Shell with pipe

Martin Mårtensson

October 06, 2020

Shell w. Pipe

Manual

Execute the `shell.py` with

```
python shell.py
```

Then you will be presented with a fully functional shell, which is ready to accept your input.

It will look something like this:

```
./your_path %
```

You can now run any Linux command on your system.

Here is some inspiration:

- `ls` (list directory contents)
- `cd` (change directory)
- `cat` (concatenate files and print to stdout)
- `grep` (print lines that mach patterns)
- `awk` (pattern scanning and text processing language)
- `sort` (sort lines of text files)
- `vim` (A programmer's text editor)

And many more.

> The Reason this works is because the python program is launching system calls which means the program speaks directly to the kernel (Linux) of the operating system.

Run a command

To run a command, you just need to type in the name of the command and possible arguments

Example:

```
ls -l /
```

This command will list the root directory of your Linux file system.

the `-l` is a flag that enables a special list view which will show each entity of the content of the directory on a separate line with some extra information (out of this scope)

That's it.

Piping commands

The shell also has the functionality of piping, which means you can take the output of one command, and pass it into the input of another command.

> Piping is a system call, which makes it possible to communicate data between processes on the OS.

A single pipe So to give an example, we can use the command that we used before, to get an output, and then use that input in another command. That other command could be awk.

Example:

```
ls -l / | awk '{print $9 "\t" $3}'
```

This command will take the output of `ls -l /` and redirect it to a pipe which will then put as input into the awk command which will filter the data and print a list of the files/folders in the root directory but only with their names and the name of the owner.

Multiple pipes So to give an example, we can use the commands that we used before, and then use `sort -r`, to present the output in reversed alphabetic order.

Example:

```
ls -l / | awk '{print $9 "\t" $3}' | sort -r
```

So basically the same happens, here. + The output of `ls -l /` is passed into the pipe + The pipe redirects the data into awk's input + The next pipe then consumes the output of awk + And redirects it into the input of sort + The output of sort doesn't have a pipe so it will be passed to stdout and the final result will be printed on the screen

Implementation

This section is written as an alternative to commenting the code

Implementation of the project has been done in python, because + python has a way to directly communicate with the Linux system through system-calls. + Another reason is because, in python it is easy to work with strings and lists which is used a lot in this program while tokenizing the commands and later for dealing with them.

Some important python features used are: + Slicing of a list `list[x:i]` + String cleaning with `str.strip()` + String splitting with `str.split()`

The `tokenizer` function which is responsible for setting up the command returns a list of strings or a list of lists of strings.

All other functions return `None` because their only responsibility is to run system-calls in the right order, based on the command given by the user.

Libraries

```
from re import search
from os import execvp,wait,fork,close,pipe,chdir,dup2,getcwd,_exit
```

- The `search` function from `re` library is also used to help tokenizing the user-input.
- The most important library in the imports is the `os` library which includes the functions used to communicate directly with the Linux kernel (the system calls).
  - `execvp` replaces current process with new process image.
  - `wait` makes the current process wait for its child process to change state/finish.
  - `fork` creates a new process by duplicating the current process.
  - `close` closes a file descriptor.
  - `pipe` creates a uni directional data channel in/out
  - `dup2` creates a copy of one file descriptor and redirects it to another file descriptor.
  - `getcwd` copies the absolute path of the current working directory to STDOUT
  - `_exit` terminates the current process

Default values

stdin, stdout and child has been declared for transparency.

```
STDIN,STDOUT,CHILD = 0,1,0
```

command

```
def command(cmd):
    try: execvp(cmd[0].strip(), cmd)
    except OSError: print('Command not found.'); _exit(127)
```

This function has been made to avoid redundancy and to improve flexibility in the program.

- First the function takes a command of type `list`
- Then it tries to run the `execvp(cmd_name: str, cmd: list)`
  - which will will replace the current process
  - and try to execute a command from the operating system.
- If any error will appear during the execution of `execvp`, the function will raise an `OSException`
  - and the command will print "Command not found!" Because that is almost always the reason that this error is raised.
  - also the exit code takes the number 127 which is an exit code that will imply that the command was not found.

copy

```
def copy(_close, _duplicate, fd=STDOUT):
    close(_close)
    dup2(_duplicate, fd)
```

This function has also been made to avoid redundancy and to improve flexibility in the program.

- First the function takes a file-descriptor as a parameter, this file-descriptor is gonna be closed
- Secondly the function takes another file-descriptor as a parameter, this file-descriptor is gonna be taking the place of the third file-descriptor in the parameters.
- Lastly the function takes a third file-descriptor as a parameter, this file-descriptor is gonna stand down, and let the second file-descriptor in the parameters take over.

the last parameter id set to `STDOUT` pr. default because it is the file-descriptor mostly used

fork

```
def my_fork(child, parrent=lambda:None):
    pid = fork()
    if pid > CHILD: wait(); parrent()
    elif pid == CHILD: child()
    else: _exit(1)
```

This is the function with responsibility for forking.

