COMPSCI 340 S2 C 2014
SOFTENG 370 S2 C 2014
Operating Systems
Assignment 1 – Implementing a Shell

Due Monday 18th of August 9:30 pm

Introduction

One of the long standing jokes in the Unix world is that if you don't like a particular shell environment like bash, csh, tcsh, sh etc., then you can write your own. In this assignment you will do just that (and hopefully find that it is not too difficult).

What you have to do

Write a shell in Python that starts normal Unix commands, provides command pipelines, has a simple history command mechanism and provides job control, along with some basic built-in commands. The Python shell should work in a very similar way to bash. However it is much more limited than bash. Theoretically there is nothing to stop you extending the assignment to make it as powerful as bash or any other shell. You start your shell by typing `python3 psh.py`.

The appendix shows example output from running the shell using input from test files available on the assignment web page.

1. The first task is to get the shell running and accept one command at a time and execute that command. The shell waits for that command to complete before prompting for more.
2. Get the built-in change directory (`cd`) command working.
3. Implement the history mechanism. The history maintains the last 10 commands executed. Any of those commands can be rerun by typing its history number. The history mechanism is not as powerful as that with a normal shell; you do not have to provide history editing. See the description below.
4. Make the pipeline system work. The user should be able to type a series of commands where the output of each command gets passed as input to the next command in the pipeline. Commands are separated by the `|` pipe character.
5. Commands which finish with an ampersand `&` are started in the background. Background commands run but the shell does not wait for them to complete. This means after a command is started in the background the user can immediately enter another command. When a background command begins it is known as a job and information about the job is written to the terminal. e.g. `sleep 10 &` might produce

   ```
   [1]   8034
   ```

   The number in brackets is the job number. Job numbers start at one and are consecutive. When jobs finish their numbers may be reused. The next job number is always one more than the greatest current job number (or one if there are no current jobs).

   The second number, `8034` in this case, is the process id of the process executing the job.

   Every time around its loop and just before printing the prompt the shell checks to see if any background jobs have completed. If they have it prints the information before the prompt. See the `test6` output for an example.
6. Implement the `jobs` command associated with background processes. The `jobs` command prints out information about all background (and stopped) jobs. The output includes the job number, the state of the job (sleeping, runnable, stopped, idle, zombie or done), and the complete command line which started the job. See `test5` and `test6` for examples.

All errors should be handled nicely by the shell and sensible error messages shown to the user. The shell shouldn't crash given bad input.

SOFTENG 370 students also have to implement the following.

7. Typing `CTRL-Z` should stop any command running in the foreground. Then the `fg` command can be used to continue the job in the foreground. The `bg` command continues the job in the background. The `kill`

command terminates the job. These commands can also accept a parameter, the number of a job (the same number shown by the `jobs` command), to carry out the operation on that particular job.

Starting a pipeline

When you come to implement the pipeline system you will need to create pipes to communicate between processes and fork processes to have several running simultaneously. The following pseudocode is courtesy of an old book "The Design of the Unix Operating System" by Maurice Bach. Even though the book is out of date, it still gives a wonderful insight into the internals of reasonably early Unix. My Python solution includes an almost exact translation of this pseudocode. I will explain this code in detail in class.

```
/* read command line until "end of file" */
while (read(stdin, buffer, numchars))
{
      /* parse command line */
      if (/* command line contains & */)
            amper = 1;
      else
            amper = 0;
      /* for commands not part of the shell command language */
      if (fork() == 0)
      {
            if (/* piping */)
            {
                  pipe(fildes);
                  if (fork() == 0)
                  {
                        /* first component of command line */
                        close(stdout);
                        dup(fildes[1]);
                        close(fildes[1]);
                        close(fildes[0]);
                        /* stdout now goes to pipe */
                        /* child process does command */
                        execlp(command1, command1, 0);
                  }
                  /* second component of command line */
                  close(stdin);
                  dup(fildes[0]);
                  close(fildes[0]);
                  close(fildes[1]);
                  /* standard input now comes from the pipe */
            }
            execve(command2, command2, 0);
      }
      /* parent continues over here...
       * waits for child to exit if required
       */
      if (amper == 0)
            retid = wait(&status);
}
```

Built-in commands

You need to implement the following built-in commands. All normal commands are other programs which you will invoke using `os.execvp`.

pwd

The print working directory command. The shell maintains the notion of a current working directory. This command prints the full pathname to the current working directory.

cd

The change directory command. This can accept either a partial or full pathname of the directory the user wants to change the current working directory to. As in most shells the `..` characters can be used to represent parent directories. If no directory is specified the command changes the current working directory to the home directory. The home directory is the directory the program starts running in (`os.getcwd()` when the program starts).

