OS 2019
Exercise1

Supervisor – Eytan Lifshitz

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

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

The goal of this assignment is to help you become familiar with the “strace” command. “strace” is a Linux command, which traces system calls and signals of a program. It is an important tool to debug your programs in advanced exercises.
In this assignment, you should follow the strace of a program in order to understand what it does. You can assume that the program does only what you can see by using strace.
To run the program, do the following:

- Download *WhatIDo* into an empty folder in your login on 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. The first “interesting” lines comes immediately after the following lines:

```
mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9ff9d96000
mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9ff9d94000
arch_prctl(ARCH_SET_FS, 0x7f9ff9d94740) = 0
mprotect(0x7f9ff932a000, 16384, PROT_READ) = 0
mprotect(0x7f9ff984d000, 4096, PROT_READ) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9ff9d93000
mprotect(0x7f9ff9bc0000, 40960, PROT_READ) = 0
mprotect(0x7f9ff9df3000, 4096, PROT_READ) = 0
munmap(0x7f9ff9d98000, 362622)        = 0
brk(NULL)                             = 0x1e48000
brk(0x1e7a000)                         = 0x1e7a000
```

Your assignment is to supply a brief description of what the program does in the README file, with the title “Assignment 1”.
Tip: google and the command “*man*” may be useful for this assignment.

Assignment 2 - The Costs of Trap and Virtualisation (80 points)

<u>Background</u>

Operating system code runs in "kernel mode", in which the full range of instructions of the architecture is allowed by the CPU. The “kernel mode” includes instructions that control the hardware and should not be used by normal code which runs in “user mode”. When code in “user mode” executes a system call, the CPU must change the execution mode. This operation is called a "trap".

In the first part of the exercise we’ve seen strace - an application which inserts itself between the user’s code and the operating system. In the Tirgul, we’ve seen valgrind, inserting itself between the user’s code and memory access. Taking this principle to the extreme, virtualisation runs code in a simulated computer with its own OS and hardware.

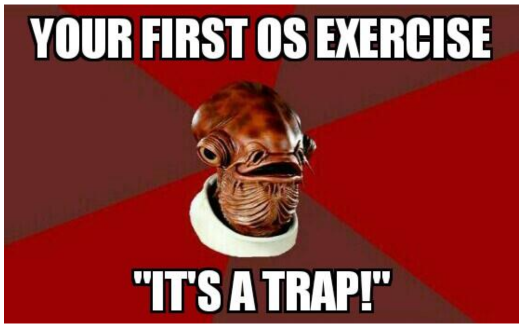
In this exercise, we will measure the overhead involved in traps and virtualisation.

<u>Assignment</u>

This assignment has four parts:

1. Coding a library that measures the time it takes to run various operations
2. Setting up a virtual machine (VM)
3. Setting up a container
4. Using the library to measure times
    - Directly on the computer
    - inside the VM
    - inside the container

    and showing the results in a graph



<u>1. The OSM Library</u>

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

1. A single addition instruction.
2. An empty function call with no arguments.
3. A trap.

To measure the time it takes to perform an operation, you must use the function *gettimeofday* (run *man gettimeofday* for more details). Using other functions is not allowed. Since any single operation takes a very short time, you have to run each operation multiple times (iterations) between two *gettimeofday* calls and calculate the average time the operation took. The exact number of iterations is supplied 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 uses a trap to 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. Obviously, you don't have to check the return status of this function (it is an empty function, so it always runs successfully).

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 operation, it is advisable that you perform loop unrolling, that is: in every iteration of the loop, run many operations instead of just one. When measuring instructions, 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.

Furthermore, verify that your code actually measures what you think it does. You can do so by examining the assembly resulting from compiling your functions. For this you can either use https://godbolt.org/ or compile your code with the -S flag:

```
g++ -S -fverbose-asm -g osm.cpp -o osm.s
```

2. The Virtual Machine

You will use kvm and qemu for running a machine with *Tiny Core Linux*. This distribution is very small and minimalistic so as part of setting up, you will install things we usually take for granted, such as ssh and g++.

The following steps should result in a machine suitable to the needs in this exercise:

1. Download the VM image here: https://sourceforge.net/projects/gns-3/files/Qemu%20Appliances/linux-tinycore-linux-6.4-2.img
2. Run the machine using QEMU:

   ```
   qemu-system-x86_64 -hda linux-tinycore-linux-6.4-2.img
   ```

3. Inside the VM start the terminal and run:
   a. tce-load -wi openssh.tcz
   b. tce-load -wi compiletc.tcz

   This will install g++ and scp on the VM

2. The Container

You will use Singularity and Docker for running the container. Just execute those 3 commands in your terminal:

1. module load singularity
2. singularity build dgcc.simg docker://gcc
3. singularity run dgcc.simg

Now every command you are running in this terminal is being executed inside the container. You can try to run Strace here and find out it doesn’t exist in this context. When running your program in here you will be measuring the running times of instruction executed inside a container.

Note: The command “singularity build dgcc.simg docker://gcc” downloads the container from the internet, and saves it to your computer. If you don’t have enough free space in your disk, you can save it to a disk on key, and run it from there. It is less than 400MB. Just run:

```
singularity build /path/to/your/diskonkey/dgcc.simg docker://gcc
```

```
singularity run /path/to/your/diskonkye/dgcc.simg
```

3. Measurement

Now that everything is ready, you can measure the average times of all three operations both inside and outside of the VM. After gathering the data, you will create a graph of the results and submit it as an image file bundled together with your code.

In order to measure the times inside the VM you will have to copy your source code there and compile it. You may want to use the “scp” program (check *man scp*) to transfer files over the network (e.g., from your home folder).

Example:

```
scp <username>@pond.cs.huji.ac.il:~/osm.cpp ./
```

Use the 9 measured values to create a bar graph comparing the results directly on the machine, inside the VM, and inside the container (The running times of the container will be very similar to one of the other measurements). You can use python/Matlab/Excel or any other program. The x-axis should have 3 values for the 3 operations measured. Use a logarithmic scale on the y-axis if necessary.

Make sure that the graph is self-explanatory, namely:

1. It has a legend.
2. It has a caption.
3. Its axes have labels and units where applicable.

Guidelines

- **Do not change the** header file**. Your exercise should work with our version of** osm.h**.**
- The programming in this exercise is trivial. But you must look at the results and think how to make them reasonably accurate. This may take time.
- There is no single solution, and multiple ideas may work.
- **How can you tell if 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 affect measurements.
    - Above a certain threshold, the number of iterations should not affect the average time of an operation
    - The time you measure for a function call should be at least an order of magnitude larger than that of a single instruction, the time for a trap is orders of magnitude larger than both.
- Make sure to check the exit status of all system calls you use.
- Make your code readable (indentation, function names, etc.).

Submission

Submit a tar file 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 assignments**. 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 Assignment 2.
- Your Makefile for Assignment 2. Running *make* with no arguments should generate the *libosm.a* library. You can use this Makefile as an example.
- An image file with the graph of the results. (in .png format)

Late submission policy						
Submission time	24.3, 23:55	25.3, 23:55	26.3, 23:55	27.3, 23:55	28.3, 23:55	29.3
Penalty	0	-3	-10	-25	-40	**-100**