Introduction to Operating Systems

Project 1B: Linux kernel debugging (KGDB) and profiling (PERF)

Deadline:

2020-10-30 (Fri) 23:59:59

Q&A:

If you have any questions, please post it on the E3 discussion board, and it will be answered in two days.

Deliverables:

1. **Demo video** (5-7 minutes – upload to YouTube – add link in the first page of the report).
2. **Report** (pdf file with file name of the form **OS_Project1B_StudentID.pdf**)
    - Screenshots + one explanation paragraph per screenshot.
    - Answers to questions below.
3. In the demo video and report, for each screenshot, explain:
    a. What has been done, and
    b. The reasoning behind the steps.

Objective: The objective of this project is to help the student to get familiar kernel debugging and profiling. The student will learn the definitions of debugging and profiling, how to prepare its working environment, how to trigger break points and how to profile and interpret the profiling results of kernel functions.

Scope: Understand the concept of kernel debug, kernel profile, and to learn how to prepare the working environment and how to interpret the profiling results.

Table of contents:

- Section 1: Debugging Kernel Functions
- Section 2: Profiling Kernel Functions

Questions to be answered in the report:

1. What is a kernel function? What is a system call?
2. What is KASLR? What is it for?
3. What are GDB's non-stop and all-stop modes?
4. Explain what the command **echo g > /proc/sysrq-trigger** does.
5. Questions of section 2.2 (page 23).

Also, in your report, remember to include:

6. **Do it yourself** exercise of Section 1 (page 12).
7. **Do it yourself** exercise of Section 2 (page 24).

Useful links:

- Project 1A tutorial:
  https://www.youtube.com/playlist?list=PL4l5tG9L6WbQrIvxdfg9KXt0OLxfrwdlu
- How to disable KASLR:
  https://stackoverflow.com/questions/49360506/in-kgdb-i-cannot-set-the-breakpoint
  https://askubuntu.com/questions/964540/gdb-qemu-cant-put-break-point-on-kernel-function-kernel-4-10-0-35
- Linux system calls table:
  https://filippo.io/linux-syscall-table/
- Linux kernel 4.19.148 source code:
  https://elixir.bootlin.com/linux/v4.19.148/source
- Examples on KGDB:
  https://developer.ridgerun.com/wiki/index.php/How_to_use_kgdb
- How to build the perf tool:
  https://blog.csdn.net/tang05505622334/article/details/103057179
- Notepad ++ compare plugin:
  http://www.technicaloverload.com/compare-two-files-using-notepad/

If you have any questions, please remember to search online before posting them in the forum.

SECTION 1:

