PROGRAMMING ASSIGNMENT #1

*Reading virtual /proc files*

CSCI 480 100 points Spring 2024

Check Blackboard for due date. Due by 11:59 PM.
Late penalty as in syllabus.

This assignment will be programmed in C and/or C++ on turing/hopper. Get yourself familiarize with the Linux system and the related environment and commands on turing.cs/hopper.cs. Read the "useful links" provided in course website to refresh your skills if necessary.

**Background:**

The Linux `/proc` file system is a "pseudo" file system that exists only in kernel memory and is used primarily for querying various kernel and per-process statistics. Although the `/proc` file system is virtual, you can open and read the virtual files the same way as any other file. So you can start this assignment without the instructor discussing the `/proc` file system.

You will answer questions about *turing* by writing a program that reads some virtual files in the `/proc/sys/kernel/` directory, and the four virtual files: `/proc/cpuinfo`, `/proc/uptime`, `/proc/stat`, and `/proc/swaps`.

a) Review the C++ `string` class and use it in this program. Use member functions of the string class to process strings. For example, to search for a substring in a string, you can use the `string` member function `find()`.

b) In `/proc/cpuinfo`, the majority of the input fields repeat for each processor. Processor numbers need not be contiguous, so you need to count the number of processors. On *turing*, each processor block in the `/proc/cpuinfo` corresponds to a CPU core. A physical chip can have multiple cores. A host such as *turing* can have several physical multi-core chips.

c) For `/proc/cpuinfo`, you need to read the file, save the fields you need, and then format your output.

You do not know which order the blocks will come in, and you do not assume the order of the keywords within the block except the first line is "processor: #". For example, do not assume the keyword "Physical id" is the 10th line of the file. Instead, you need to read line by line and check if the keyword in a line matches the keyword you are looking for. To get the total number of physical multi-core chips, you must keep track of the content of "*physical id"* fields (instead of only reading the last block in the file).

d) Look at the "*cpu cores*" field in `/proc/cpuinfo` to get the number of cores for each physical chip.

e) The file `/proc/stat` records information about the system since it was started, including:

cpu

> Total CPU time spent in user mode, low-priority user mode (nice mode), system mode, idle, and others. Here the time is recorded as the number of timer interrupts that have occurred. The time unit is the so called *jiffy*. In Linux, the value of the tick rate HZ determines the frequency of the timer interrupt, and its value varies by machine type and architecture. For example, if the value of HZ is 100, a timer interrupt occurs 100 times

per second, or every 10 milliseconds, so jiffy value is 10 ms. The value of HZ on *turing* is 100.

cpu*N*

Times for CPU *N*.

f) `/proc/uptime` provides two numbers. Use the manual page for *proc* to find out what they are.

g) The file `/proc/swaps` provides information on swap devices. The units of size and usage are kilobytes.

h) Command "man proc" is the manual page and can give you more details of the /proc file system.

**Program Output Format**

You need to express each output in a numbered *complete sentence*, e.g.:

```
A: Questions about turing's OS:
1. The type of turing's OS is __ .
…

B: Questions about turing's processor:
1. Turing has __ processors.
2. Turing has __ physical multi-core chips. (if there is only one: 1 physical
multi-core chip.)
…
```

Your program output is expected to answer the following questions. You can decide the exact sentence as long as you format the answers in a human readable way.

A. Questions about turing's OS. (Read the following files under the directory `/proc/sys/kernel/` and format the output in a readable way.)
1. ostype
2. hostname
3. osrelease
4. version

B: Questions about turing's processors:
1. How many processors does turing have?
2. How many physical multi-core chips does turing have?
3. How long has turing been up in seconds?
4. Express that time in days, hours, minutes and seconds
(e.g., 7 days, 1 hour, 2 minutes, and 4.3 seconds)

C. For *processor 0*, answer these questions:
1. Who is the vendor?
2. What is the model name?
3. What is its physical address size?
4. What is its virtual address size?

D. For *processor 5*, answer these questions:
1. How long has it spent time in user mode (just the 1st column, not nice mode) in seconds?
2. How long has it spent time in system mode in seconds?
3. How long has it been idle in seconds?
4. How long has it been idle in days, hours, minutes and seconds?

E. What is the size of turing's swap device in *MB*?

**Additional Notes**

The terms "processor", "CPU core", "chip" can be confusing sometimes, based on the context.

In the context of /proc/cpuinfo, "Each processor block corresponds to a CPU core", NOT a physical processor chip. A computer can have several physical processor chips, each chip can have multiple cores.

