Assignment 3
Operating Systems and Networks | Monsoon 2020

Extending the shell

Due on Tuesday, September 29, 2020 at 23:55 IST

This is an individual assignment, in continuation of Assignment-2. We recommend you start early.

The goal of this assignment is to enhance your user-defined interactive shell program so that it can handle **background and foreground processes, signals** sent to them as well as **input/output redirection** and **command pipelines**.

The following are the specifications for this assignment. For each of the requirements an appropriate example is given along with it.

Specification 1: Input/Output Redirection

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:~> diff file1.txt file2.txt > output.txt

# input redirection
<tux@linux:~> sort < lines.txt

# input/output redirection
<tux@linux:~> sort < lines.txt > sortedlines.txt
```

Specification 2: Command Pipelines

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:~> more file.txt | wc

# three commands
<tux@linux:~> grep "new" temp.txt | cat somefile.txt | wc
```

Specification 3: I/O Redirection within Command Pipelines

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

Hint: *I/O redirection binds tighter than pipes, and can be treated as arguments to the command.*

Example:

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

Specification 4: User-defined Commands

Your shell should support the following internal commands:

1. **`setenv var [value]`**
   Initially, your shell inherits environment variables from its parent process. This command allows the user to set the value of the environment variable **`var`** to **`value`**, creating the environment variable if it doesn't already exist. If **`value`** is omitted, the variable's value is set to the empty string. It is an error for the **`setenv`** command to be invoked with zero or more than two command-line arguments.

2. **`unsetenv var`**
   This command should cause your shell to destroy the environment variable **`var`**, if it exists. It is an error for the **`unsetenv`** command to be invoked with zero or more than one command-line arguments.

3. **`jobs`**
   This command prints a list of all currently running background processes spawned by the shell in order of their creation times, along with their job number (a sequential number assigned by your shell), process ID and their state, which can either be **`running`** or **`stopped`**.

   Example:

   ```
   <tux@linux:~> jobs
   [1] Running emacs assign1.txt [221]
   ```

```
[2] Running firefox [430]
[3] Stopped vim [3211]
[4] Stopped gedit [3213]
# Here [4] gedit is the most recently created background job,
# and [1] emacs is the oldest.
```

4. **kjob <job number> <signal number>**
   Takes the job number (assigned by your shell) of a running job and sends the signal corresponding to **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:~> kjob 2 9
   # sends SIGKILL (signal number 9) to the process firefox (job
   # list as per the previous example), causing it to terminate
   ```

5. **fg <job number>**
   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 3
   # brings [3] vim to the foreground
   ```

6. **bg <job number>**
   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 4
# Changes the state of [4] gedit to running (in the
# background).
```

7. **`overkill`**
   This command kills all background processes at once.

8. **`quit`**
   This command, you guessed it, exits the shell. Your shell should also exit on pressing **`CTRL+D`**, which corresponds to the **`EOF`** character.

Specification 5: Signal Handling

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.

   Hint: *When you run your shell from the standard Unix shell, your shell is running in the foreground process group. If your shell forks any child processes, they will also be, by default, a part of the foreground process group. Pressing* **`CTRL-C`** *sends the* **`SIGINT`** *signal to every process in the foreground process group, including your shell, interrupting all of them.*

However, we only want the foreground process forked from your shell, if any, to be interrupted.

Bonus Specification 1: Last Working Directory

Implement the **-** argument for the **cd** command. This switches to the previous working directory and prints its path.

Example:

```
<tux@linux:~/oswald/chesterfield/> cd cobblepot
<tux@linux:~/oswald/chesterfield/cobblepot> cd -
~/oswald/chesterfield
<tux@linux:~/oswald/chesterfield>
```

The previous working directory need not be maintained across shell sessions. This command should do nothing (except printing the path) if the directory was never changed in the current shell session prior to calling this.

Bonus Specification 2: Exit Codes

Exit codes indicate whether a process exited normally, or encountered an error. You'll be handling and displaying them for all external commands, as well as implementing them for internal commands (even though they aren't separate processes).

Information about the previous command's exit code should be displayed alongside the next prompt as **:')** if the command exited successfully, or as **:'(** if it encountered an error.

Example:

