CS333 Autumn 2016

Lab Quiz 1

Goal

The goal of this assignment is to test the concepts of processes, threads, and signal handling from labs 1, 2 and 3. This lab has two parts: you will first write a simple multi-threaded program, followed by a simple command shell for Linux.

You are provided several helper files along with this lab in a tar gzipped file. You may run `tar -zxvf labquiz1_files.tgz` to extract the following helper files: `integers.c`, `n.txt`, `make-tokens.c`.

Part A: Multi-threaded data processing

You must create a multi-threaded program `multi-avg.c` to find the average of integers spread across multiple files. You must compute the average as the sum of all the integers present in all files divided by total number of integers in all files. Your program will take the following as command line arguments (you can hard code them if you cannot get command line arguments to work):

1. Number of threads N you must spawn.
2. A list of N filenames.

Each file will have one integer per line: a test file `n.txt` with the format we will be using has been provided for your reference. Your code must spawn N threads, open one file in each thread, and process all files in parallel. After processing all files, you must join all threads, and print out the average of all numbers across all threads. We do not expect you to use any synchronization primitives in this exercise: you can maintain separate counters for each thread, in order to avoid updating shared counters. You are given a sample file `integers.c` that reads and prints out the integers in a file, to help you navigate the file parsing. You may extend this file to add the logic for multi-threading and computing averages.

Below is sample output from the execution of this program.

```
$./multi-avg 3 a.txt b.txt c.txt
Average: 10.435
```

Once you write your program and check its basic correctness, proceed to the following exercise and record your observations in your report. Generate 8 files with 10,000 random integers each. Calculate the time t_p taken to calculate the average of these integers in parallel using 8 parallel threads. That is, you must measure the time taken by `multi-avg` from the time it starts the threads to stops them after completion, for $N = 8$. Next, calculate the time t_s taken to compute the average of the integers in these

8 files in a single threaded program (you may have to write another program for this purpose). Measure the time in a similar fashion across both cases, and ensure that you read from disk and not buffer cache.

Now, compute the speedup ratio t_s/t_p achieved by parallelization, and justify this number. For example, you may justify this observation using information about the number of CPU cores on the machine, or any such thing.

Helpful tips

- Use the file `integers.c` to help you get started with file parsing.
- To clear buffer cache do "sudo drop_cache"
- To generate random integers in a file using bash: "for i in $\{1..100\}$; do echo $RANDOM; done"
- Helpful man pages—man 2 open, man read, man close, man pthread_create, man pthread_join.
- Add -pthread tag to compile C or C++ programs using pthreads.

Part B: A simple Linux shell

You will write `simple-shell.c` that implements a simple command shell for Linux. You are provided sample code `make-tokens.c` that takes a string input from the user and tokenizes it; you may reuse this as part of your shell. Use any creative prompt message that you want. Below are the the commands you need to implement in the shell, and the expected behavior of the shell for that command. **Note:** You must use the `fork` and `exec` system calls to implement the shell. You must not use the `system` function provided by the C-library, which simply calls an existing Linux shell. Also, you must execute Linux system programs wherever possible, instead of re-implementing functionality. You may use C++ for the string-handling parts of the assignment.

- `cd` *`directoryname`* must cause the shell process to change its working directory. This command should take one and only one argument; an incorrect number of arguments (e.g., commands such as `cd`, `cd dir1 dir2` etc.) should print an error in the shell. `cd` should also return an error if the change directory command cannot be executed (e.g., because the directory does not exist). *For this, and all commands below, incorrect number of arguments or incorrect command format should print an error in the shell. After such errors are printed by the shell, the shell should not crash. It must simply move on and prompt the user for the next command.*

- All *simple* built-in commands of Linux e.g., `ls`, `cat`, `echo`) should be executed, as they would be in a regular shell. There is no need to support I/O redirection or pipes or background execution or any such features in the simple Linux commands. All such commands should execute in the foreground, and the shell should wait for the command to finish before prompting the user for the next command. Any errors returned during the execution of these commands must be displayed in the shell.

- Any list of simple Linux commands separated by ";;" (e.g., `sleep 100 ;; echo hi ;; ls -l`) should all be executed in the **foreground** and **sequentially** one after the other. The shell should start the execution of the first command, and proceed to the next one only after the previous command completes (successfully or unsuccessfully). The shell should prompt the user for the next input only after all the commands have finished execution. An error in one of the commands should simply cause the shell to move on to the next command in the sequence. An incorrect command format must print an error and prompt for the next command.

- Any list of simple Linux commands separated by "&&" (e.g., `sleep 100 && echo hi && ls -l`) should all be executed in the **background** in **parallel**. The shell should start the execution of all the commands in parallel by creating child processes for each, and come back to prompt the user for the next command without waiting for all the commands to finish. For each command that completes in the background at a future time, the shell should print a message about the background process being completed. An incorrect command format must print an error and prompt for the next command.

- Any simple Linux command followed by the output redirector > (e.g., `echo hi > hi.txt`) should cause the output of the command to be redirected to the specified output file. The command should execute in the foreground. An incorrect command format must print an error and prompt for the next command.

- `exit` should cause the shell to terminate, along with all child processes it has spawned.

- For any other command that doesn't match what is specified above, the shell should print an error and move on to prompt the user for the next command.

The handling of the SIGINT (Ctrl+C) signal by the shell should be as follows. While the shell is running a command in the foreground, typing Ctrl+C should cause the foreground child process to terminate, but not the shell itself. When the user is executing a set of commands in sequence using ";;", Ctrl+C should terminate the execution of the currently executing command, and all future commands in that sequence, and must cause the shell to return to the user prompt. Hitting Ctrl+C should not terminate any background processes started by using &&. Typing the `exit` command should cause all background child processes to receive the SIGINT signal and terminate; the shell itself must terminate after cleaning up all state and dynamic memory.

The shell must also carefully reap all its children that have terminated. For commands that must run in the foreground, the shell must wait for and reap its terminated foreground child processes before it prompts the user for the next input. For commands that create background child processes, the shell must periodically check and reap any terminated background processes, while running other commands. When the shell reaps a terminated background process at a future time, it must print a message to let the user know that a background process has finished. By carefully reaping all children (foreground and background), the shell must ensure that it does not leave behind any zombies or orphans when it exits.

You must implement all the commands above in your shell. Test your shell using several test cases, and record them all in your report.

Helpful tips

- You can start with `make-tokens.c` to write your shell.
- You can make any assumptions you need on the maximum number of tokens in a command, the maximum number of commands provided for serial or parallel execution, and such things.
- Helpful man pages—man fork, man exec, man signal, man wait, man 7 signal, man setpgid, man 2 kill, man getpgid, man getpid, man gettimeofday, man chdir, man 2 open, man read, man 2 write, man close, man dup, man dup2.

Submission and Grading

You must solve this lab individually, in the specified lab slot. No extensions will be allowed. Even if you haven't finished fully, please submit whatever you have when the time is up. Please create a tar gzip with your roll number as the filename and submit on Moodle. Your tarball must contain the following files.

- Your code files `simple-shell.c, multi-avg.c`. Comment and indent your code so that it is readable.
- `report.pdf` describing the execution time results in part A and shell test cases in part B.

We will evaluate your submission by reading through your code, executing it over several test cases, and by reading over your report.