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

Important:

- **This is NOT the actual lab 2 specification. Just to show you the original draft which is quite a bit more involved than the current version. (Check out ex2 and ex3). ☺**
- **You are welcome to give it a try as they are actually compatible with the current released Lab 2.**

Warning:

- Beware of the ***fork bomb***! **Do NOT** try on sunfire server. A buggy program can easily bring down the 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 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).

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 will read 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 "Child X is done" message. Due to the nature of the output, there is **no sample test cases provided. [**Testing hint: use the "**`sleep()`**" command 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)

As shown during the lecture, a command line interpreter (aka command prompt or shell) involves most of the major process operations. So, you are going to implement an interpreter with various features in the remaining exercises. In exercise 2, only a few simple functionalities are required. You should take the opportunity to plan and design your code since the same code can be reused in exercise 3.

Note that when you unpacked the directories for these two exercises, you'll find several C files and a file named "**`makefile`**" in each exercise directories in additional to the usual skeleton files **`ex2.c`** and **`ex3.c`**. These extra C files are tiny program with different 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.

Summary of the extra programs:

`clock`	Prints out a message after **`X`** seconds and repeat for **`Y`** 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 (token).

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.

THREE possible user requests:
a. Quit the interpreter. Format: `quit` **Behavior:** • Print message "Goodbye!" and terminates the interpreter.

b. Run a command. Format:
`command_path`
//Read the path of the command
//The `command_path` is assumed to be less than 20 characters
 	
Behavior:
a. If command exist, run the command in a child process
• Wait until the child process is done
b. Else print error message "**`XXXX not found`**", where **`XXXX`** is the user command.

c. Repeat the last command. Format: `last` **Behavior:** • Repeat the last user command (even if it was invalid). • Ignore if there is no last command.

For (b), you need to check whether the ***command_path*** exists (i.e. valid). This can be achieved by various methods, one simple 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. When using **`execl()`**, you can assume that the path of the program is the same as command line argument zero (i.e. name of the program).

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

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. You can assume user requests are "syntactically correct".
You can assume the `command_path` has less than 20 characters.

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

YWIMC > last
a.out      ex2.c      ...          //repeat the "ls" command

YWIMC > ./alarmClock               //Use the provided program
Time's up. 3 seconds elapsed.          //after ~3 second delay

YWIMC > ./anything
./anything not found               // "./anything" does not exist

YWIMC > last
./anything not found               // repeat command even if it was
                                   //  invalid
YWIMC > 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 Interpreter)

This exercise extends the capabilities of the interpreter from exercise 2. The general flow is the same as the basic interpreter in ex2. The changes are in the user requests. Please note the enhanced or new requests below:

FOUR possible user requests:

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

```
quit                    //No change from ex2.
```

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

```
command_path [arg1 to arg4]
```

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

Behavior:

a. If the specified command exists, run the command in a child process with the supplied command line arguments
   - 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_path [arg1 to arg4] &
```

//Note the "&" 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 exists, 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:

a. 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)

b. 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:

`printchild`

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 a child process. Format:

`result`

Output format:

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

Behavior:

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

g. [No change] Repeat the last command. Format:

```
last          //No change from ex2. Note that all user commands from (b)
              //  to (f)can be repeated.
```

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 to help.
- You only need a **simple way** to keep track of the background job PIDs.

Suggestions on approach:

- This exercise is fairly involved. Again, implement 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. ☺] [Hint: take a look at the sample programs…]

Sample Session:

User Input is shown as **bold**.

```
YWIMC > /bin/ls          //Path of ls may be different on your system
a.out      ex3.c ......          //output from the "ls" command

YWIMC > /bin/ls -l          //same as executing "ls –l"
total 144
-rwx------   1 sooyj    compsc       8548 Aug 13 12:06 a.out
.....................          //other files not shown

YWIMC > result            //successful "ls" returns 0 to shell
0

YWIMC > ./alarmClock 20 &       //Background job. See Note*
Child 12345 in background       //PID 12345 is just an example

YWIMC > ./showCmdArg one 2 3 four
Arg 0: ./showCmdArg
Arg 1: one
Arg 2: 2
Arg 3: 3
Arg 4: four

YWIMC > last
Arg 0: ./showCmdArg
Arg 1: one
Arg 2: 2
Arg 3: 3
Arg 4: four

YWIMC > wait 12340              //Try to wait for  PID12340
12340 not a valid child

YWIMC > printchild
Unwaited Child Processes:
12345

YWIMC > wait 12345   // Will wait until the previous alarmClock
                     // process terminates. Note: It is possible that
                     // alarmClock has already terminated by this time.
```


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".

```
                                  // In that case, this request will return immediately.

YWIMC > printchild
Unwaited Child Processes:         // empty in this case

YWIMC > ./anything 1 2 3
./anything not found              // "./anything" does not exist

YWIMC > ./return 188 &            // simply return "188" to shell
Child 12366 in background         //PID 12366 is just an example

YWIMC > wait 12366

YWIMC > result                  //show the return result of the waited process
188

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