assumed that file always exist.

3.3. `cat`

SYNOPSYS

```
cat file
```

DESCRIPTION

`file` Output the file to standard output.

`file` It assumed that file always exist.

3.4. `cp`

SYNOPSYS

```
cp file1 file2
```

DESCRIPTION

`file1` Make a copy of the file, and name it file2.

`file1` It assumed that file always exist. `file2` It cannot be overwritten.

3.5. `mv`

SYNOPSYS

```
mv file1 file2
```

DESCRIPTION

`file1` Renamed file1 to file2.

SEE ALSO

`rename(2)`

3.6. `rm`

SYNOPSYS

```
rm file
```

DESCRIPTION

`file` Remove file.

SEE ALSO

`unlink(2)`

3.7. `cd`

SYNOPSYS

```
cd dir
```

DESCRIPTION

Change current working directory to `dir`.

SEE ALSO

`chdir(2)`

3.8. pwd

SYNOPSYS

`pwd`

DESCRIPTION

Print currnet working directory to standard output.

SEE ALSO

`getcwd(3)`

3.9. exit

SYNOPSYS

`exit [NUM]`

DESCRIPTION

Print the string exit on standard error, then exit *swsh*.

If NUM is specified, NUM is returned as the exit value of the program, 0 otherwise.

SEE ALSO

`exit(3)`

3.10. Errors (Extra problem)

If you solve this problem, you can get extra points. If an error occurs while executing the mv, rm, cd command(cmd_type3), the message specified in the table below is combined as follows according to the type of the error and output as standard error. (This problem can be solved by using <errno.h>)

EACCESS	Permission denied
EISDIR	Is a directory
ENOENT	No such file or directory
ENOTDIR	Not a directory
EPERM	Permission denied
Other errors	Error occurred: <ERRNO> (Print error number)

```
command: ERROR_MESSAGE
e.g.)
mv: Permission denied
cd: Not a directory
```

4. Background

4.1. strace

A program that keeps track of system calls that occur when a program executes. For example,
`$ strace ls`
shows list of system call which is used to run ls.

You can also use the -p option to check the system call of the currently running process. In bash, the current command to see your pid is echo $$. If you open a new shell and pass the pid of this checked bash process to the strace -p option, you can see the system call used to run the shell.

4.2. whereis // which

To find the location of a file, you can use the whereis or which command..

```
$ which ls
/bin/ls
```

5. Restrictions

- Use Linux system calls and library functions which you had learned in this class.
- If a resource is dynamically allocated, it must be freed before the program terminates.
  - Resources refer to files, memory, and child processes.

6. Hand in instruction

- Write your **Name** and **Student ID** in a comment at the top of your source code.
- The program must be written in **one source code. (student_id.c)**
- Source code must be compiled using **make** command. (You should write **Makefile**)
- Compress your **source code** and **Makefile** into **"student_id.tar.gz"** and submit to **iCampus**.
- If you don't follow submission format or the code is not compiled with make, you will get extremely low point.

7. Logistics

- This is not a team project.
- The submission time is based on the iCampus submission time. If you submit your assignment late, 10% will be deducted immediately after the due date, and an additional 10% will be deducted every 24 hours. **(0 points after 3 days)**
- **If you copy something else, both of students get zero score.**
- If you have any questions, ask through the **Q&A board** in iCampus.