CS2106: Operating Systems
Lab 2 – Process Operations in UNIX

Important:

- **The deadline of submission to Luminus is 5th March 2359**
- The total weightage is **7% + 1%**:
  - Exercise 1: 1% **[Lab Demo Exercise]**
  - Exercise 2: 2 %
  - Exercise 3: 4 %
  - Bonus Exercise: 1 %

Warning:

- Beware of the ***fork bomb***! Exercise 2 has some tips reducing the chance of igniting a fork bomb by accident.

Section 0. Basic Information

Please refer to the "Introduction to OS Labs" document if you have forgotten how to unpack / setup the lab exercises.

You only need to do this if you have just formed a team for lab 02. If you have already registered a team for lab 01, there is **no need** to re-register. Please use the google form to register your team:

https://forms.gle/2XGm3zFYpXdFTgpc6

Select your name and your buddy's name from the dropdown list to form a team. Your team will be in effect **from this lab onward**.

Section 1. Exercises in Lab 2

The purpose of this lab is to learn about process operations in Unix-based OS. There are a total of **three exercises** in this lab. The first exercise, which is the demo exercise, requires you to try out two system calls covered in lecture 2b. The remaining two exercises will see you building a simple shell interpreter. Since exercises 2 and 3 are quite involved, we will describe them in a separate section (section 2). We have an additional challenge for you in the **bonus** section if you breeze through the exercises.

1.1 Exercise 1 [Lab Demo Exercise]

In this exercise, you will use a combination of `fork()` and `waitpid()` to perform simple process creation and synchronization. The `waitpid()` library call is a variant in the `wait()` family which enables us to wait on a specific pid. Don't forget to check the manual page ("`man -s2 waitpid`"). The requirement is best understood through a sample session.

Sample Input:	
3	//3 child processes

Sample Output:

```
Child 1[818]: Hello!      //818 is the process id of child
Parent: Child 1 [818] done.
Child 3[820]: Hello!
Child 2[819]: Hello!      //Order between child processes is not fixed
Parent: Child 2[819] done.
Parent: Child 3[820] done.
Parent: Exiting.
```

The program reads in an integer **N** between 1 to 9 (inclusive) which indicates the number of child processes to spawn. Each of the child processes print out a "`Child X[PID]: Hello!`" message, where X is the index number and PID is the process id of the child. The parent process will then check for the termination for each of the child and print out the corresponding message, "`Parent: Child X[PID] is done.`". **The parent process should spawn all child processes before checking on them.** This check is performed in the spawning order, i.e. Child **1, 2, … N**.

Note that the order in which the child process print out message is not fixed, so your output may not exactly match the sample session. However, certain order must always be respected, e.g., Child X must print the "Hello" message before the parent can print out the matching "Child X is done" message. Due to the nature of the output, there is **no sample test cases provided. [**Testing hint: use the "**`sleep()`**" library call to introduce random small delays for the child processes. Pay attention to the order of messages.**]**

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

By now you have issued a number of commands through the shell command prompt on the Compute Cluster nodes. The shell prompt is also known as ***command line interpreter***. It is actually just another program implemented using the process-related system calls discussed in lecture. In the remaining exercises, you are going **to implement an interpreter** with various features. In exercise 2, only a few simple functionalities are required. You should take the opportunity to plan and design your code in a modular way, **since the same code can be reused** in exercise 3.

Note that when you unpack the directories for these two exercises, you'll find several C files and a file named "**`makefile`**" in each exercise directory in addition to the usual skeleton files **`ex2.c`** and **`ex3.c`**. These extra C files are tiny programs with various runtime behavior to aid your testing later. To build these programs, simply enter "**`make`**" command under the directories. The "**`make`**" utility invokes the **`gcc`** compiler automatically to prepare the executables for you.

Other than exhibiting interesting execution behavior, you can also learn a few useful tricks from these programs. Feel free to open them up and browse through the code.

Summary of the extra programs:

`clock`	Prints out a message after **`Y`** seconds and repeat for **`X`** times. **`X`** and **`Y`** can be specified as command line arguments. See given code for more details.
`alarmClock`	Prints out a message after **`X`** seconds and terminates. **`X`** can be specified as command line argument. See given code for more details.
`infinite`	Goes into infinite loop, never terminates.
`return`	Return the user specified number **`X`** to the shell interpreter.
`showCmdArg`	Shows the command line arguments passed in by user.
`stringTokenizer`	Shows how to split user input into subparts (tokens).

