[SWE2024: System Programming Experiment – Spring 2023]

Programming Assignment #2

Due: 17th May. (Wed), 23:59:59

1. Introduction

In this assignment, you will make a Mini 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. Mini shell 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>)

commandline	::=	pipeline	
pipeline	::=	commands	or
		pipeline | commands	
commands	::=	command	or
		command < filename	or
		command > filename	or
		command >> filename	or
		command < filename > filename	or
		command < filename >> filename	
command	::=	executable [option(s)] [argument(s)]	or
		builtin_cmd [argument]	
executable	::=	{ ls, man, grep, sort, awk, bc, head,	
		tail, cat, cp, mv, rm, pwd }	or
		path	
builtin_cmd	::=	{ cd, exit }	
path	::=	{ pathname with leading "./" }	

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 IPC 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 of the process with the file [`filename`]
- Output redirection: Replace the standard output of the process with the file [`filename`]
  - If redirected by >, the file is initialized and rewritten.
  - If redirected by >>, append to the end of the file.

Also, for convenience of implementation, it is assumed that **each keyword of <table> is separated by a single 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 commandline

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
```

- `commandline` ➔ `pipeline` ➔ `commands` ➔ `command` ➔ `builtin_cmd [argument]`
  - `builtin_cmd` ➔ `cd`
  - `[argument]` ➔ `dir3`

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

```
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, one uses the following command.

```
./a.out
```

In other words, if [`command`] is a string (path) starting with "`./`", it can be derived from `executable`. (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.**

```
mini: command not found
```

2.2. Evaluating commandline

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 the command is implemented by yourself, create a child process (if necessary) to call that routine.
   B. If it is not implemented by 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, `Mini shell` only detects the following cases:

If the file specified by input redirection exists, it is assumed to be a normal file with read permission. If it does not exist, the following string is displayed as standard out.

```
mini: No such file or directory
```

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

Mini shell has 2 types of commands to be processed.

- `executable`: Programs that need to be loaded (fork – exec*) and run

- `builtin_cmd`: Must be implemented by yourself (Mini shell built-in command)

The commands corresponding to `executable` extracts options or arguments and creates a child process. Then, loads and executes the binary through the `exec` family of functions.
`builtin_cmd` must be implemented within the `Mini shell`.
For the executables that need to be implemented, section 3 defines the function of each command and mentions the system call / library functions required.

2.3. Process group

If `Mini shell` 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 `Mini shell` 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)
```

(Reference: $ 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

`Mini shell` does not terminate when it receives `SIGINT` and `SIGTSTP` signals.

(Refer to section 3.9 exit for exiting `Mini shell`)

3. Programs

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

For the commands that need to be implemented, below defines the function of each command and mentions the system call / library functions required. The implemented executables must be compiled with Make command and executable binaries must be created.

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 from the top of the file to standard output.

`-n K`

Print `K` lines instead of 10 lines. -n K

`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.

When missing parameters, print cp: missing file operand

When missing one parameter, print cp: missing destination file operand after ‘file1’

3.5. mv

SYNOPSYS

```
mv source destination
```

DESCRIPTION

`source` Rename `source` to `destination` or move `source` to `destination`

When missing parameters, print mv: missing file operand

When missing one parameter, print mv: missing destination file operand after 'source'

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 the current working directory to `dir`

SEE ALSO

`chdir(2)`

3.8. pwd

SYNOPSYS

```
pwd
```

DESCRIPTION

Print current working directory to `stdout`.

SEE ALSO

`getcwd(3)`

3.9. exit

SYNOPSYS

`exit [NUM]`

DESCRIPTION

Print the string exit on standard error, then exit `Mini shell`.

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

SEE ALSO

`exit(3)`

3.10. Errors

If an error occurs while executing `head, tail, cat, cp, mv, rm, cd`, the message specified in the table below is printed according to the type of the error and output as standard error.

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

Example of handling errors

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

4. Grading policy (total of 100 points)

- Executable (44)
  - Implementation and execution of [`ls, man, grep, sort, awk, bc`] (16)
  - Implementation and execution of [`head, tail, cat, cp, mv, rm, pwd`] (26)
  - Arbitrary binaries on `path` (2)
- Implementation and execution of `builtin_cmd` (6)
- Pipe and Redirection (40)
- Behavior of `Mini shell` when receiving SIGINT, SIGTSTP (5)
- Report (5)

- **10 points** are deducted each late day.
- **If the process group ID of the child process created by the `Mini shell` is the same as the process group ID of the `Mini shell`, the submission is not scored.**
  - This can be confirmed by executing a mini shell in one terminal, entering a command line, and then entering the following command in another terminal.
    ```
    $ ps fo user,pid,pgid,cmd | grep pa2
    ```
- Put your Makefile and *.c files in pa2 folder
- Submit using the submit command.
  ```
  $ ~swe2024-41_23s/bin/submit pa2 pa2
  ```
- Compiled binary name should be **"pa2"**
- The binaries for executables that you implemented must also be created.

5. Restrictions

- This is a personal project. You can discuss the task together, but you must write the source code by yourself.
- Use the Linux system call/library function you have learned so far to implement the task.
- Use of system() function is prohibited.
- If a resource is dynamically allocated, it must be freed before the program terminates.
  - Resources refer to files, memory, and child processes.
- **Failure of Make command or improper execution of functions will result in 0 point**.
- Submit the summary report about your design of the project and implementation as [studentID_report_pa2].pdf to iCampus.