Assignment 3 - Extending the shell

Operating Systems and Networks | Monsoon 2021

Deadline: 18th September, 5 PM

There would be NO deadline extensions.

The goal of this assignment is to enhance your user-defined interactive shell program so that it can handle background and foreground processes and handle signals sent to them. It should also be able to handle input/output redirections and pipes.

The following are the specifications for the assignment.

Specification 1: Input/Output Redirection [10 marks]

Using the symbols <, > and >>, the output of commands, usually written to stdout, can be redirected to another file, or the input taken from a file other than stdin. Both input and output redirection can be used simultaneously. Your shell should support this functionality.

Your shell should handle these cases appropriately:

- An error message should be displayed if the input file does not exist.
- The output file should be created (with permissions 0644)if it does not already exist.
- In case the output file already exists, it should be overwritten in case of > and appended to in case of >>.

Example:

```
# output redirection
<tux@linux:~> echo "hello" > output.txt
# input redirection
<tux@linux:~> cat < example.txt
# input/output redirection
<tux@linux:~> sort < file1.txt > lines_sorted.txt
```

You are NOT required to handle multiple inputs and outputs.

Example: echo hello >> test1.txt > text2.txt

Specification 2: Command Pipelines [10 marks]

A pipe, identified by |, redirects the output of the command on the left as input to the command on the right. One or more commands can be piped as the following examples show. Your program must be able to support any number of pipes.

Example:

```
# two commands
<tux@linux:~> cat file.txt | wc
# three commands
<tux@linux:~> cat sample2.txt | head -7 | tail -5
```

Specification 3: I/O Redirection within Command Pipelines [10 marks]

Input/output redirection can occur within command pipelines, as the examples below show. Your shell should be able to handle this.

Example:

```
<tux@linux:~> ls | grep *.txt > out.txt
<tux@linux:~> cat < in.txt | wc -l > lines.txt
```

Specification 4: User-defined Commands [40 marks]

1. **jobs**: This command prints a list of all currently running background processes spawned by the shell in alphabetical order of the command name, along with their job number (a sequential number assigned by your shell), process ID, and their state, which can either be running or stopped. There may be flags specified as well. If the flag specified is -r, then print only the running processes else if the flag is -s then print the stopped processes only.

Example:

```
<tux@linux:~> jobs
[1] Running emacs assign1.txt [221]
[2] Running firefox [430]
[4] Stopped gedit [3213]
[3] Stopped vim [3211]
# These are all sorted in alphabetical order.
# The job number indicates the order in which they were
# created.
<tux@linux:~> jobs -r
[1] Running emacs assign1.txt [221]
[2] Running firefox [430]
<tux@linux:~> jobs -s
[4] Stopped gedit [3213]
[3] Stopped vim [3211]
```

2. **sig:** Takes the job number (assigned by your shell) of a running job and sends the signal corresponding to the signal number to that process. The shell should throw an error if no job with the given number exists. For a list of signals, look up the manual entry for 'signal' on manual page 7.

Example:

```
<tux@linux:~> sig 2 9
# sends SIGKILL (signal number 9) to the process firefox (job # list as per the previous example), causing it to terminate
```

3. **fg:** Brings the running or stopped background job corresponding to job number to the foreground, and changes its state to running. The shell should throw an error if no job with the given job number exists.

Example:

```
<tux@linux:~> fg 4
# brings [4] gedit to the foreground
```

4. **bg:** Changes the state of a stopped background job to running (in the background). The shell should throw an error if no background job corresponding to the given job number exists, and do nothing if the job is already running in the background.

Example:

```
<tux@linux:~> bg 3
# Changes the state of [3] vim to running (in the # background).
```

Specification 5: Signal Handling [15 marks]

1. CTRL-Z It should push any currently running foreground job into the background, and change its state from 'running' to 'stopped.' This should have no effect on the shell if there is no foreground process running.
2. CTRL-C It should interrupt any currently running foreground job, by sending it the SIGINT signal. This should have no effect on the shell if there is no foreground process running.
3. CTRL-D It should log you out of your shell, without having any effect on the actual terminal.

Specification 6: Autocompletion [15 marks]

The `tab` key can be pressed at any time of your prompt. When you press the `tab` key, it will print all files which have the same starting letters as the `ls` command and then again show the same prompt. If there is only one file/folder which has the same starting characters, the prompt should be filled with that. If it's a file, a space should be placed after the file. If it's a folder, a '/' should be put after the folder name.

