OS 2015 – EX1

Before you start, don't forget to read the course guidelines!
Pay attention to our Cheating Zero Tolerance Policy discussed there.

Task 1 - Using strace to understand what a program is doing (15 points)

The purpose of this task is to practice using the "strace" command. This command is a debugging utility for Linux. It can assist you in debugging your programs in advanced exercises in this course, by enabling you to understand the reasons for system-calls' failure rapidly without diving into code lines. More details about this command will be discussed in the exercise sessions. We also suggest you read about the command via the shell ("strace –help" or "man strace").

In this task, you should follow the strace of a program (written by us) in order to understand what the programs does. You can assume that the program does only what you can see by strace (meaning, it doesn't have internal code in addition to what you can see).
To run the program, do the following:

- Download WhatIDo into an empty folder in your login in the CS computers.
- Run the program using strace.
- Follow strace output. Tip: many lines in the beginning of the output are caused by the initialization of the program. Actually, the first "interesting" lines start after the "**munmap"** system call that comes immediately after the **last "mprotect"** system call.

Your task is to supply a brief description of what the program does in the README file, under "Task 1" category. In addition, explain how you concluded that by describing what each relevant system call does (including its parameters and returned value). Tip: google may be useful to understand that.

Task 2 – Open Questions (15 points)

Please answer the following questions briefly in the README file, under "Task 2" category.

1. (7.5 points) While interrupt handling saves and restores the current state of the running process, it still may have side-effects on the running process. This may result in unwanted behavior of the program. Please specify two different side-effects.
   (Hint – what happens when two programs try to write to the same file, and there was an interrupt in the middle of one of them? Moreover, how do interrupts influence a time-based program?).
2. (7.5 points) A possible solution to the above problems is to block all interrupts when a certain program is running. Is this a good idea? What are the implications?

Task 3 – The Cost of Trap (70 points)

Background

Operating system code runs in "kernel mode", in which the full range of instructions of the architecture is allowed by the CPU. This includes many privileged instructions that are used to control the hardware, and must not be used by normal user code. Therefore, changing execution mode from "user mode" to "kernel mode" so that the operating system can run is not trivial. This operation is called a "trap", and it occurs at the beginning of each system call. In this exercise we will measure the overhead involved in executing a trap.

Assignment

Your assignment is to build a library called **osm** that provides functions to measure the time it takes to perform three different operations: a simple instruction (such as addition or bitwise and), an empty function call with no arguments, and a trap. It also provides a function which returns a `struct` containing all the data we are interested in (see below). The header file for the library is osm.h - it includes the `struct` definition, and you should implement all its functions.

To measure the time it takes to perform an operation, you should use the function *gettimeofday* (run `man gettimeofday` for how to use it). Since any single operation takes a very short time, you will need to measure each operation over many iterations done in a loop, and calculate the average time the operation took. The number of iterations is fed as an argument to each function.

To measure the time it takes to perform a trap, we have provided you with an empty system call, called 'OSM_NULLSYSCALL', which traps into the operating system but does nothing. It is defined in the library header file. Be aware that this call works on CS labs computers (and "river", for connecting from outside), but it may not work on other versions of Unix-based 64-bit operating systems.

As mentioned, all functions require, as an argument, the number of iterations needed. It denotes the number of loop iterations to perform, and if the argument received isn't valid, the function should default to 50,000 iterations. To measure the time it takes to run a single instruction, it is advisable that you perform loop unrooling: in every iteration of the loop, run many instructions instead of just one instruction, and divide the time by the total number to get the average. Try to make the individual instructions independent from each other, so as to avoid delays in the processor's pipeline. Note that if you use loop unrolling, it is permissible to round UP the number of iterations to a multiple of the unrolling factor.

The main function you need to implement is:

`timeMeasurmentStructure measureTimes (unsigned int numOfIterations)`

`this function calls the others and returns all the results in a struct` containing all the data we are interested in, comparing the various times. The `struct` requires the following data:

1. *Machine name* - check `man 2 gethostname`.
2. *Number of iterations*.
3. *Instruction time* - in nano-seconds.
4. *Function time* - in nano-seconds.

5. *Trap time* - in nano-seconds.
6. *Function/instruction ratio* - the respective times divided.
7. *Trap/instruction ratio* - the respective times divided.

Notice that *gettimeofday* has a resolution of microseconds.

If any calculation had an error, the value of respective `struct` member should be -1. If there is an error in the machine name, a null string should be the value of the relevant `struct` member.

Guidelines

- **Do not change the** header file**. Your exercise should work with our version of** osm.h.
- You are required to add an explanation in the README file, under "Task 3" category, about how and why your library functions are built the way they are.
- The programming in this exercise is trivial. But you need to look at the results and think about how to make them reasonably accurate, and this may take time.
- **How can you tell your measurements are good enough?**
  It is hard to get exact measurements. Approximate measurements are good enough for this exercise.
  You should check that:
  - When you run the measurements several times on the same machine, you get similar results. Note that machine load can effect measurements.
  - The time you measure for a function call should be several times the time of a single instruction, and the time for a trap is significantly higher.
  - Ideally, the time for a simple operation should be one cycle, and other times should be a multiple of this.
- Make sure to check the exit status of all the system calls you use except the provided empty system call (OSM_NULLSYSCALL) that can't fail.
- Make your code readable (indentation, function names, etc.).

Submission

Submit a tar file on-line containing the following:

- A README file. The README should be structured according to the course guidelines and contains the required information described in **all of the tasks** (see the README template).
- The source files for your implementation of the library described in Task 3.
- A Makefile for Task 3. Running *make* with no arguments should generate the *libosm.a* library. You can use this Makefile as an example. Note that it is not a good idea to compile with optimizations in this exercise (can you see why?).