Indian Institute of Information Technology Guwahati
Operating Systems Lab (CS232)
CG1 / Assignment 1

Dated 13th January, 2019 Total marks: 30

- **Submission instructions:** Submit your assignment as roll_no.extension to the address http://172.16.1.102:8080/ strictly within deadline. Only 'c', 'cpp' and 'sh' files will be accepted for the submission.
- **Deadline: 0800 hrs, 20th January, 2019**
- **This assignment has two components. The first one needs to be completed in the today's lab itself. The Takeaway component, part II will be evaluated in the next lab session.**

Part I:

1. The /proc file system is a mechanism provided by Linux for the kernel to report information about the system and processes to users. Know more details about the /proc file system by referring the proc man page. Understand the system-wide proc files such as meminfo and cpuinfo, and per-process files such as status, stat, limits, maps and so on. Now answer the following questions using the /proc filesystem.

   (a) Run command more /proc/cpuinfo and explain the following terms: processor and cores. [Hint: Use lscpu to verify your definitions.]
   (b) How many cores does your machine have?
   (c) How many processors does your machine have?
   (d) What is the frequency of each processor?
   (e) How much physical memory does your system have ?
   (f) How much of this memory is free ?
   (g) What is the total number of number of forks since the boot in the system ?
   (h) How many context switches has the system performed since bootup ?

2. Write a shell script for counting the number of lines in a given text file. Your program should additionally report the number of bytes and the number of words as well. All these output to be written into another text file.

   Example shell script for adding two numbers as arguments:

```
#!/bin/bash
for arg in "$@"
do
index=$(echo $arg | cut -f1 -d=)
val=$(echo $arg | cut -f2 -d=)
case $index in
X) x=$val;;
Y) y=$val;;
*)
esac
done
((result=x+y))
```

```
echo "X+Y=$result"
```

Run `$ ./test.sh X=44 Y=100`

3. In this question, we will understand how the Bash shell runs user commands by spawning new child processes to execute the various commands.
   (a) Compile the program cpu-print.c given to you and execute it in the bash as follows.
   $ gcc cpu-print.c -o cpu-print
   $ ./cpu-print
   This program runs in an infinite loop printing output to the screen. Now, open another terminal and use the ps command with suitable options to find out the pid of the process spawned by the shell to run the cpu-print executable. You may want to explore the ps command thoroughly to understand the various output fields it shows.

   (b) Find the PID of the parent of the cpu-print process, i.e., the shell process. Next, find the PIDs of all the ancestors, going back at least 5 generations (or until you reach the init process).

   (c) We will now understand how the shell performs output redirection. Run the following command. ./cpu-print > /tmp/tmp.txt & Look at the proc file system information of the newly spawned process. Pay particular attention to where its file descriptors 0, 1, and 2 (standard input, output, and error) and pointing to. Using this information, can you describe how I/O redirection is being implemented by the shell?

   (d) Next, we will understand how the shell implements pipes. Run the following command. ./cpu-print | grep hello & Once again, identify the newly spawned processes, and find out where their standard input/output/error file descriptors are pointing to. Use this information to explain how pipes are implemented by the shell.

   (e) Consider the following commands that you can type in the bash shell: cd, ls, history, ps. Which of these are system programs that are simply executed by the bash shell, and which are implemented by the bash code itself?

Part II:

The OS command interpreter is the program that people interact with in order to launch and control programs. On UNIX systems, the command interpreter is often called **shell**: a user-level program that gives people a command-line interface to launching, suspending, and killing other programs. `sh`, `ksh`, `csh`, `tcsh`, `bash`, ... are all examples of UNIX shells. You use a shell like this every time you log into a Linux machine at a URCS computer lab and bring up a terminal. It might be useful to look at the manual pages of these shells, for example, type "`man bash`".
The most rudimentary shell is structured as the following loop:

1. The shell should run continuously, and display a prompt (something similar to '$') when waiting for input. [Include the full path of the current directory in the prompt followed by your chosen prompt symbol.]
2. The shell should read a line from input one at a time.

3. Parse the line into the program name and an array of parameters;
4. Use the `fork()` system call to spawn a new child process;
    a. The child process then uses the `exec()` system call (or one of its variants) to launch the specified program;
    b. The parent process (the shell) uses the `wait()` system call (or one of its variants) to wait for the child to terminate;
5. Once the child (the launched program) finishes, the shell repeats the loop by jumping to 1.

