OS 2014 – EX1

Supervisor – Netanel Zakay

Before you start, don't forget to read the course guidelines!

Task 1 - Using strace to understand what a program is doing (20 points)

The goal of this task is to let you become familiar with "strace" command. "strace" is a linux command, which traces system calls and signals of a program. It is a strong tool to debug your programs in advanced exercises.
In this task, you should follow the strace of a program (written by us) in order to understand what a programs does. You can assume that the program does only what you can see by strace (meaning, it doesn't have internal logic in addition to what you can see by strace).
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 are part of the load of the program. Actually, the first "interesting" lines comes immediately after the following lines:

```
arch_prctl(ARCH_SET_FS, 0x7f486b183740) = 0
mprotect(0x7f486a78d000, 16384, PROT_READ) = 0
mprotect(0x7f486acac000, 4096, PROT_READ) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f486b182000
mprotect(0x7f486af9a000, 32768, PROT_READ) = 0
mprotect(0x7f486b1d9000, 4096, PROT_READ) = 0
munmap(0x7f486b187000, 325994)          = 0
```

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

Task 2 - The Costs of Trap and disk access (80 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.

Accessing the hard disk drive (HDD) is very slow relating to accessing local data in the memory. One main reason for that is the mechanical process that involved in each such access.

In this exercise, we will measure the overhead involved in executing a trap and accessing the disk.

Assignment

Your assignment is to build a library called **osm** that provides functions to measure the time it takes to perform four different operations:

1. A simple instruction (such as addition or bitwise and)
2. An empty function call with no arguments,
3. A trap
4. Accessing the disk.

The library also provides a function, which returns a struct containing all the data we are interested in. 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 must use the function *gettimeofday* (run *man gettimeofday* for more details). While this function is not perfect, it's the best for our purpose, and using other functions is not allowed. 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. Of course that you don't need to check the return status of this function (it is empty function, it always succeeds).

To measure the time it takes to access the disk, we strongly suggest you to use functions on files. In general, files are saved in the disk, and file's functions will access the disk. For example, several relevant function are fopen, fclose, fread, fprintf, fgets, fputs, fetc, fputc, fflush. One problem that you might have is that the OS accesses the disk in chunks. This has two main consequences:

1. When reading from file, the OS fetches more information than you requested. As a result, two successive calls to "*read*" may access the disk only once.
2. When writing to a file, the OS may save the data locally and write the results only later. Therefore, multiple "writes" may access the disk only once.
   [You probably already got familiar with this phenomenon, when you tried to print a message to the screen, your program collapsed after the print, but nothing was printed.

Google may help you perform to measure the disk time properly.

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 (0 is the only invalid number), the function should default to 1,000 iterations. To measure the time it takes to run a single instruction, it is advisable that you perform loop unrolling, that is: 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, 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.

There are several functions you need to implement. Four function that measure each operation described above, and a single function that measure all of them named measureTimes. This function doesn't return a primitive variable, but a struct containing all the data we are interested in, comparing the various times. The struct is described in details in *osm.h* file. Notice that *gettimeofday* has a resolution of microseconds, and the times in the structure are in nano-seconds.

If any calculation had an error, the value of respective struct member should be -1. If there is an error in the machine name, an empty string (also called 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 should write an explanation (in the README file, under "Task 2" 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.
- There is no single solution, and multiple ideas may work.
- The accurate of the cost of accessing the disk may not be as good as the others. We are aware of that, and it's fine.
- **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 the following.
  - 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, the time for a trap is significantly higher, and the time for accessing the disk is much higher too.
  - 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.
- 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 **both of the tasks**. In order to be compliant with the guidelines, please use the README template that we provided.
- The source files for your implementation of the library described in Task 2.
- Your Makefile for Task 2. 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?).

Late submission policy						
Submission time	3.3, 23:50	4.3, 11:50	6.3, 23:50	7.3, 23:50	8.3, 23:50	9.3, 23:50
Penalty	0	3	10	25	40	**Course failure**