COMP3430 **Assignment 1** Due: February 5th, 2018 @11:59PM

Assignment Objectives:

- *To learn something more about the processes needed to execute your code under Linux.*
- *To gain a better understanding of Unix/Linux (i.e. "*nix") processes and basic concurrent program execution.*
- *To gain experience calling more *nix system calls (and reading the man pages for them)*
- *To write your first, simple, p-threads application in C under Linux.*

Assignment Instructions:

- *You must hand in your assignment electronically via UMLearn using the "Assignments" functionality under the "Assessments" drop-down menu. A folder for Assignment 1 has been created there. You can hand in multiple files if so desired but we would ask that you be kind to the markers and hand in everything in one batch at the end if at all possible. Recall that you must agree to the online honesty document before the dropbox will become visible. You can read back in the announcements on UMLearn for details on how to do this though you should have already done this to hand in you first lab.*
- *In general, for written questions PDF (.pdf), Word (.docx), PowerPoint (.pptx), Rich Text Format (.rtf) and Text (.txt) format files are acceptable. Handwritten then scanned PDF files are **not** acceptable. For any programming questions, please hand in all **source** files and any requested results as well as a makefile to generate the executable file(s) for the target platform. You **must** also include a text file named 'README' for each programming question that briefly describes how to use your program.*
- *Your programs must run on the Linux lab machines, but you can try to develop your code on other platforms if you like though this is **not** recommended. In some cases, this may simply not be possible since your running programs will be making Linux system calls. Should you choose to use different platforms for development, you **alone** will be responsible for dealing with any differences in the compilers, tools, etc. that you may encounter. In such a case, be sure to leave plenty of time to port your code to Linux. Normally this takes less than 30 minutes but sometimes it may take several hours if your code is still a little buggy. (For example, the C compiler on some systems sets all uninitialized memory to zeroes. Hence, bad pointers are treated as NULLs and thus, sometimes incorrect code works there but not with other compilers and operating systems such as gcc on Linux.)*
- *Note that assignments are to be done **entirely independently** unless otherwise explicitly stated and that inclusion of materials from online sites is entirely forbidden.*

Assignment Questions:

[5] 1. Do a **remote** login using ssh (e.g. `slogin aviary.cs.umanitoba.ca`) to one of the Linux Lab machines (`aviary` will pick one lab machine more or less randomly for you). Type the command `ps xf` to find out what processes *you* are running. Determine and record which process is running your command shell and what the program name for the shell is (i.e. what shell you are running: `sh, bash, csh, tcsh,` or …). As you know from the lab, Linux provides various information about all processes in file in the `/proc` directory. You are to use this information to determine the process-ids of the parent, grandparent, great-grandparent, etc. of your shell all the way back to process-id 1 as well as what program each process is running. This information can be easily presented as a sequence of <process-id,

program-name> pairs. To find the necessary information, and assuming that the process-id of your shell happens to be `27814`, then you should begin by exploring the contents of the **file** named `/proc/27814/status`. Compare the information you ultimately find to the diagram given in the concurrency section of the course notes. Use the `man` command to learn a little about each program being run by one of your shell's ancestor processes and also bear in mind how you connected to the machine you are using. You will notice that the process with pid 1 says its parent has pid 0 but no entry exists in /proc for a process with pid=0. Why do you suppose there is no entry for process 0?

[3] 2. Explain why it does not make sense to make an `exec` call in multi-**threaded** code. What do you think might happen if two threads (running in the same address space) both tried to execute an `exec()`? Explain!

[2] 3. In the first lab, you probably used the string tokenizer (`strtok`) library routine to separate the arguments to each program entered as input to your shell. The routine `strtok` was designed to be used with **non**-concurrent code. There is another version called `strtok_r` that can be safely used with concurrent programs. Explore this version (using the `man` command or checking the web) and briefly explain why this version is necessary and how it circumvents the problem that would occur if `strtok` were to be used in a concurrent (e.g. multithreaded) program.