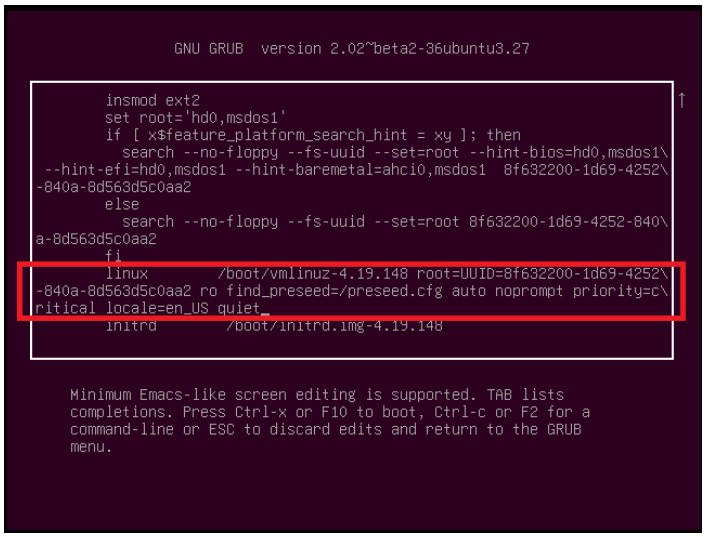
DEBUGGING KERNEL FUNCTIONS

Section 1: Debugging kernel functions

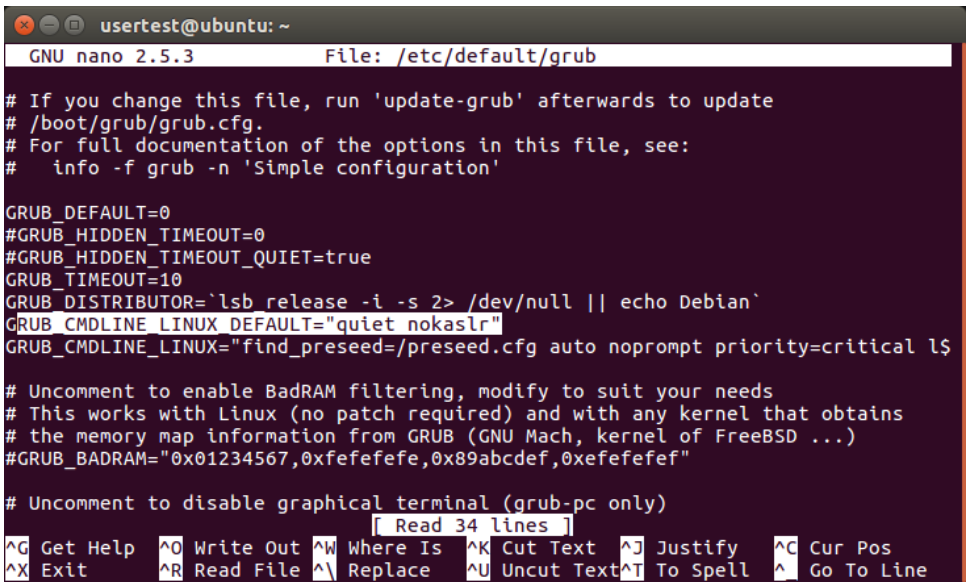
In this section, we will debug some basic kernel functions.

NOTE: You need to finish project 1A in order to do this one. If you haven't finished it, please refer to these tutorial videos:
https://www.youtube.com/playlist?list=PL4l5tG9L6WbQrIvxdfg9KXt0OLxfrwdlu

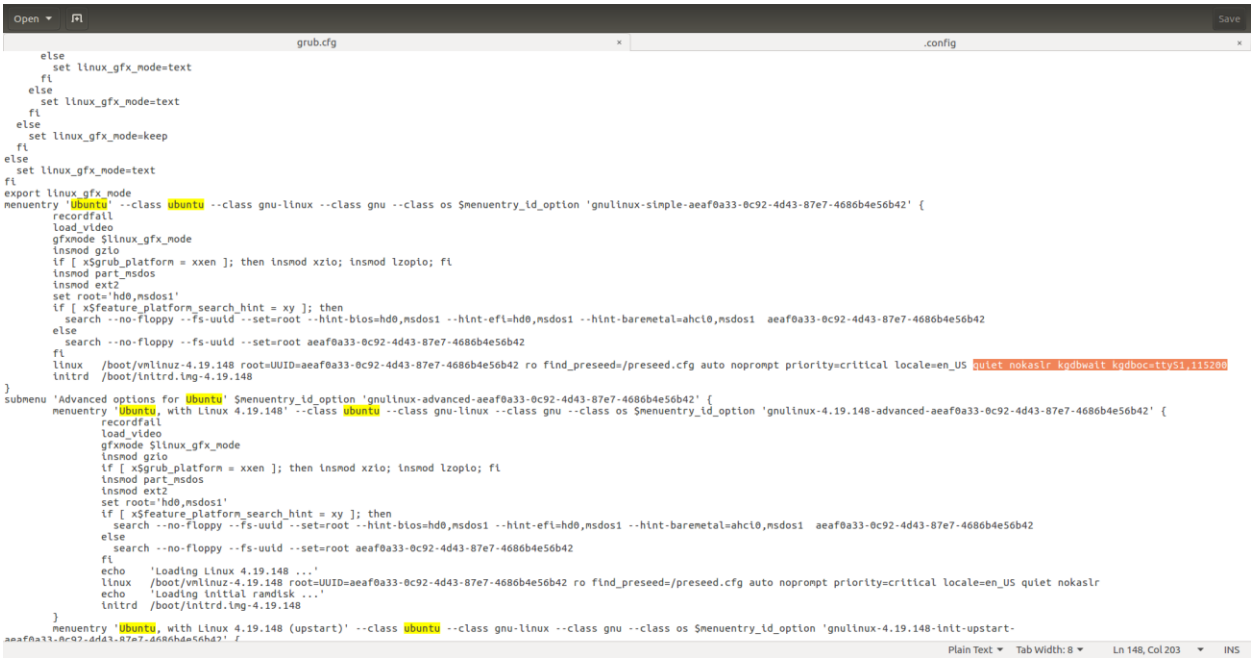
1. Turn on the Target virtual machine and bypass the kgdb commands in the grub entry. (Refer to Section 4.0.2 in project 1A)