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 Interpreter)

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:
a. Quit the interpreter. Format: `quit` **Behavior:** • Print message "Goodbye!" and terminates the interpreter.
b. Print the current **search path**: `showpath` **Behavior:** • Print the current **search path** (see (d) for the usage of this **search path**) • By default, the search path is "." (i.e., the current directory). User can change it using the command `setpath` below.
c. Change the current search path: `setpath user_supplied_path` **Behavior:** • Change the current **search path to** ***user_supplied_path*** • You can assume the search path contains less than 20 characters

d. Run a command. Format:
`command`
//Read command to be executed
//The `command` is assumed to be less than 20 characters
 	
Behavior:
1. If **path/command** exist, run the command in as a child process. The "**path**" is the current search path (e.g., can be the default "." or any path set by user via command (c) above.
• Wait until the child process is done
2. Else print error message "**`XYZ not found`**", where **`XYZ`** is the user command.

For (d), you need to check whether the **path/command** exists (i.e., valid). This can be achieved by various methods, one simpler way is to make use of a library call:

```
stat()
```

Find out more about this function by "`man -s2 stat`". This library function has various usages, and some are quite involved. However, 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. Take a good look at the various parameters of the `execl()` before you start.

Just like a real shell interpreter, **your program will wait until the command finishes before asking for another user command.**

Assumptions:
a. Non-terminating command will NOT be tested on your interpreter
b. No `ctrl-z` or `ctrl-c` will be used during testing. [If you want to tackle this, take a look at the *bonus* section.]
c. You can assume user requests are "syntactically correct".
You can assume the path/command together has less than 40 characters.

Suggestions and Cautions:

- Modularize your code! Read through exercise 3 too and plan in advance. Try to find the correct code for each of the functionalities independently. Test each function thoroughly before proceeding with another part of the program. This allows your code to be reused in exercise 3.

- Beware of **fork bomb.** Add in the "`fork`()" call only after you have thoroughly tested the basic code. You may want to test it separately without putting it in a loop first. For any child process, make sure you have a "`return` …" or "`exit`()" as the last line of code (even if there is an `execl`() before as the `execl`() can fail ☺).

- If you accidentally ignited a fork bomb ☹, your compute cluster node account may be frozen and the compute node may go down. Please contact SoC Technical Service techsvc@comp.nus.edu.sg to ask them (nicely!) to unfreeze your account and kill off all your processes.

Sample Session:

User Input is shown as **bold**. Additional comments are shown in *italic*. The prompt is "YWIMC", short for "Your whim is my command" ☺, just a way to show your shell interpreter is a devoted servant. Note that additional empty lines are added in between of user inputs to improve readability.

```
YWIMC > showpath
.                           //the default path is "."
YWIMC > clock               //effectively running "./clock" as the search
                            //path is automatically added in front
Reporting every 1 seconds for 5 times
Report No.1: 1 seconds elapsed!
Report No.2: 1 seconds elapsed!
Report No.3: 1 seconds elapsed!
Report No.4: 1 seconds elapsed!
Report No.5: 1 seconds elapsed!

YWIMC > anything
"./anything" not found //not a valid command

YWIMC > ls
"./ls" not found            //"ls" cannot be found at "./"

YWIMC > setpath /usr/bin    //see *note below

YWIMC > showpath
/usr/bin

YWIMC > ls                  //this works now as "/usr/bin/ls" is valid
a.out        alarmClock.c   ... <other files not shown>

YWIMC > clock
"/usr/bin/clock" not found

YWIMC > quit
Goodbye!
```

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.

Note:** This simple interpreter shows the idea behind **search path under Unix/Linux shell**. When you type in a command, the shell will automatically look for the command in a number of user-defined locations, known as the ***search path***. For example, you can type "echo $PATH" to see the search path for your shell on Sunfire / Compute Cluster node. We heavily simplify this idea to use a single search location only, while the actual search path can contain a number of different locations.