[10] 4. Command shells commonly provide a facility whereby shell "variables" may be defined and then used in commands provided to the shell. Such variable names normally begin with a special symbol (e.g. '`$`') and are then followed by some letters (e.g. '`$HOMEDIR`'). A mechanism must be provided to set such variables and then whenever the variable occurs, it is replaced by the string it was set to. For example, a user might type something like '`set $HOMEDIR=/home/cs/staff/pgraham`' at the shell prompt. Later, if the same user typed '`ls $HOMEDIR`', the shell would execute the command '`ls /home/cs/staff/pgraham`'. You are to extend the mini-shell you created for the last question in the lab (or the one from the sample solution) to support such variables and their use based on the syntax described in the examples above. There is lots of C code on the web which you are free to consult. To avoid possible issues of academic misconduct, however, do **not** cut and paste **any** of the code you may find online into your program. Always write your own code!

[5] 5. You are to extend the mini-shell you created for question 4 to support the pre-loading and execution of a sequence of 'set' commands from a file named '`.shell_init`' in the user's home directory. This will require you to open and read from a file in C. There is, naturally, also lots of example code on how to do this online but, again, do **not** cut and paste **any** of the code you may find online into your program.

[10] 6. You are to further extend your mini-shell (from either question 4 or question 5) to support use of output redirection and a **single** pipe. The ideas behind how this needs to be done are provided in the textbook on pages 42-43, starting with the tip "GETTING IT RIGHT (LAMPSON'S LAW)". This will require a bit of thought, careful coding and some use of the online `man` pages and/or the Internet to understand the standard I/O API and how to use the `pipe()` system call.

[10] 7. You are to write a simple C program running on Linux to find the minimum element in a vector using **multiple, concurrent** p-threads. The idea is to have each thread find the minimum value in an assigned part of the vector and then have the "parent" (main routine) find the minimum of the minimums found by each thread. (Thus, for very large vectors, the minimum finding should be sped up – a good use of threads.) Of course, this is just a variant of the common "divide-and-conquer" strategy with threads doing each part **concurrently**.

Your program will be passed two command line parameters: `NumThreads` and `NumElts`, both integers (that can be accessed using the `argc` and `argv` arguments to the main routine). Your program will begin by **dynamically** allocating (using `malloc`) an in-memory array '`ar`' large enough to contain the `NumElts` integers read from a file named '`numbers.txt`'. You may assume that `NumThreads` will divide the number of elements in the array (`NumElts`) evenly. After reading the integers (so the first entry in the array will contain the integer on the first line of the file, the second entry the value from the second line and so on), your program must create `NumThreads` threads each running a function you write called `FindMin`. Your program should pass **two** things to each thread running `FindMin`: the number of elements it is to find the minimum of (`threadNumElts`) and the starting point in the array where it is to begin its work (`start`). The routine `FindMin` should thus find the minimum of the elements in the array from element `start` up to and including element `start+threadNumElts-1`. Each thread should "return" the minimum it found to its creator. The parent/creator will have to wait until each "child" thread has finished and then find the minimum of minimums and print it out. How you handle the returning of values is up to you. (Think about the possibilities remembering that threads to **not** share stacks!) Of course, there is also lots of p-threads example code available on the web that you can look at. As before, to avoid possible issues of academic misconduct, do **not** cut and paste **any** of the code you may find into your program. Doing it yourself is how you learn and assignment material is examinable.

Total: 45 marks

Want a Challenge?

No marks for this but an interesting opportunity exists to learn some more about what a command shell needs to do as well as to learn something about what's involved "under the hood" in dealing with that pesky command line interface from a programmer's point of view. Solving the following problem will require a fair bit of "Googling". Try to further extend your mini-shell to keep track of the last 100 commands entered by the user and allow the user to use the up and down arrow keys to scroll through the history of commands and re-execute the one shown by hitting return. If you are unsure of what this means, try using the up and down arrow keys in whatever "real" command shell you are using and you will see what is needed. You do **not** need to be able to edit the commands that have been recalled (even though your shell will support this). No marks so, obviously, no handin is required either.