history or h

The history command (also abbreviated as `h`) can accept one parameter. If no parameter is passed it prints the last 10 command lines entered by the user. e.g.

```
psh> history
    1: ls
    2: cd
    3: pwd
    4: ls -l
    5: ls -l | grep ^d | wc -l
    6: cd Desktop
    7: ls
    8: sleep 1&
    9: jobs
   10: history
```

Each command has a unique number associated with it. If the history command is called with a number the corresponding command (if available) is executed again and that command is entered as the current element in the history list. e.g. continuing on from the list above:

```
psh> history 5
       4
psh> h
    3: pwd
    4: ls -l
    5: ls -l | grep ^d | wc -l
    6: cd Desktop
    7: ls
    8: sleep 1&
    9: jobs
   10: history
   11: ls -l | grep ^d | wc -l
   12: h
```

jobs

The jobs command lists all current jobs (processes running in the background) started up by the shell. Each job is represented as a job number, a state, and the command line which started it. e.g.

```
psh> perl -e 'for ($c = 10; $c > 0; $c--) { print "a\n"; sleep 1 }'&
[1]  19914
psh> a
a
jobs
a
[1] <Sleeping> perl -e 'for ($c = 10; $c > 0; $c--) { print "a\n"; sleep 1 }'&
psh> a
a
a
a
a
a
a

psh>
[1] <Done> perl -e 'for ($c = 10; $c > 0; $c--) { print "a\n"; sleep 1 }'&
psh>
```

As is shown here the output `a`'s appear jumbled up with the command `jobs` typed at the prompt, as the processes are executing simultaneously.

When a new job is created, either by backgrounding a command with `&` or (for SOFTENG 370 students) by using `CTRL-Z` (to suspend the currently executing process) the new job number is one more than the current maximum job number. If there are no current jobs the new job number is always one.

One way to discover the state of a process is by grabbing the output of `ps -p pid -o state=`, where `pid` is the process id of the process. You will occasionally see zombie processes. The way to remove a zombie process is to wait for it.

SOFTENG 370 students also need to implement the `fg`, `bg` and `kill` commands. If these commands have no parameter they work on the last stopped command. Otherwise they work on the specified job. The kill command should always succeed (use `SIGKILL`).

Testing

The markers will test your program by running it with a number of input files. The files will be very similar to the test input files on the assignment web page. The files will test the following capabilities:

test1

Running simple commands (some with options).

test2

Built-in commands `pwd` and `cd`.

test3

`History` and history substitution.

test4

Command pipelines.

test5

Simple backgrounding and the built-in `jobs` command.

test6

Backgrounding pipelines.

Note

This shell does not deal with multiline commands, nor with multiple commands on a single line separated with semi-colons.

References and info

When you need to determine whether input is coming from a file or from the keyboard you can use:

```
redirected = not os.isatty(sys.stdin.fileno())
```

You can see that `sys.stdin` is the standard input file. Similarly `sys.stdout` and `sys.stderr` are the output and error files.

Read the Python documentation for: `os.pipe()`, `os.fork()`, `os.execvp()`, `os.dup()`, `os.dup2()`, `os.close()`, `os.waitpid()`, `os.getcwd()`, `os.chdir()`, `sys.exit()`

And for the 370 extension: `signal.signal()`, `os.kill()`

The following man pages:

`man ps`, `man 2 kill`, `man 7 signal` (find out what SIGSTOP, SIGCONT, SIGTSTP mean).

Questions

Answer the following questions. Put the answers in a simple text file called `a1Answers.txt`.

1. Provide your Python code equivalent to the pipeline pseudocode provided on page 2. Under each line explain what it is doing and why.

2. In a properly running shell (either yours or bash) if you type `cd ..` and then type `pwd` you will see the parent directory. However, if you type `cd .. | pwd` you see the current directory. Explain why this is so.

Prizes for Extra Component

There are prizes associated with this assignment kindly donated by Serato (serato.com).

The first prize includes a DDJ-SB (RRP $500)

http://pioneerdj.com/english/products/controller/ddj-sb.html

and a licence of Serato DJ and Serato Video (RRP $300)

The second prize is a pair of AIAIAIAI TMA-1 headphones (RRP $300)

http://www.aiaiai.dk/store/headphones/tma-1-nomic

Because standard Python has a Global Interpreter Lock, we can't usually use multiple cores with Python using threads. The Python approach to use multiple cores is to spin off new processes.

