CS2106: Operating Systems
Lab Assignment 2 – Process Operations in UNIX

Important:

- **The deadline of submission through LumiNUS is 25th September 2pm**
- The total weightage is 5% (+1% bonus):
    - Exercise 1: 1% **[Lab Demo Exercise]**
    - Exercise 2: 2 %
    - Exercise 3: 2 %
    - Exercise 4: 1 % [**Bonus marks**]
- **You must ensure the exercises are implemented in C and work properly on the lab machines (i.e. Linux on x86)**

Warning:
- Beware of the ***fork bomb***! **Do NOT** try on sunfire server. A buggy program can easily bring down your account / whole server ☺.

Section 1. Exercises in Lab 2

The purpose of this lab is to learn about process operations in Unix-based OS. There are **four exercises** in this lab. The first exercise, which is the demo exercise, requires you to try out file manipulation for multiple processes. The remaining three exercises will see you building a simple shell interpreter. Exercises 2, 3, and 4 are described in section 2.

1.1 Exercise 1 [Lab Demo Exercise]

In this exercise, you will use `fork()` and a few other system calls related to file manipulation to investigate the outcome of multiple processes reading from a file. The following actions should be performed by your program:

1. The program takes in a file name and an integer `n` as command line arguments
2. Open the file for reading using the system call `open`
3. Read 1 character from the file using the system call `read`
4. Create `n` child processes
5. All processes read from the file one character at a time and print at ~~STDIN~~ using `write`
6. Each process sleeps for 1ms between read operations (use `usleep`)
7. Once the end of file is reached, the processes close the file and end their execution (exit)
8. Parent waits for the child processes
9. Repeat steps 2-8 using `fopen`, `fread`, `fwrite`, `fclose` instead.

You may find information about `open, read, write, close` using: "`man 2 read`" in a terminal. Explain the behavior that you observe.

Sample Input:

```
./ex1 file_abc        3              //read file_abc from 3 child processes
```

Sample Output:

```
Parent [815]: a
Child 1[818]: b
Parent [815]: c
Child 3[820]: d
Child 2[819]: e        //Order among child and parent processes is not fixed
Parent: Child 2[819] done.
Parent: Child 1[818] done.
Parent: Child 3[820] done.
Parent [815]: a
Child 1[822]: b
Child 1[822]: c
Child 2[823]: b
Parent [815]: b
Child 3[824]: b
Child 2[823]: c
Child 1[822]: d
Child 3[824]: c
Child 1[822]: e
Child 2[823]: d
Parent [815]: c
Child 3[824]: d
Parent [815]: d
Child 3[824]: e
Parent [815]: e
Child 2[823]: e        //Order among child and parent processes is not fixed
Parent: Child 2[823] done.
Parent: Child 1[822] done.
Parent: Child 3[824] done.
Parent: Exiting.
```

Note that the order in which the child processes print out messages is not fixed, so your output may not exactly match the sample session. Due to the nature of the output, there is **no sample test case provided. [**Testing hint: use the "**`usleep()`**" command to introduce random small delays for the child processes. Pay attention to the order of the messages.**]**

Section 2. Command Line Interpreter (Exercise 2 and 3)

A command line interpreter (aka command prompt or shell) allows the execution of multiple user commands with various number of arguments. The user inputs the commands one after another, and the commands are executed by the command line interpreter. You are going to implement an interpreter, named Genie, with various features in the remaining exercises. In exercise 2, Genie will get one simple functionality. You should take the opportunity to plan and design your code since the Genie will be extended in exercise 3.

Due to the nature of the interpreter, it is hard to test using sample input/output. So, instead of sample test cases, a sample usage session is included in the later sections of this document.

2.1 Exercise 2 (Basic Genie)

Let us implement a simple working interpreter with limited features in this exercise.

General flow of basic interpreter
1. Prompt for user request. 2. Carry out the user request. 3. Unless user terminates the program, go to step 1.

TWO possible user requests:
1. Quit the interpreter (a.k.a. put Genie back in the bottle). Format: `quit` **Behavior:** • Prints message "Goodbye!" and terminates the interpreter.

2. Run a command (ask your Genie to fulfill a wish). Format:
`command [arg1 [arg2 [arg3 …]]]`
//Read the path of the command and the arguments
//The `command` is assumed to be less than 19 characters
//The arguments are separated by character space.
//Number of **arguments** is assumed to be less than 10 arguments
//Each **argument** is assumed to be less than 19 characters
 	
Behavior:
a. The command can be a command path or a simple command (such as `ls` in shell).
• If the command path is not provided, you may assume that the command is located in /bin.
b. If the command exists, run the command in a child process

- Wait until the child process is done

c. Else print error message "**`XXXX not found`**", where **`XXXX`** is the user command.