2.2 Exercise 3 (Advanced Interpreter)

This exercise extends the capabilities of the interpreter from exercise 2. Please note the enhanced or new requests below:

SIX possible user requests:

a. [No change] Quit the interpreter. Format:

`quit` //No change from ex2.

b. **[Enhanced]** Run a command. Format:

`command [arg1 to arg4]`

//the user can supply **up to 4 command line arguments**

Behavior:

a. If the specified command exists at **path/command**, run the command in a child process with the supplied command line arguments. You can assume each argument is no more than 10 characters long.
   - Wait until the child process is done.
   - Capture the child process' return value (see "`result`" command).

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

c. **[New]** Run a command in the background. Format:

`command [arg1 to arg4] &`

//Note the "**&**" symbol at the end of the command. The user can supply **up to 4 command line arguments, same as (b).**

Behavior:

a. If the specified command exist at **path/command**, run the command in a child process with the supplied command line arguments.
   - Print "`Child XXXX in background`", `XXXX` should be the PID of the child process.
   - Continue to accept user request.

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

d. **[New]** Wait for a background process. Format:

`wait job_pid`

//"wait" followed by an integer `job_pid`

Behavior:

1. If the `job_pid` is a valid child pid (generated by the request in (c)) and has not been waited before:
   - Wait on this child process until it is done, i.e., the interpreter will stop accepting request.
   - Capture the child process' return value (see "`result`" command).

2. Else print error message "**`XXXX not a valid child pid`**", where **`XXXX`** is **`job_pid`** entered by user.

e. **[New]** Print the pids of all unwaited background child process. Format:

`pc` //short for `print child`

Output format:

`Unwaited Child Processes:` //Just an output header

`<Pid of Unwaited Child 1>` //May be empty if there is no

`<Pid of Unwaited Child 2>` // unwaited child

.....

Behavior:

a. PID of all "unwaited" background child processes (including terminated ones) are printed.

f. **[New]** Print the return result (i.e. exit status) of the child process just ended (foreground child) or just waited (background child). Format:

`result`

Output format:

`<return result>` //A single integer number

Behavior:

a. This command is **only valid after a successful (b) or (d)**.

Assumptions:

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

b. No `ctrl-z` or `ctrl-c` will be used during testing. As we have not covered signal handling in Unix, it is hard for you to take care of these at the moment.

c. **Each** command line argument for (b) and (c) has less than 20 characters.

d. There are **at most 10 background jobs** in a single test session.

e. **You can assume all user requests are "syntactically correct".**

Notes:

- You need to learn a few C library calls by exploring the manual pages. Most of them are variants of what we discussed in lecture.
- Instead of **`execl()`**, it is easier to make use of **`execv()`** in this case. Read up on **`execv()`** by "**`man -s2 execv`**".
- Remember to use the **`waitpid()`** call in exercise 1.
- You only need a **simple way** to keep track of the background job PIDs. (hint: a simple array will do….).

Suggestions on approach:

- This exercise is fairly involved. Again, implementing the required functionalities incrementally is the best approach.
- You will need to manipulate strings 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. ☺] [Hint: take a look at the sample programs…]

Sample Session:
User Input is shown as **bold**.

```
YWIMC > showCmdArg cs2106 is fun //Multiple command line arguments
Arg 0: showCmdArg
Arg 1: cs2106
Arg 2: is
Arg 3: fun

YWIMC > alarmClock 30 &   //Run alarmclock in the background
Child 1178045 in background //Your PID can be different

YWIMC > showCmdArg still work with no wait //Interpreter continue
Arg 0: showCmdArg                          //to work without waiting
Arg 1: still                               //for alarmClock
Arg 2: work
Arg 3: with
Arg 4: no
Arg 5: wait

YWIMC > wait 12345          //Attempt to wait on invalid child PID
12345 not a valid child pid

YWIMC > pc                  //Show all currently unwaited child
Unwaited Child Processes:
1178045

YWIMC > wait 1178045        //Wait for the alarmClock. Use the PID
Time's Up! 30 seconds elapsed   // you see in the previous step

YWIMC > result
0

YWIMC > return 123 //The "return" command simply return the argument

YWIMC > result      //Check the return value of previous command
123

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


Note*: The background job printing will intermix with your interpreter input / output. There is no need (and no way ☺) to make background job print "nicely".

2.3 Bonus Exercise (Interrupting the Interpreter)

As an additional challenge, let us add the ability to handle *signal* to the interpreter. For simplicity, swe are going to handle only one signal known as `SIGINT` (Interrupt Signal). This is the signal generated when you press ctrl-C on the keyboard, commonly used to stop the current running program.

First, browse to the `bonus/` folder and perform a "`make`". Other than the tiny programs used in exercises 2 and 3, there are now two more programs:

`undead`	A program to demonstrate how to capture the <Ctrl-C> SIGINT signal. This program goes into infinite loop when you run it, pressing <Ctrl-C> actually **wont kill it**. (That's why it is an undead). Read the code and explore how this is achieved. [Tips for killing undead: Ctrl-Z to pause it, "ps" to find its pid, use "kill <pid>" to kill it.
`easyTarget`	This program goes into infinite loop and will print out a message "*Someone killed me!*" when you kill it using <Ctrl-C>. Useful for testing later.

Now, what we want to mimic is the common behavior of a shell interpreter:

- If the shell is currently running a **foreground process**, pressing Ctrl-C will cause the foreground process to be killed.
- Otherwise, pressing Ctrl-C has **no effect**. We will print out a message "Nothing to kill." to show that the Ctrl-C is captured properly.

Sample Session:
User Input is shown as **`bold`**.