To be considered for these prizes you could extend the shell so that you can start multiple copies of the same program. It is up to you to design and describe the shell commands to do this. You probably want to include a way to send different input to each instance of the program and possibly collect the results from all instances when they finish. Alternately any addition to the shell in the spirit of multiple processes will also be considered for the prizes.

If you are taking part in this extra component you submit your `psh.py` file as normal including its extra capabilities and you also submit a text or pdf document called `extra.txt` or `extra.pdf` explaining your additions and giving an example of their use.

Submitting the assignment

Make sure your name and upi is included in every file you submit.

Use the assignment drop box to submit

Any work you submit must be your work and your work alone – see the Departmental policy on cheating http://www.cs.auckland.ac.nz/compsci340s2c/assignments/.

Marking guide

The markers will redirect data from text files as the standard input. Your program must work with input both from the keyboard and from redirected input and produce visible output. You will not receive any marks for a section which does not produce output (even if it works perfectly behind the scenes).

The shell can start one command at a time and waits until that command is finished before prompting for a new command. (4 marks)

`pwd` and `cd` work correctly. (3 marks)

The `history` command works correctly. (3 marks)

Command pipelines work correctly. (4 marks)

Commands can be started in the background (they continue running). (2 marks)

Pipelines can be started in the background. (2 marks)

The `jobs` command works correctly. (2 marks)

Question 1 (3 marks)

Question 2 (2 marks)

Total marks for COMPSCI 340 students are 25.

SOFTENG 370 students also should have:

Job control works correctly with CTRL-Z to stop foreground jobs, and the bg, fg and kill commands. (5 marks)

Total marks for SOFTENG 370 students are 30.

This assignment is worth 7% of your final grade.

Appendix

Your output will be different and may have different orders and states for processes.

Example output using test1

```
python3 psh.py < test1
psh> ls
A1 2007               A1.pages  pipeline.py  psutil-2.1.1       shlextry.py  test2
test4  test6  worker.py
A1 handout.pages  a1.py     psh.py       shell_separate.py  test1   test3 test5  test7
psh> ls -l
total 17
drwx------ 1 robert robert    0 Jul 14 10:25 A1 2007
drwx------ 1 robert robert    0 Jul 28 21:43 A1 handout.pages
…
-rwx------ 1 robert robert  275 Jul  3 13:53 worker.py
psh> ps -o state,pid,ppid,wchan,command
S   PID  PPID WCHAN  COMMAND
S  3909  3899 wait   bash
S  5145  3909 wait   python3 ./psh.py
R  5148  5145 -      ps -o state,pid,ppid,wchan,command
psh> diff -y test1 test1
ls                                                   ls
ls -l                                                ls -l
ps -o state,pid,ppid,wchan,command                   ps -o state,pid,ppid,wchan,command
diff -y test1 test1                                  diff -y test1 test1psh>
```

Example output using test2

```
python3 psh.py < test2
psh> pwd
/home/robert/Desktop/A1
psh> cd /bin
psh> pwd
/bin
psh> cd /doesnotexist
cd: /doesnotexist: No such file or directory
psh> pwd
/bin
psh> cd
psh> pwd
/home/robert/Desktop/A1
psh> cd /usr/bin
psh> pwd
/usr/bin
psh> cd ../..
psh> pwd
/
psh>
```

Example output using test3

```
python3 psh.py < test3
psh> history
    1: history
psh> ls -l
total 60
-rw-rw-r-- 1 robert robert    0 Jul 28 21:46 A1 2007
-rw-rw-r-- 1 robert robert    0 Jul 28 21:46 A1 handout.pages
…
-rw-rw-r-- 1 robert robert  275 Jul  3 13:53 worker.py
psh> h
    1: history
    2: ls -l
    3: h
psh> h 2
total 60
```

```
-rw-rw-r-- 1 robert robert    0 Jul 28 21:46 A1 2007
-rw-rw-r-- 1 robert robert    0 Jul 28 21:46 A1 handout.pages
…
-rw-rw-r-- 1 robert robert  275 Jul  3 13:53 worker.py
psh> h
    1: history
    2: ls -l
    3: h
    4: ls -l
    5: h
psh> history 4
total 60
-rw-rw-r-- 1 robert robert    0 Jul 28 21:46 A1 2007
-rw-rw-r-- 1 robert robert    0 Jul 28 21:46 A1 handout.pages
…
-rw-rw-r– 1 robert robert  275 Jul  3 13:53 worker.py
psh> history
    1: history
    2: ls -l
    3: h
    4: ls -l
    5: h
    6: ls -l
    7: history
psh>
```