2. We need to disable KASLR from the grub.
https://stackoverflow.com/questions/49360506/in-kgdb-i-cannot-set-the-breakpoint
Run the command
$sudo nano /etc/default/grub
And in the GRUB_CMDLINE_LINUX_DEFAULT line, add "nokaslr".

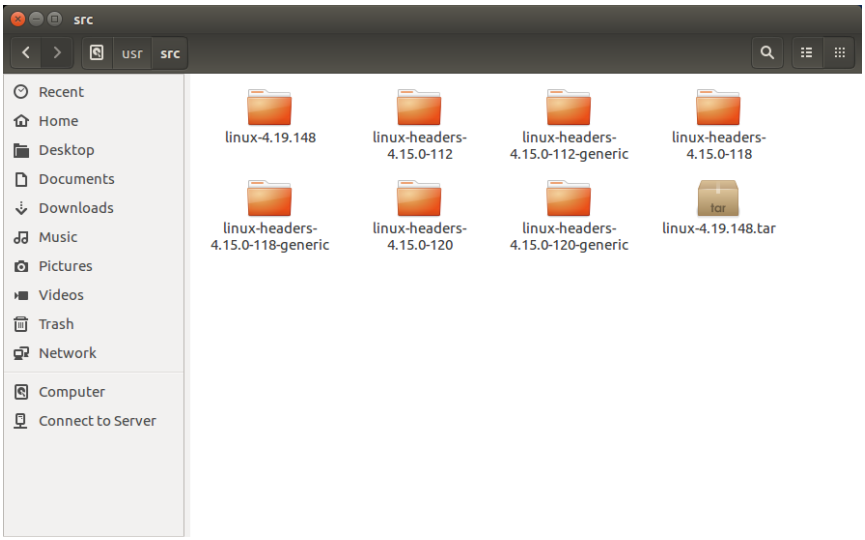


3. Update the grub with the command
   $sudo update-grub
4. In the /boot/grub/grub.cfg file, re-add the commands **kgdbwait kgdboc=ttyS1,115200** (You need to do this step after every time you run the update-grub command).



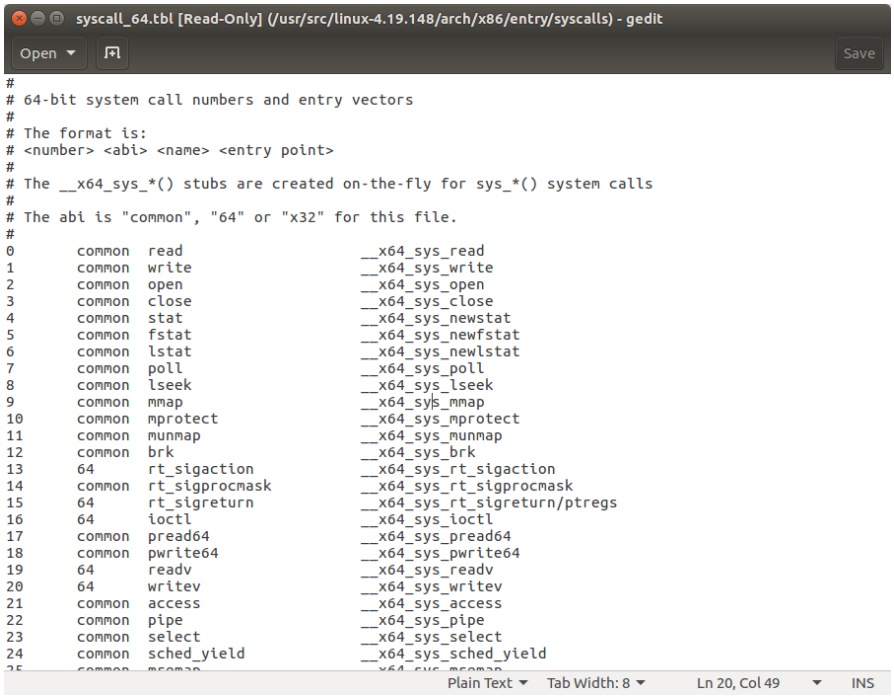
[Screenshot # 1 and #2: Create screenshots showing how you update the grub and the grub.cfg file.]