Example:
Assume the files in this directory are: `first.c, sec.c, second.cpp`
Assume the folders in the directory are: `third, seventh`

```
1. <Name@UBUNTU:~/> cat fi<tab>
2. <Name@UBUNTU:~/> cd s<tab>
3. <Name@UBUNTU:~/> ls the<tab>
4. <Name@UBUNTU:~/> <tab>
```

So, the output of pressing <tab> on cases are

<cur> marks the place where your cursor should be present.

```
1. <Name@UBUNTU:~/> cat first.c <cur> #Space is present after first.c
2. sec.c
   second.cpp
   seventh/
   <Name@UBUNTU:~/> cd se<cur>
# Since all the files had 'se' as common, 'e is completed'
3. <Name@UBUNTU:~/> ls third/<cur>
# Since there is only one folder, it is directly completed
# with '/' as the last character waiting for input
4. first.c
   sec.c
   second.cpp
   third/
   seventh/
   <Name@UBUNTU:~/> <cur>
# The order of the files displayed does not matter
```

Bonus(optional) [Ungraded]

Replay

Implement a ‘replay’ command which executes a particular command in a fixed time interval for a certain period.

Example:

```
<Name@UBUNTU:~> replay -command echo "hi" -interval 3 -period 6
```

This command should execute the echo "hi" command after every 3 seconds until 6 seconds are elapsed. In this example, the echo "hi" command should be executed 2 times, once after 3 seconds and then after 6 seconds.

Nightswatch <num> <command>

Look up the man page entry for the ‘watch’ command - ‘man watch’. You will be implementing a modified, very specific version of the watch. It executes the command every <num> seconds until the ‘q’ key is pressed.

The three commands to execute are:

1. Interrupt [3 marks]: print the number of times the CPU(s) has(ve) been interrupted by the keyboardcontroller (i8042 with IRQ 1). There will be a line output to stdout once in every time interval that was specified using <num>. If your processor has 4 cores (quadcore machine), it probably has 8 threads and for each thread, output the number of times that particular CPU has been interrupted by the keyboard controller.

Example:

```
<Name@UBUNTU:~> nightswatch 5 interrupt
CPU0 CPU1 CPU2 CPU3 CPU4 CPU5 CPU6 CPU7
2    13   2    1    0    2    1    0
2    13   4    1    0    4    1    0
...
...
...
```

A line every 5 seconds until ‘q’ is pressed.

2. newborn [3 marks]: print the PID of the process that was most recently created on the system (you cannot use system programs for this).

Example:

```
<Name@UBUNTU:~> nightswatch -n 1 newborn
26120
20192
26106
...
```

A line every 1 second until ‘q’ is pressed.

3. dirty [3 marks]: print the size of the part of the memory which is dirty .

Example:

```
<Name@UBUNTU:~> nightswatch -n 1 dirty
968 kB
1033 kB
57 kB
...
```

A line every 1 second until ‘q’ is pressed.

General Notes

1. Use of popen, pclose, system() calls is not permitted.
2. Some helpful routines and systemcalls: getenv, signal, dup, dup2, wait, waitpid, getpid, kill, execvp, malloc, strtok, fork, setpgid, setenv and getchar.
3. Use the exec family of commands to execute system commands. If the command fails to run or returns an error, it should be handled appropriately. Look at perror.h for appropriate routines to handle errors.
4. Use fork() for creating child processes where needed and wait() for waiting for and reaping them.
5. Use signal lhandlers to handle signals when background processes exit.
6. The user can type the command anywhere on the command line, leaving spaces and tabs in between. Your shell should be able to handle this.
7. The user can type in any command, including running another process instance of your shell program. In all cases, your shell should be able to execute the command or show an appropriate error message if the command cannot be executed.
8. You need not implement background functionality for internal commands such as cd, ls, etc.
9. You have to implement piping and redirection for internal commands.
10. You need not implement redirection operators like 2>&1, &>, >& or 2>.
11. The symbols <, >, >>, &, |, ; , - would always correspond to their special meaning and would not appear otherwise, such as in inputs to echo etc.

Submission Format

1. Upload a compressedfile, rollnumber.tar.gz, which creates a folder rollnumber on extracting, containing all your files.
2. Make sure you write a makefile with appropriate flags and linker options for compiling your code.
3. Include a proper README file briefly describing your work and which file corresponds to what part.