CS2106: Lab 4 (Graded)

Exercise 1: Infinite Forking! (Not Graded)

Goals: We experiment to see the effect of endless forking.

With this exercise, you may have to reboot your PC if it is running Linux or the VM if Linux is running in VirtualBox. **Do not try** this on the Sunfire server, as it will cause problems to other users.

Read `lab4-forkall.c`, it attempts to create as many processes as possible. Try the following:

- Compile it to `lab4-forkall`. As it is possible that you may need to crash/reboot the system, you should save all files and then run `sync` (see the `sync` man).
- Open 3 windows. In Window 1, run `lab4-forkall`. In Window 2, run `top` which shows you process statistics and the most active processes.
- After a while you will notice that fork is failing. Check the number of tasks and the `%Cpu`.
- You may notice that Linux is running quite sluggishly. Also if you have a laptop, you may hear the laptop fan turning ON. This is because the CPU is now working hard. Try running any commands, e.g. `ls`, in Window 3. You will find that you may have hit the process limit and shell commands now fail - due to failure to create new processes.
- In Window 1, try killing `lab4-forkall` by typing CTRL-C. If that fails, you may need to reboot.

You may want to repeat this experiment a few times to see what is going on. You can also modify the code to change the value of `D` (higher or lower).

Since `lab4ex1` creates many processes. You can study the effect of restricting the total number of processes created. In the `bash` shell, you can use the "`ulimit -u` n" command to limit the number of processes to n. Most likely, your default setting is *unlimited*, meaning that the limit is the system limit (it doesn't mean infinite!). You can try $n = 500$. What happens now when you rerun the experiment. Be careful in setting the limit too low as it cannot be increased, only decreased. This means that you if you want to make a new higher limit, you will need to start a new terminal (e.g. shell) which is not a child of the process with the limit.

Exercise 2: Observing Process Scheduling (Not graded)

Goals: Explore which processes are scheduled

The program `lab4-forkorder.c` attempts to show which process parent or children is running first. It uses the `write()` system call which is unbuffered to write to standard output. Experiment with the commented out code to see if the order changes.

Note that the result of this experiment depends very much on the Unix kernel implementation and can even vary between different kernel versions of Linux.

Exercise 3: A utility for running many processes - Basic (Graded 30%)

Goals: Understanding creating processes, using `fork` and `exec`*XX* (the exec family of system calls)

Please try Exercise 3 first which covers `fork` and `exec` before Exercise 4 which needs more system calls. The percentage mark here reflects that Exercise 3 is easier than Exercise 4.

The task of Exercise 3, which is to write `lab4ex3.c`, is illustrated using the following example:

```
$ ./lab4ex3 /bin/echo going to sleep + /bin/sleep 5 + /bin/echo finished
```

The effect of `lab4ex3` is to mimic running the following 3 programs/processes are in the shell:

```
$ /bin/echo going to sleep
$ /bin/sleep 5
$ /bin/echo finished
```

There 2 binaries used with the full pathname (full filename) are: `/bin/echo` and `/bin/sleep`. In the event, that these standard binaries are installed elsewhere, you can use the `where` command to find the pathname, e.g. `where sleep`. The `echo` program just prints its command line arguments, something like Lab 3, while the `sleep` program waits for the number of seconds given in its argument. So `lab4ex3` runs commands given its in arguments with a '`+`' separating each command. You can assume that for `lab4ex3` that the full pathname for the executable is given. Your code **must** be written to execute the executable using the `execve()` system call.

We will call the command to be executed a "sub-command line"[1] The arguments to `lab4ex3` consists of sub-command lines are separated by `+`. A sub-command line may be empty, in which case, the sub-command is treated as that there is nothing to run for that sub-command, or equivalent to the command `/bin/true` (note that it is not an error). You an assume that the number of words in a sub-command line is at most 8, e.g. `argv[7]` is the max as `argv[0]` is the program name. There will not be test cases which exceed 8 words.[2]

[1] Sub-command because there can be multiple commands. Note that in this context, line, does not mean there is a newline. In the example, it refers to 3 command lines.

[2] It is possible to write more robust code which need not use this assumption, analogous to Lab 1. Students who want more robust code can experiment but this is for your interest.

You will find that creating *executable shell scripts* will be useful for testing your code. Shell scripts were discussed in Tutorial 3. Briefly an executable need not be a native code file but also any file for which you specify an interpreter (and it is the interpreter which is the binary being run). The sample file `lab4-test.sh` will run the `bash shell` on the file with the pathname to `bash` specified in the first line. You will need to make sure that the file permission is set to be executable, e.g. `chmod u+x lab4-test.sh`. Just try running `./labt-test.sh`. Note that in Unix there is no such thing as a file suffix, so the "`.sh`" is only a convention to remind the user that it is a shell script.

Errors:
Some errors can arise during execution. Error messages should be written to the `standard error` (`stderr`) stream. `lab4ex3` should exit on the first error. If the `fork()` fails then the error message should be:

`fork failure` If the `execve()` fails then the error message should be:

`exec failure:` S

where S is the relevant sub-command line.