5. (This step is optional, but it could be useful for later steps).
   In the Host machine, download again **the same kernel source code that you have in the target machine**. Download the tar.xz, decompress it and untar it and save it in /usr/src/.



(An online version is here https://elixir.bootlin.com/linux/v4.19.148/source, but beware that it may not be accurate. It is advised to re-download the kernel source code).

6. Restart the target machine. It should reach the "*Waiting for connection from remote gdb*" message.
7. KGDB will debug **KERNEL FUNCTIONS**. In the kernel source code folder, check for the **/arch/x86/entry/syscalls/syscall_64.tbl** file. This has the list of the kernel functions with its system call number.

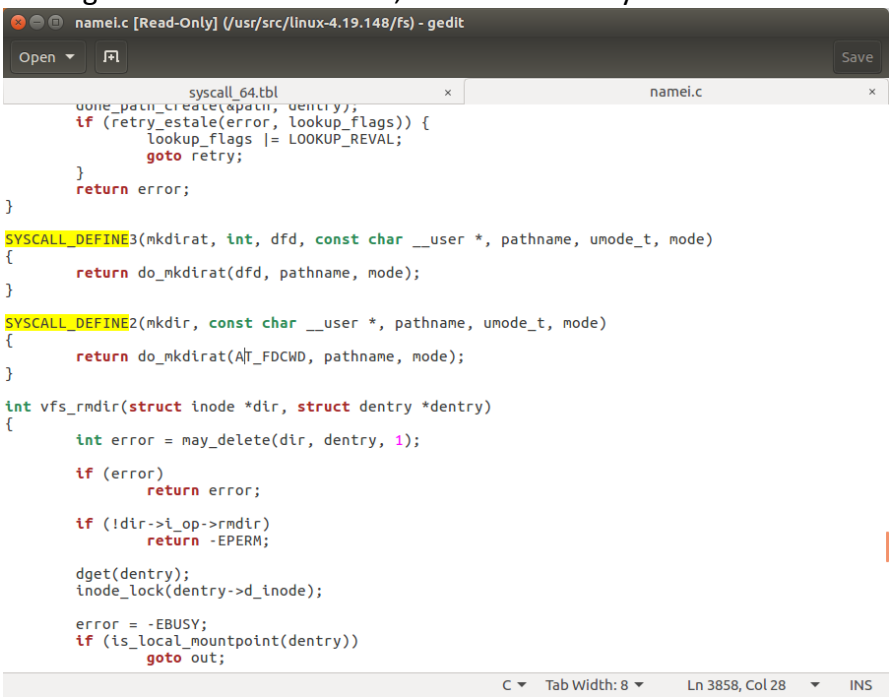


8. We will create a break point for the **mkdir** function, which will be triggered when we create a new folder.
   Look online for at least 3 tables showing where each kernel function is implemented. An example is https://filippo.io/linux-syscall-table/, but keep in mind that some functions inside the syscall_64.tbl file might be missing in the online tables. (Check for Linux system calls only, not other OSs, and the tables should be consistent between them).
   In the table, we find that the mkdir function is implemented in the **fs/namei.c** file. (These paths are inside the kernel folder).