For 2, you need to check whether a **command_path** exists (i.e. valid). This can be achieved by various methods, one simple way is to make use of the library call **`stat()`**. Find out more about this function by "**`man -s2 stat`**". This library function has various usages, but you don't need to investigate all of them. Moreover, you don't need to have the full knowledge to use this call effectively. [Hint: look carefully at the return type of this function.]

Make use of the **`fork()`** and **`execl()`** combo to run valid commands. The **`execl()`** function call discussed in the lecture is sufficient for this exercise, but you are free to use any version of `exec`.

Just like a real shell interpreter, Genie will wait until the command finishes before asking for another user request.

Handling input and output for Genie's commands:

- The **standard output** of the command (in 2.) must be printed by Genie as standard output.
- The **standard error** of the command must be printed by Genie as standard error.
- No need to support redirecting Genie's standard input as the command's standard input (commands do not require reading from standard input).

Assumptions:

a. Non-terminating commands will NOT be tested on your interpreter

b. No `ctrl-z` or `ctrl-c` will be used during testing. So you do not need to use signal.

c. You can assume user requests are "syntactically correct", but the commands might not exist.

You can assume each `command and arguments` have less than 19 characters. No more than 10 arguments will be provided for a command.

Suggestions on approach:

- Modularize your code! Try to find the correct code for each of the functionality independently. Test the function thoroughly then work on another part of the program. This allows your code to be reused in exercise 3.

Sample Session:

User Input is shown as **bold**. Your Genie awaits your command by showing the prompt "`GENIE > `" ☺. Note that additional empty lines are added in between of user inputs to improve readability.

```
GENIE > /bin/ls                       //See note*
a.out     ex2.c     …                 //output from the "ls" command

GENIE > ls -al            //Use ls from /bin with arguments

GENIE > ps
PID TTY          TIME CMD
    4 tty1     00:00:00 bash
   34 tty1     00:00:00 ps

GENIE > ps -o ppid -o pid //Use ps with arguments
PPID   PID
    3     4
    4    37

GENIE > ps-l
/bin/ps-l not found          // "ps-l" does not exist

GENIE > quit
Goodbye!                              // Interpreter exits
```

Note*: The "**`ls`**" command may be located at a different location on your system. Use "**`whereis ls`**" to find out the correct path for your system.

2.2 Exercise 3 (Advanced Genie)

This exercise extends the capabilities of the interpreter from exercise 2. Genie is enhanced as shown below:

Possible user requests:
1. [No change] Quit the interpreter. Format: **`quit`** //No change from ex2.
2. [No change] Run a command (ask your Genie to fulfill a wish). Format: **`command [arg1] [arg2] [arg3] …`** //No change from ex2.
3. **[NEW]** Pipe between two commands. Format: **`command1 [arg1 to arg10] | command2 [arg1 to arg10]`**   **Behavior:**   a. If the specified commands exist, run the commands in child processes with the supplied command line arguments.   b. The output from **command1** is used as input for **command2**. To achieve this, use `pipe` and `dup` system call (from Lecture 4)   c. The interpreter will   • Wait until the child processes are done.   d. If any of the commands do not exist, print error message "**`XXXX not found`**", where **`XXXX`** is the user entered command.

e. Note that this is a subset of the case with pipes among multiple commands (point 4)

4. **[NEW]** Pipes among multiple commands. Format:

```
command1 [arg1 to arg10] | command2 [arg1 to arg10]
| command3 [arg1 to arg10] | command4 [arg1 to arg10]
| command5 [arg1 to arg10] …
```

Behavior:

a. If all specified commands exist, run the commands in child processes with the supplied command line arguments. There will be at most 10 chained jobs in a command.

b. For a chain of jobs, for each pipe ("|"), the output of the job before the pipe is input for job the after the pipe.

c. The interpreter will
   - Wait until the child processes are done.

d. If any of the commands does not exist, print error message "XXXX not found", where XXXX is the user entered command. No other jobs should be executed.

Assumptions:

a. Non-terminating command will NOT be tested on your interpreter.

b. No `ctrl-z` or `ctrl-c` will be used during testing.

c. **Each** command line argument has less than 19 characters.

d. There are **at most 10 command line arguments** for each job (commands)

e. There are **at most 10 chained jobs** in a command.

d. **You can assume all user requests are "syntactically correct"**, but the commands might not exist.