Note that if it is a *hyper-threading* core, then a processor block in the *cpuinfo* would represent a *virtual* core. But *turing* does not have this issue, so you do not need to worry about it for this assignment.

Because there is no virtual core, then the number of processor blocks in /proc/cpuinfo on turing is the same as the number of CPU cores, which is also equivalent to "the number of physical chips" multiplying "the number of CPU cores per chip".

In other words, to answer B1, you have two ways to do it: 1) You count directly how many processor blocks are there in the *cpuinfo* (preferred); or 2) you can do the multiplication: (number of physical chips) x (number of CPU cores per chip). For turing, these two ways will give you the same result. But if a system has hyper-threading, then the two results may be different.

The Linux command "lscpu" can give you some additional information. In fact, lscpu is also based on parsing the information from /proc/cpuinfo, same as what you are doing for the assignment.)

***Coding style and documentation standards:***
Programs must be consistently indented and commented so that a reasonable person can understand them. Use of consistent descriptive variable names is required.

At a minimum, follow the following rules that you have used in earlier programming courses, including the use of a header box at the top of the program:

http://www.cs.niu.edu/~byrnes/csci240/240doc.htm

Here are some of the important rules:

This block of comments should be at the top of your source code, before anything else. For example:

```
/*************************************************************
CSCI XXX - Assignment X - Semester (Fall/Spring) Year

Programmer: Your name goes here
```

```
Section:    Your section number goes here
TA:         Your Teaching Assistant's name goes here
Date Due:   The assignment due date goes here

Purpose:    A brief (2-4 sentences) description of what the program does
            goes here. For example:

            This program accepts a single number from the keyboard
            representing a temperature in Fahrenheit. It then converts
            it to Centigrade and displays it.
*****************************************************************/
```

- Variable Names: Most variable names in your program should describe the quantity or item that they represent. If a variable is to hold the number of students in a class, don't name it "n" or even "num" or "count"; do name it "studentCount" or "numberOfStudents". When declaring variables, group and format them in a neat and logical manner.
- Indentation: You must **consistently indent** the body of loops and the alternate blocks of a decision structure.
- Line, Section and Function Documentation: A moderate amount of documentation in the main body of your program may be advisable, but if you name your variables and functions with meaningful names, you should not need much. It is advisable to have a documentation box for each important function that explains its purpose. Ask yourself – *will you still understand your code six months later?*

***Administration:***

Submit the source on Blackboard. We will compile and run them on `turing`. *No credit if your program does not compile on `turing`.*

The compressed file contains your source code and a Makefile. It needs to be named as "your-zid_project1.tar" and must be created following the procedure described below:

1. Put all your source code files (NO OBJECT or EXECUTABLE FILES) and your Makefile in a directory called "your-zid_project1_dir". Example: z1234567_project1_dir. **Note**: 'z' must be in lower case.

   In your Makefile, you need to make sure your compilation produces the executable file called "your-zid_project1". For a student with z1234567 as her zid, the executable would be *z1234567_project1*.

   In addition, in your Makefile, you need to include the pseudo-target `clean` to remove object code and executable files. This is to help the TA to clean up their directory since their size quota is limited. Example:

   ```
   clean:
       -rm *.o z1234567_project1
   ```

2. In the parent directory of your-zid_project1_dir, compress this whole subdirectory by the following command:

   ```
   tar  -cvvf  your-zid_project1.tar  your-zid_project1_dir
   ```

Example:

```
tar -cvvf z1234567_project1.tar      z1234567_project1_dir
```

"your-zid_project1.tar" is now the compressed file containing the whole subdirectory of your files. You can then transfer (e.g. using an ftp client) the tar file from turing (or hopper) to a computer on which you can open a web browser for your final submission to the Blackboard system.

***Grading:***

When the program is graded, a script runs a sequence of the following commands:

*tar –xvf z1234567_project1.tar*
*cd z1234567_project1_dir*
*make*
*./z1234567_project1*

These procedures should yield a working program. You need to verify by yourself that your compressed tar file can be opened properly. There will be a penalty if programs are incorrectly named or otherwise do not follow directions (see Syllabus for the details of penalty).

The grading of this assignment will be based on output correctness (40%), programming (40%), coding style and documentation (20%), under the condition that it complies and runs on `turing`.

For full credit, your program must follow the specs and documentation standards above. In other words, it is possible that you will get deduction even when your program is generating correct results if your coding style is bad. Here are some examples of bad style:

1. Lack of necessary documentation/comments. Or have incorrect comments;
2. Hard to understand variable names or function names;
3. Hard-coded magic numbers without obvious meaning or explanation;
4. Inconsistent style of indentation or curly braces to the point of reducing understandability.