79	getcwd	sys_getcwd	fs/dcache.c
80	chdir	sys_chdir	fs/open.c
81	fchdir	sys_fchdir	fs/open.c
82	rename	sys_rename	fs/namei.c
83	mkdir	sys_mkdir	fs/namei.c
84	rmdir	sys_rmdir	fs/namei.c
85	creat	sys_creat	fs/open.c
86	link	sys_link	fs/namei.c

[Screenshot # 3, #4, #5 and #6: Create a screenshot showing your syscall_64.tbl file (displaying your student ID) and a screenshot showing each syscall table you found online.]

In the namei.c file, check for an entry of the form **SYSCALL_DEFINE[N](mkdir,…)**, where [N] is an integer number. In this case, we find the entry as below



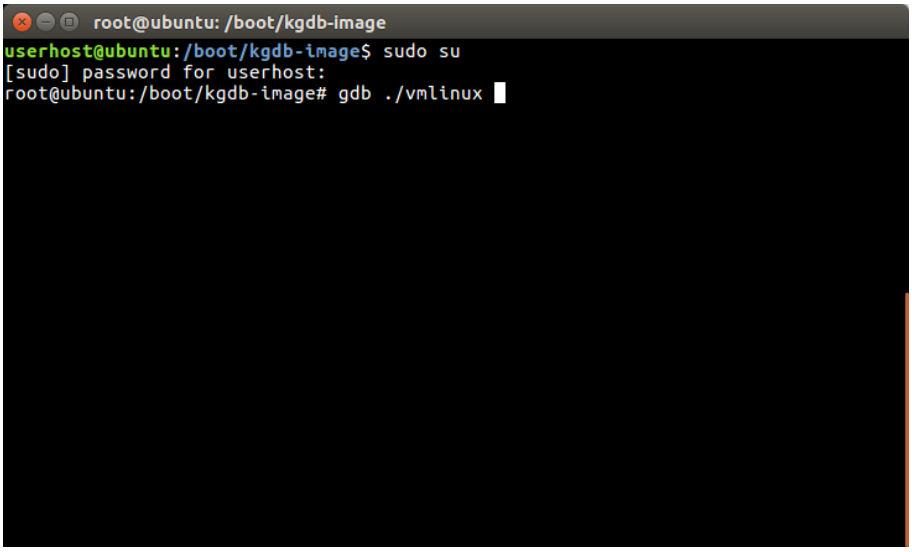
From the last picture we see that the mkdir function calls the **do_mkdirat** kernel function, and has the pathname as one of the parameters. We will create a breakpoint in this function.