Notes:

- You need to learn a few C library calls by exploring the manual pages. Most of them are variants of what we discussed during the lecture.
- You may use any function from `exec` family to solve this question.
- Remember to use the **`wait()`** system call in exercise 2.

Suggestions on approach:

- This exercise is challenging. Again, implementing the required functionalities incrementally is the best approach.
- You will need to manipulate string to some extent. Try to write a program just to test out your code first. [Warning: String manipulation is pretty frustrating in C. Don't be demoralized. ☺]

Sample Session:

User Input is shown as **bold**.

```
GENIE > /bin/ls -al    //list files in the current folder
total 32              //output from the "ls" command
drwxrwxr-x 2 ccris ccris 4096 Sep  3 15:51 .
drwxrwxr-x 3 ccris ccris 4096 Aug 31 12:42 ..
-rw-rw-r-- 1 ccris ccris   55 Sep  3 15:51 file
-rwxrwxr-x 1 ccris ccris 9208 Aug 31 11:09 filefork
-rw-rw-r-- 1 ccris ccris  823 Aug 31 11:09 filefork.c
-rw-rw-r-- 1 ccris ccris  534 Aug 31 12:34 genie.c


GENIE > cat filename |tail -n 1 //list last line of
                                //content from filename
JKLMNOPQRSTUVWXYZ               //output

GENIE  >  cat  file  |  grep  a     //search  for  lines  that
contain
                       //"a" in "file"
abcdefghijklmnopqrstuv

GENIE > ls -al | head -2  //list first two lines from
                          //the output of ls -al
total 32
drwxrwxr-x 2 ccris ccris 4096 Sep  3 15:51 .

GENIE>  ls  |  cat  -n  |  wc  -w     //list  the  files,  print
their
           //names preceded by index, and count the words
8

GENIE > ls -l | wc -l | echo "Genie is here"
Genie is here

GENIE> quit
Goodbye!                       // Interpreter exits
```

2.3. Exercise 4 (Enhanced Genie)

This is a bonus exercise. It is not mandatory to solve it! This bonus mark can be used to offset any other marks you may lose in lab assignments, but it cannot be used to offset marks that you lose in midterm or final assessment paper.

Your Genie needs to remember some things now. In your advanced Genie (Exercise 3), implement support for setting, using and unsetting of environment variables as follows:

- An environment variable is set using the special command "`set`".
- To unset it, you must support the special command "`unset`".
- You must support evaluating the value of an environment variable, when it is passed as an argument to a command, prefixed by the special character '`$`'.
- The environment variables are local to your interpreter, and until unset, can be used by any command executed by your shell.

Additional user requests:
1. [NEW] Set an environment variable: **`set NAME value`** **Behavior:** Sets environment variable `NAME` to value `value`.
2. [NEW] Unset an environment variable: **`unset $NAME`** **Behavior:** Unsets the value for variable `NAME`
3. [NEW] Use an environment variable in any command: **`$NAME`** **Behaviour:** A variable can be used in any command in the interpreter using its name prefixed by **`$`**. The variable is replaced by its value. Assume that $ is not used by any other command.

User Input is shown as **bold**.

```
GENIE > set ENV1 filename   //set the environment variable
                            //ENV1 to be "filename"

GENIE > echo $ENV1    //print ENV1
filename

GENIE > cat $ENV1 |tail -n 1 //list last line of
                             //content from filename
JKLMNOPQRSTUVWXYZ            //output


GENIE > unset $ENV1   //unset $ENV1

GENIE > echo $ENV1    //print ENV1
                      //no output

GENIE> quit
Goodbye!                      // Interpreter exits
```

Section 3. Submission

Zip the following files as E0123456.zip (**use your NUSNET id, NOT your student no A012…B, and use capital 'E' as prefix):**

a. **`ex2.c`**
b. **`ex3.c`**
c. **`ex4.c`**

Do **not** add additional folder structure during zipping, e.g. do not place the above in a "lab2\" subfolder etc.

Upload the zip file to the "Student Submission→Lab 2" workbin folder on LumiNUS. Note the deadline for the submission is **25th September, 2pm**.

Please ensure you follow the instructions carefully (output format, how to zip the files etc). **Deviations will be penalized.**

**~~~ Your Genie awaits your command! ~~~**