- First the function takes 2 other functions in its parameters
  - a child function which will be responsible for executing stuff in the child process.
  - a parent function which will be responsible for executing stuff in the parent process.
- If the pid of the fork() is greater than `CHILD` which is 0, then the process will wait for child to finish.
- Else if pid is 0 then `child()` will run, lastly
- Else if pid is less than 0 it means that an error occurred and the process will exit with the exit code 1 which means something abnormal happened.

my_pipe

```
def my_pipe(cmd, _r=None, _w=None):
    def parrent():
        if _r is not None: copy(_r, _w)
        copy(w,r,STDIN); command(cmd[len(cmd)-1])
    def child():
        if len(cmd) > 2: my_pipe(cmd[:-1], r, w)
        else: copy(r,w); command(cmd[0])
    r, w = pipe()
    my_fork(child,parrent)
```

This is a very advanced little recursive function.

note: recursion will only happen if more than one pipe has been entered in the command line

Walking into recursion + Firstly the function takes a `list` as an argument, + The `list` is a list of lists of strings, and can be from 2 to `infinite` long. + The first time the function is called it will be without the outer pipe-channels + `_r` for outer_reading and + `_w` for outer_writing. + Then two inner functions are declared. + The functions have access to the variables declared inside their container function, + this means all 4 pipe channels. + Then the first pipe is made + The `parent` and `child` functions are passed into `my_fork()` + which will create a fork + which will execute `child()` first i + and then `parent()` + Inside `child` we will check if the `list` of commands is longer than 2. + If `True` then the function will call itself but slice away the last command from the list which will be run in the current child process's belonging parent process, later. + Also the pipe-channel belonging to the current process pair, will be passed into the function call's parameter. + This means that the function will keep on spawning new pipes, and processes, and each **n**th process will be able to communicate with the **n+1**th process through the outer pipe-channels.

Walking out of recursion

- When all `pipes` has been run through and there is only 2 commands left in the `list`,
- then all `parent` processes will be waiting for the `child` processes to finish and the last `child` process will **skip the if statement** and go to the else statement
  - where it will execute the first command in the `list` of commands.
- Then `copy(r,w)` will do its job, and the parent process will run.
- If there is more than one pipe
  - then the outer channels will not be `None`
  - and the outer `pipein` (named `_r`) will close
  - and outer `pipeout` will take input from `STDOUT`,
  - at the same time the inner `pipeout` will close
  - and the inner `pipein` will put its data that it got from the `child` process's `execvp` command, into the `STDIN` which will be used as argument in the `execvp` which will now be run, in the `parent` process.
- And then all the `parent` processes for each `child` process will repeat the same
- until we are all the way back to the first `parent` process
  - which will be waiting to execute the last command in the `list` of commands
  - which were originally passed into the outermost `my_pipe`
  - and because this function does not have a outer `pipe`
  - then the `STDOUT` will not be redirected by `dup2`
  - and then the final result will be printed to the screen.

process

```
def process(cmd):
    if 'cd' == cmd[0]: chdir(cmd[1])
    elif 'exit' == cmd[0]: exit(0)
    else: my_fork(lambda: my_pipe(cmd)) if type(cmd[0]) == list else my_fork(lambda: command(cmd))
```

This is the function which will see if the command issued by the user requires to run a `fork` and a `pipe` or if it is just a `cd` or `exit`

- If the command is a list of lists it means the command has pipes, because this is how the `tokenizer` function tokenizes the string given by the user.
- If exit is called then the system call `_exit(0)` will be run. 0 means that everything is OK.

tokenizer

```
def tokenize(cmd):
    if '|' in cmd: return [tokenize(x.strip()) for x in [x.strip() for x in cmd.split('|')]]
    elif search('\'.*?\'', cmd): return [x for x in cmd.split("'") if x]
    else: return [x.strip() for x in cmd.split()]
```

This is a simpler recursive function which has responsibility for tokenizing strings into something that the rest of the program can understand.

- First the function checks for pipes
- if there is a pipe then the function will split the string for each pipe-sign into a list of strings, and then call itself for each string in the list. (which will now be without a pipe-sign) and then lastly return everything as a list of lists.
- if there is no pipe-sign in the string then the function will use regex to check if there is a defined string in the command (‘string’)(only single quotes are allowed) it will then return a list splitted for each space except for spaces in the defined string which will be in the list as a string for itself.
- Lastly if there is no pipes and no strings, then the function will just split the string for each space into a list of strings, and strip potential whitespace that might occure.

The main loop

```
while True:
    try: process(tokenize(input(f'./{getcwd().split("/")[-1]} % ')))
    except IndexError: pass
```

This loop will go on until the process somehow will be terminated. + First the program will print the current working directory and wait for user input, which will be passed into the `tokenizer` function which will return its output into the `process` function. + If the user just presses enter then the program will encounter an `IndexError` because functions later in the flow will look for specific indexes in the `cmd` list this will be overlooked and `pass`ed by `except`

Bibliography

- I have used the python documentation as a reference to learn about the system calls
- I got some inspiration from this Stack Overflow question How to use pipes an redirects using os.execv(), if possible?
- I asked two questions on Stack Overflow regarding the assignment, and I guess they should be in the bibliography as well.
  - How to pipe the output of execvp to variable in Python
  - How to deal with multiple pipes in shell program?
- And then of course i used the man pages in Linux.