9. In the host machine, connect KGDB to the target machine (project 1A – Section 4.1) with the commands

   cd /boot/kgdb-image

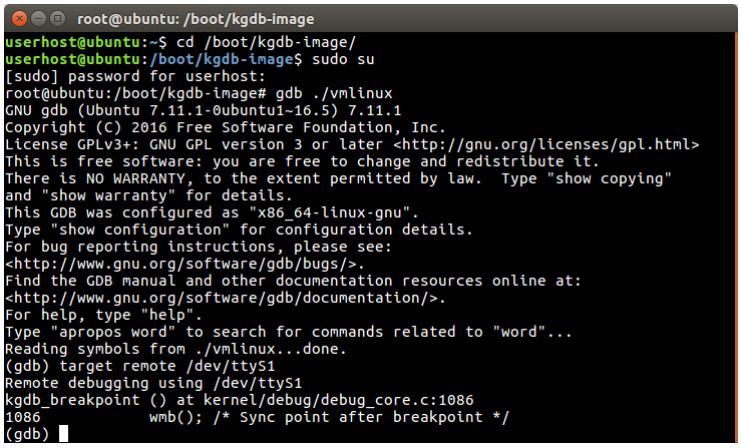
   sudo su

   # gdb ./vmlinux

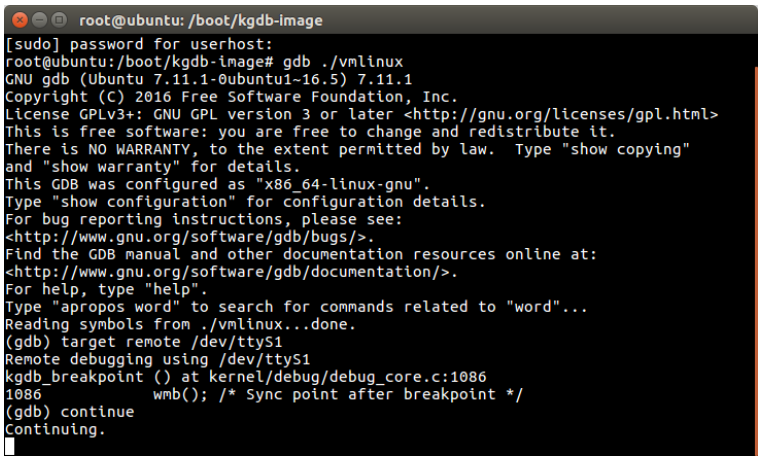


And connect to the target by using

(gdb) target remote /dev/ttyS1



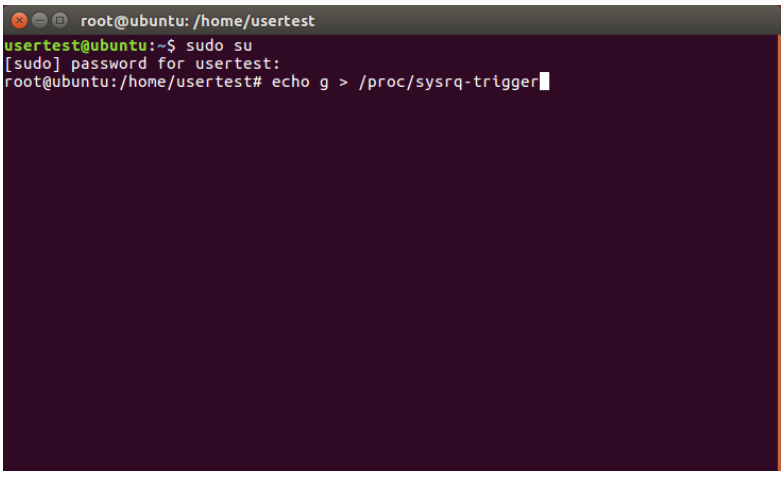
Type **continue**, so the target machine boots normally.



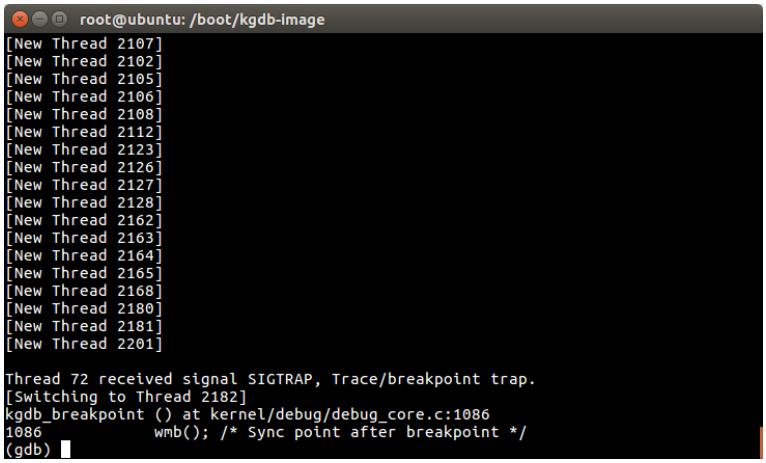
[Screenshot # 7: Create a screenshot (showing your student ID) of this step.]

10. In the target machine, we need to send a signal to the host to re-take control in GDB. In a terminal (as root), run the command

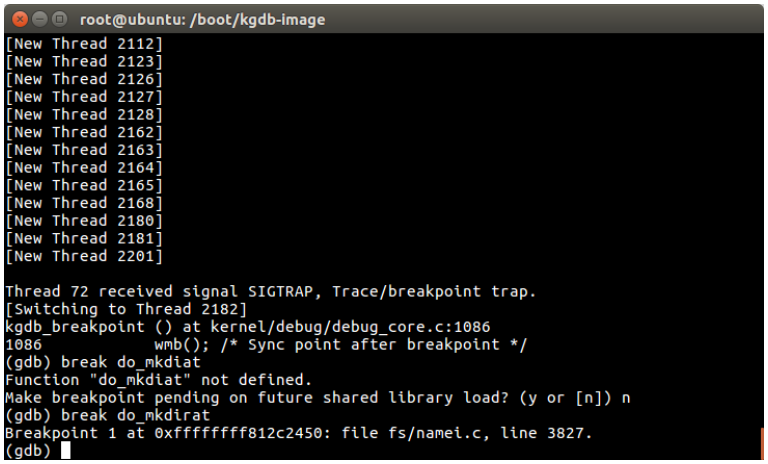
    # echo g > /proc/sysrq-trigger



The Target machine will freeze and in the host machine, you will get access again to GDB.