Example output using test4

```
python3 psh.py < test4
psh> ls -l | wc
     18     157     928
psh> ls -l | grep ^d
psh> ls -al | grep ^d
drwxrwxr-x 2 robert robert 4096 Jul 28 21:46 .
drwxr-xr-x 3 robert robert 4096 Jul 28 21:46 ..
psh> ls -al | grep ^d | wc -l
2
psh> ls -l|echo ok
ok
psh> ls -l | echo bad |
Invalid use of pipe "|".
psh> | ls -l
Invalid use of pipe "|".
psh> ls -l || wc
Invalid use of pipe "|".
psh>
```

Example output using test5

```
python3 psh.py < test5
psh> perl -e "for ($c = 2; $c > 0; $c--) { print 'x'; sleep 1 }"
xxpsh> jobs
psh> ps -o state,pid,ppid,wchan,command
S   PID  PPID WCHAN  COMMAND
S  3909  3899 wait   bash
S  5234  3909 wait   python3 psh.py
R  5236  5234 -      ps -o state,pid,ppid,wchan,command
psh> perl -e "for ($c = 2; $c > 0; $c--) { print 'x'; sleep 1 }"&
[1]   5237
psh> jobs
[1] <Sleeping> perl -e "for ($c = 2; $c > 0; $c--) { print 'x'; sleep 1 }"&
psh> ps -o state,pid,ppid,wchan,command
S   PID  PPID WCHAN  COMMAND
S  3909  3899 wait   bash
S  5234  3909 wait   python3 psh.py
S  5237  5234 hrtime perl -e for ($c = 2; $c > 0; $c--) { print 'x'; sleep 1 }
R  5239  5234 -      ps -o state,pid,ppid,wchan,command
psh> sleep 4
xxpsh> jobs
[1] <Done> perl -e "for ($c = 2; $c > 0; $c--) { print 'x'; sleep 1 }"&
psh> ps -o state,pid,ppid,wchan,command
S   PID  PPID WCHAN  COMMAND
```

```
S  3909  3899 wait   bash
S  5234  3909 wait   python3 psh.py
R  5242  5234 -      ps -o state,pid,ppid,wchan,command
psh>
```

Example output using test6

```
python3 psh.py < test6
psh> perl -e 'for ($c = 2; $c > 0; $c--) { sleep 1 }'
psh> jobs
psh> ps -o state,pid,ppid,wchan,command
S   PID  PPID WCHAN  COMMAND
S  3909  3899 wait   bash
S  5254  3909 wait   python3 psh.py
R  5256  5254 -      ps -o state,pid,ppid,wchan,command
psh> perl -e 'for ($c = 2; $c > 0; $c--) { sleep 1 }'&
[1]   5257
psh> jobs
[1] <Sleeping> perl -e 'for ($c = 2; $c > 0; $c--) { sleep 1 }'&
psh> ps -o state,pid,ppid,wchan,command
S   PID  PPID WCHAN  COMMAND
S  3909  3899 wait   bash
S  5254  3909 wait   python3 psh.py
S  5257  5254 hrtime perl -e for ($c = 2; $c > 0; $c--) { sleep 1 }
R  5259  5254 -      ps -o state,pid,ppid,wchan,command
psh> sleep 3
[1] <Done> perl -e 'for ($c = 2; $c > 0; $c--) { sleep 1 }'&
psh> jobs
psh> ps -o state,pid,ppid,wchan,command
S   PID  PPID WCHAN  COMMAND
S  3909  3899 wait   bash
S  5254  3909 wait   python3 psh.py
R  5262  5254 -      ps -o state,pid,ppid,wchan,command
psh> perl -e 'for ($c = 10; $c > 0; $c--) { print $c }' | wc &
[1]   5263
psh> jobs
      0       1      11
[1] <Zombie> perl -e 'for ($c = 10; $c > 0; $c--) { print $c }' | wc &
psh> ps -o state,pid,ppid,wchan,command
S   PID  PPID WCHAN  COMMAND
S  3909  3899 wait   bash
S  5254  3909 wait   python3 psh.py
R  5266  5254 -      ps -o state,pid,ppid,wchan,command
[1] <Done> perl -e 'for ($c = 10; $c > 0; $c--) { print $c }' | wc &
psh> sleep 1
psh> ps -o state,pid,ppid,wchan,command
S   PID  PPID WCHAN  COMMAND
S  3909  3899 wait   bash
S  5254  3909 wait   python3 psh.py
R  5268  5254 -      ps -o state,pid,ppid,wchan,command
psh>
```