```
<tux@linux:~> ls
oswald           henry           pingu           pingo
```

```
:')<tux@linux:~> ps -Q
error: unsupported SysV option
...
:'(<tux@linux:~>
```

- If an external foreground process was stopped and sent to the background using **`CTRL-Z`**, it corresponds to an unsuccessful exit.
- On executing **`fg`**, the exit status information corresponds to that of the process which was resumed in the foreground, unless an error occurred while executing **`fg`** itself, in which case it is an unsuccessful exit.
- On executing **`bg`**, the exit status is successful for as long as the signal to continue was delivered to the appropriate process.
- On executing an external command in the background using **`&`**, the exit status is successful for as long as a child process was successfully forked, irrespective of whether the command was successfully **`exec`**'d or what its exit code was.
- For a pipeline or a semicolon separated list of commands, the exit status corresponds to that of the last command in the pipeline/list.
- An erroneous exit status follows parse errors, invalid commands, etc.

Bonus Specification 3: Command Chaining with Logical AND and OR

The logical **`AND`** and **`OR`** operators can be used to chain commands, such that the exit code of the entire chain is the logical **`AND`** or **`OR`** (respectively) of the individual exit codes (a successful exit corresponds to **`true`** and an unsuccessful one corresponds to **`false`**). We'll use the symbols **`@`** to denote **`AND`** and **`$`** to denote **`OR`**. These operators bind looser than **`|`**, **`>`**, **`<`**, **`>>`**, **`&`**, but tighter than **`;`**.

These operators short-circuit, executing their second operands only if needed to evaluate their truth value. This means:
For **`command-1 @ command-2`**
In case **`command-1`** exits successfully, **`command-2`** is executed and governs the truth value of the operator. Otherwise, **`command-2`** is not run and the operator evaluates to **`false`**.
For **`command-1 $ command-2`**
In case **`command-1`** exits unsuccessfully, **`command-2`** is run and governs the truth value of the operator. Otherwise, **`command-2`** is not run and the operator evaluates to **`true`**.
`$` and **`@`** have the same precedence, hence are evaluated left-to-right in a chain consisting of multiple **`@`**'s and **`$`**'s.

Here, the commands in between could contain **`&`**, **`|`**, **`<`**, **`>`**, **`>>`**.

Example:

```
<tux@linux:~> ls $ echo penguins
oswald            henry            pingu            pingo
:')<tux@linux:~> ls @ echo penguins
oswald            henry            pingu            pingo
penguins
:')<tux@linux:~> ps -Q $ ps -Q $ ls @ echo penguins $ ls
error: unsupported SysV option
...
error: unsupported SysV option
...
oswald            henry            pingu            pingo
penguins
:')<tux@linux:~> ps -Q $ ps -Q
error: unsupported SysV option
```

```
...
error: unsupported SysV option
...
:'(<tux@linux:~> ls @ echo successful $ echo fail
oswald            henry            pingu            pingo
successful
:')<tux@linux:~>
```

Hint: *The chain is evaluated from left to right, because* `@` *and* `$` *have the same precedence. In the last example above, the* `@`*, being on the left, is evaluated first. Because* `ls` *(its first operand) runs successfully,* `@` *does not short-circuit, executing* `echo successful` *and evaluating to* `true`*. Now, the* `$` *is evaluated, with a left-hand-side that evaluates to* `true`*. Thus, it short-circuits, and evaluates to* `true` *without executing the command on its right. And we see a* `:')` *symbol alongside the next prompt. Thus,* `com-1 op com-2 op com-3 op com-4` *corresponds to* `(((com-1 op com-2) op com-3) op com-4)`*, where* `op` *is one of* `@` *and* `$`*..*

Useful to know

1. Use of **popen**, **pclose**, **system()** calls is not permitted.
2. Some helpful routines and system-calls: **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 handlers to handle signals (surprise!) 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 compressed file, **`rollnumber_assgn3.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 **`readme`** file briefly describing your work and which file corresponds to what part.