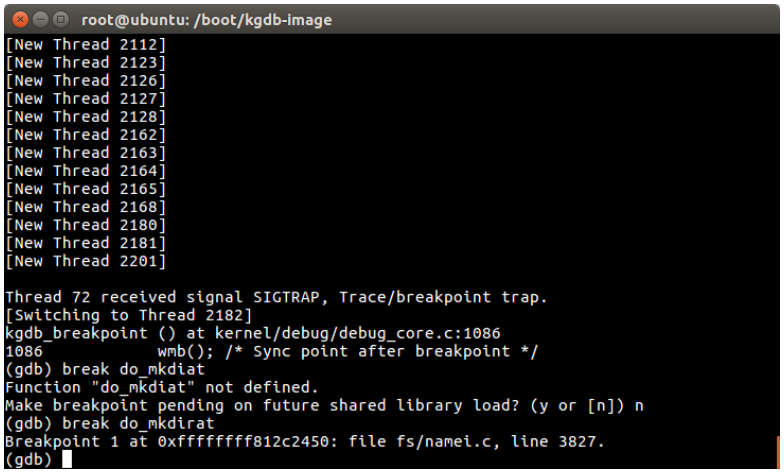


11. We create a breakpoint with the command
    (gdb) break do_mkdirat



    It will create a breakpoint and give to you its ID (in this case is 1).

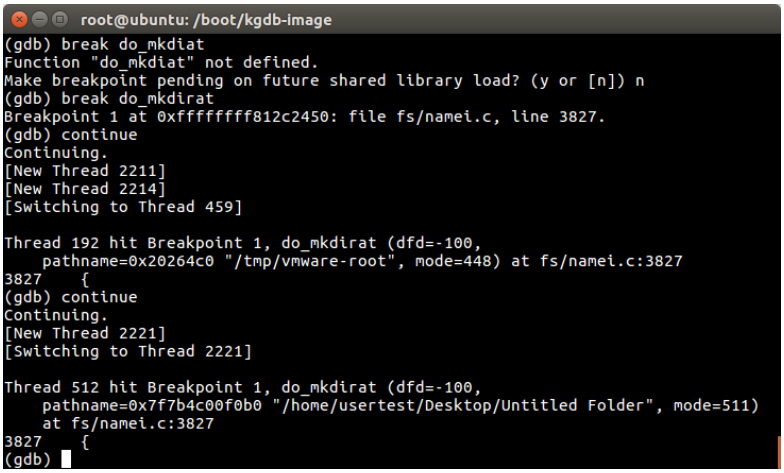
12. Type **(gdb) continue**, and the target machine will unfreeze.



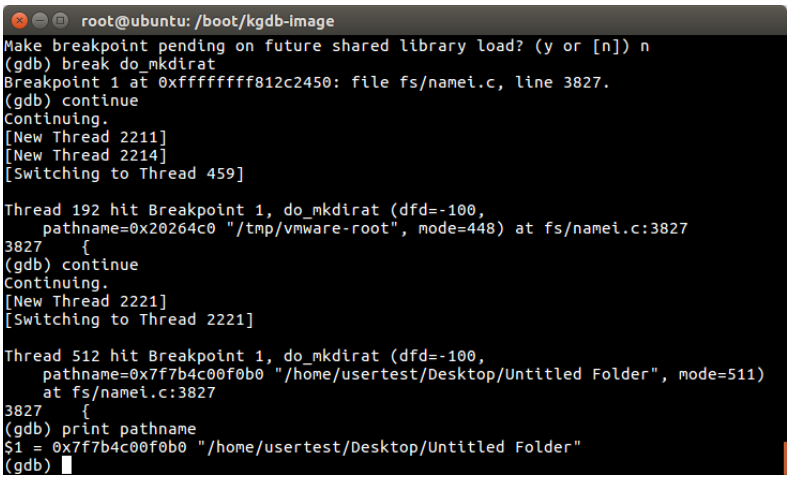
    (When you go back to the target, if it is still frozen, come back to GDB and check if a breakpoint was hit. Type continue until the target machine is responsive again).