Hints:
You need to use the `fork()` system call to create a new process. To print to the `standard error` stream use `fprintf()`, e.g.

```
fprintf(stderr, "Value of x=%d\n", x);
```

Exercise 4: A utility for running many processes - Full (Graded 70%)

Exercise 3 is the pre-cursor to Exercise 4, hence, it is from basic to full. The idea of `lab4ex4` is a shell-like tool for running repetitive commands. Note that it has a different command line syntax from `lab4ex3`. The following example illustrates the usage of `lab4ex4`:

```
$ ./lab4ex4 gcc -c %
```

Assume that the `standard input` to the above example is:

```
a.c
b.c
c.c
```

The result would be as if the following was run:

```
$ gcc -c a.c
$ gcc -c b.c
$ gcc -c c.c
```

The example illustrated the use '`%`' which is used to replace an argument with a word from the input line. An input line consists of words (non-space characters) separated by spaces up to a newline ('`\n`'). There can be multiple `%` which are replaced by subsequent words coming from

the input line. If there are more `%` than words in the line, it is then no replacement occurs and the `%` is skipped over. The next example illustrates the use of multiple `%`. Let the command line be:

```
$ ./lab4ex4 gcc % % % % %
-c a.c
-c b.c
-c c.c
-o prog a.o b.o c.o
```

In the example above, the input data follows the shell command line. (You will need to type CTRL-D to end the input). This example is similar to running the following in the shell:

```
$ gcc -c a.c
$ gcc -c b.c
$ gcc -c c.c
$ gcc -o prog a.o b.o c.o
```

We are now ready to define the syntax for the `lab4ex4` arguments. We first introduce the Unix man syntax, only what is needed for this lab (for more details see (`man man`)). Anything enclosed by square brackets is optional, which means it need not be present. *Italics* denote replace by corresponding argument. **Bold** text is to be typed literally. The syntax for the `lab4ex4` command line can be described as:

lab4ex4 [**-j** *m*] [**-p** *newpath*] *prog ...*

The `-j` option specifies that up to m processes can be running the commands resulting from the input. It will try to maximize the number of processes running at any time, e.g. once a process finishes, a new process is created to handle the next command.[3] By default, if `-j` is not given then commands are run sequentially, i.e. $m = 1$. The `-p` option is a new `PATH` given by *newpath*. We saw that `execve()` used in Exercise 3 requires the pathname to the executable. However, notice that in the examples for Exercise 4, the pathname to `gcc` was not given. Some of the `exec` family of system calls makes used of the `PATH` environment variable which gives a number of directories separated by '`:`'. For example, suppose `PATH` is "`/bin:/usr/bin`" then `execvp()` will try running the executables in the order "`/bin/`*prog*" then "`/usr/bin/prog`", and executing the first one which succeeds. See also `man execvp`, you can see that some exec calls use *path* (pathname) and *file* (uses `PATH` with *file*).

For `lab4ex4`, you must use `execvp()` or `execve()` to execute the command.[4] The errors and error messages from Exercise 3 apply to Exercise 4. `lab4ex4` will exit on the Exercise 3 errors even if it has child processes running. You can assume that the length of input never exceeds 128 characters (but we will not test this).[5] The maximum number of `%` is 8.

[3] This is different from the job limit in Tutorial 2. In Tutorial 2, the OS waited for all the jobs in the job limit to complete before reading new jobs. Here, once a job finishes, the next one can be started.

[4] In Linux, `execve()` is in Section 2 of the manual, so can be considered a system call while the rest of `exec` is in Section 3, so are library functions which are implemented using `execve()`. For simpler code, you can use `execvp()`.

[5] Recall from Lab 1, how to deal with unbounded lines.

The motivation for Exercise 4 is not only learn how to program with processes in Unix but also to create a semi-useful tool. With more options, it can be a reasonable tool which one might use. An existing tool with similarities is `xargs` (`man xargs`). Although the description of the lab has many features, the code can be written quite compactly so it will not be too much programming effort. However, there is the problem of dealing with bugs in C such as "`segmentation faults`" when there is a pointer bug.

Hints:
Rather than doing a lot of string manipulation, it is possible to reuse many of the original strings in the process! Dynamic memory allocation is not needed given the assumptions, however, one could also use `malloc()` for a bit more generality. In order to convert numbers from the string to the integer, you can use `atoi()`. The `wait` family of system calls will be useful. To read a line from `stdin` into a buffer: (assumes `buf` to be an array of `char`)

```
fgets(buf, sizeof(buf), stdin);
```

The function `strtok()` may be useful for breaking up a line into words. Look at `setenv()` for environment variables.

Submission

You should submit the following files using our **filename conventions** for **all** files (including the zip). Remember to include your information as a comment at the top of the code. Submit `lab4ex3.c` and `lab4ex4.c` (with the correct naming convention) in a single zip file to the appropriate workbin.

Submissions for Lab 1 has been treated as a special case and some leniency has been allowed. For all labs, other than Lab 1, compilation errors are treated as the program being **incorrect**. Please make sure all code compiles–sometimes students modify the code with comments after they have finished testing which may introduce new errors. So you should double check that everything compiles. Any runtime error means the test case used is failed, i.e. "`segmentation fault`" is treated as a manifestation of a bug for that test case.