Although most commands people type on the shell prompt are the names of other UNIX programs (such as `ps` or `cat`), shells also recognize some special commands (called internal commands) that are not program names. For example, the `exit` command terminates the shell, and the `cd` command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking a child process to handle them.

Requirement details:

Your job is to implement a very primitive shell that knows how to launch new programs in the foreground and the background. If a command is followed by &, the command must be executed in the background. That is, the shell must start the execution of the command, and return to prompt the user for the next input, without waiting for the previous command to complete. The output of the command can get printed to the shell as and when it appears. It should also recognize a few internal commands. More specifically, it should support the following features.

- It should recognize the internal commands: `exit`, `ls`, `pwd` and `cd`.

  [Hint: `exit` should use the `exit()` system call to terminate the shell. `cd` uses the `chdir()` system call to change to a new directory. You can use the standard C library functions chdir, getcwd, mkdir, rmdir, readdir, stat etc. to implement the calls. Note that these are built-in commands for your shell and hence no new process will be created to execute them. They neither read from the standard input stream nor write to the standard output stream. So your program should ignore file redirection with those built-in commands. Your program should still recognize invalid commands, however.]

- If the command line does not indicate any internal commands, it should be in the following form:
  <program name> <arg1> <arg2> .... <argN>
  Your shell should invoke the program, passing it the list of arguments in the command line. The shell must wait until the started program completes.

To allow users to pass arguments you need to parse the input line into words separated by whitespace (spaces and '\t' tab characters). You might try to use `strtok_r()` for parsing (check the manual page of `strtok_r()` and Google it for examples of using it). In case you wonder, `strtok_r()` is a user-level utility, not a system call. This means this function is fulfilled without the help of the operating system kernel.
The shell runs programs using two core system calls: `fork()` and `execvp()`. Read the manual pages to see how to use them. In short, `fork()` creates an exact copy of the currently running process, and is used by the shell to spawn a new process. The `execvp()` call is used to overlay the currently running program with a new program, which is how the shell turns a forked process into the program it wants to run. In addition, the shell must wait until the previously started program. This is done with the `wait()`

system call or one of its variants (such as `waitpid()`). All these system calls can fail due to unforeseen reasons (see their manual pages for details). You should check their return status and report errors if they occur.
No input the user gives should cause the shell to exit (except when the user types `exit` or `Ctrl+D`). This means your shell should handle errors gracefully, no matter where they occur. Even if an error occurs in the middle of a long pipeline, it should be reported accurately and your shell should recover gracefully. You may need to use `signal()` and `kill()` system calls. In addition, your shell should not generate leaking open file descriptors.
Hint: you can monitor the current open file descriptors of the shell process through the `/proc` file system.
Your shell must run in one of two modes: interactive or batch. If no command-line argument is provided, your shell should interactively accept inputs from the user and execute them. If a batch file of commands is provided as command-line input to your program, then your shell must execute all commands in the batch file one after the other.

Additional requirement of Pipes:
Your shell needs to support pipes. Pipes allow the `stdin`s and `stdout`s of a list of programs to be concatenated in a chain. More specifically, the first program's `stdout` is directed to the `stdin` of the second program; the second program's `stdout` is directed to the `stdin` of the third program; and so on so forth. Multiple piped programs in a command line are separated with the token "|". A command line will therefore have the following form:

<program1> <arglist1> | <program2> <arglist2> | ... | <programN> <arglistN> [&]

Try an example like this: pick a text file with more than 10 lines (assume it is called `textfile`) and then type

```
cat textfile | gzip -c | gunzip -c | tail -n 10
```

in a regular shell. Pause a bit to think what it really does. Note that multiple processes need to be launched for piped commands and all waited on in a foreground execution. The `pipe()` and `dup2()` system calls will be useful.

Other useful links:

- GNU C Library: http://www.delorie.com/gnu/docs/glibc/libc_toc.html
- File Descriptors: http://www.delorie.com/gnu/docs/glibc/libc_169.html
- Duplicating File Descriptors: http://www.delorie.com/gnu/docs/glibc/libc_257.html
- Pipes: http://www.delorie.com/gnu/docs/glibc/libc_296.html
- I/O Redirection: http://www.tldp.org/LDP/abs/html/io- redirection.html
- GNU C Library-Signal handling: http://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html