```
YWIMC > ^CNothing to kill.   //Presss Ctrl-C when nothing is running
anything                     //User type in "anthing"
"./anything" not found
YWIMC > easyTarget           //easyTarget runs forever
^CSomeone killed me!       //Press Ctrl-C cause shell to kill easyTarget
YWIMC > easyTarget &       //Run easyTarget in the background
Child 1159811 in background
YWIMC > ^CNothing to kill. //Ctrl-C does not work on background child
pc
Unwaited Child Processes:
1159811
YWIMC > wait 1159811   //now easyTarget is the foreground process
^CSomeone killed me! //Ctrl-C now can killl easyTarget
YWIMC > quit
Goodbye!
```

To attempt this bonus exercise, simply add necessary code to your **ex3.c.**
Hint: You need to learn how *signal handler* and *signal sending* works in Linux.

Section 3. Submission Procedure

Since exercise 3 is a superset of exercise 2, you can submit **only `ex3.c`.** In case you are not able to finish ex3, **please copy your `ex2.c` to into folder `ex3/` as `ex3.c` and submit `ex3.c`**. Please make sure your source code compiles.

We will check the submitted **`ex3.c`** according to the features listed in each section, e.g. passing all features in exercise 2 grant you 2 marks, further supporting features in exercise 3 grant you additional 4 marks. This applies for bonus feature as well. **Hence, it is better submit only features working fully, rather than submitting partial code for all features but none working correctly.**

Zip the following folders and files as `A0123456X.zip` (**use your student id with "A" prefix and the alphabet suffix**). One simple way is to perform the following command **at the parent folders of the ex?/** folders (i.e. at the **`L2/`** folder):

```
zip A0123456X.zip ex3/ex3.c
```

Remember to modify the archive name and the content accordingly as needed.

A0123456X.zip should contain 1 folder with the following content:

```
ex3/
    ex3.c
```

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

3.1 Do a quick self-check

We have provided a self-checking shell script to help validating your zip archive. The script checks the following for this lab:

a. The name or the archive you provide matches the naming convention mentioned above.
b. Your zip file can be unarchived, and the folder structure follows the structure described above.
c. All files for each exercise with the required names are present.
d. Each exercise can be compiled and/or executed.

Note that we won't be able to run test cases due to the nature of this lab. Once you have produce the zip file, you will be able to check it by performing the following steps at the **`L2/`** folder:

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh A0123456X.zip (replace with your zip file name)
```

The check_zip script will print out the status of each of the checks. Successfully passing checks enable the lab TA to focus on grading your assignment instead of performing lots of clerical tasks (checking filename, folder structure, etc). Please note that **points might be deducted if you fail these checks.**

Expected Successful Output
Checking zip file.... Unzipping file: ***A0123456X***.zip ex3: Success

3.2 Upload to LumiNUS before deadline!

After verifying your zip archive, please upload the zip file to the "Student Submission→Lab 2" file folder on LumiNUS. Please note that **late penalty is quite steep,** so submit early to avoid last minute issues.

Also, remember that if you are in a team, **only one member need to submit into LumiNUS. The archive name should be in the format A0123456X.zip, where A0123456X stands for the student number of *any of the team members*; we will do "background processing" to match with your team members automatically. Both team members will receive the same score for exercises 2, 3 and bonus.**

~~~ Your whim is my command! ~~~