13. Now in the Target's desktop, create a directory. This will trigger the break point and the Target machine will freeze again.

14. In the host machine, we see that a breakpoint was hit.



We can check the parameters of this function. By using the command **(gdb) print pathname**, we should get the path where the new folder was created in the target machine.



[Screenshot # 8: Create a screenshot showing the pathname parameter value. (It should include your student ID)]

15. Type **(gdb) continue** until the folder is created successfully.

```
root@ubuntu: /boot/kgdb-image
[Switching to Thread 2221]

Thread 512 hit Breakpoint 1, do_mkdirat (dfd=-100,
    pathname=0x7f7b4c00f0b0 "/home/usertest/Desktop/Untitled Folder", mode=511)
    at fs/namei.c:3827
3827    {
(gdb) print pathname
$1 = 0x7f7b4c00f0b0 "/home/usertest/Desktop/Untitled Folder"
(gdb) continue
Continuing.
[New Thread 2228]
[New Thread 2224]
[New Thread 2225]
[New Thread 2226]
[New Thread 2229]
[New Thread 2230]
[New Thread 2227]
[Switching to Thread 1]

Thread 2 hit Breakpoint 1, do_mkdirat (dfd=-100,
    pathname=0x56295dad9350 "/sys/fs/cgroup/systemd/system.slice/NetworkManager-
dispatcher.service", mode=493) at fs/namei.c:3827
3827    {
(gdb) continue
Continuing.
[Switching to Thread 459]

Thread 192 hit Breakpoint 1, do_mkdirat (dfd=-100,
    pathname=0x2026290 "/tmp/vmware-root", mode=448) at fs/namei.c:3827
3827    {
(gdb) continue
Continuing.
[Switching to Thread 1]

Thread 2 hit Breakpoint 1, do_mkdirat (dfd=-100,
    pathname=0x56295dad9350 "/sys/fs/cgroup/devices/system.slice/NetworkManager-
dispatcher.service", mode=493) at fs/namei.c:3827
3827    {
(gdb) continue
Continuing.
[New Thread 2232]
[Switching to Thread 364]

Thread 184 hit Breakpoint 1, do_mkdirat (dfd=-100,
    pathname=0x561b2fd09347 "/run/systemd/journal/streams", mode=493)
    at fs/namei.c:3827
3827    {
(gdb) continue
Continuing.
(gdb)
```

[Screenshot # 9: Create a screenshot showing both virtual machines, the Host already passed all the continues, and the Target with the new folder created].

A video showing all the steps in this section can be found here:

https://www.youtube.com/watch?v=e-RgDwHOlPk

16. **[Do it yourself]** Select another function from the **syscall_64.tbl** table, look for where it is implemented and create your own scenario to trigger it.

Required to explain in both the report and in the video:

a. Which function you selected.
b. The file that contains it (with its path).
c. The kernel function that is called.
d. Which parameters the kernel function has.
e. Be creative on how to trigger it. If you need to create a c program to trigger it, feel free to do so.

Requirements:

a. The Target machine **must be totally on** and responsive before hitting the breakpoint.
b. You must display the value of at least one of the parameters that the function receives.
c. You must include at least 6 screenshots showing:
    i. That the machine was on and responsive **[Screenshot # 10]**,
    ii. What is your scenario and how to trigger it **[Screenshot # 11 - #12]**,
    iii. How do you create the break point **[Screenshot # 13]**,
    iv. The host machine hitting the break point, and the value of at least one parameter **[Screenshot # 14]**, and
    v. The Target machine working again, with the action totally finished **[Screenshot # 15]**.

SECTION 2:

PROFILING KERNEL FUNCTIONS

Section 2: Profiling Kernel functions

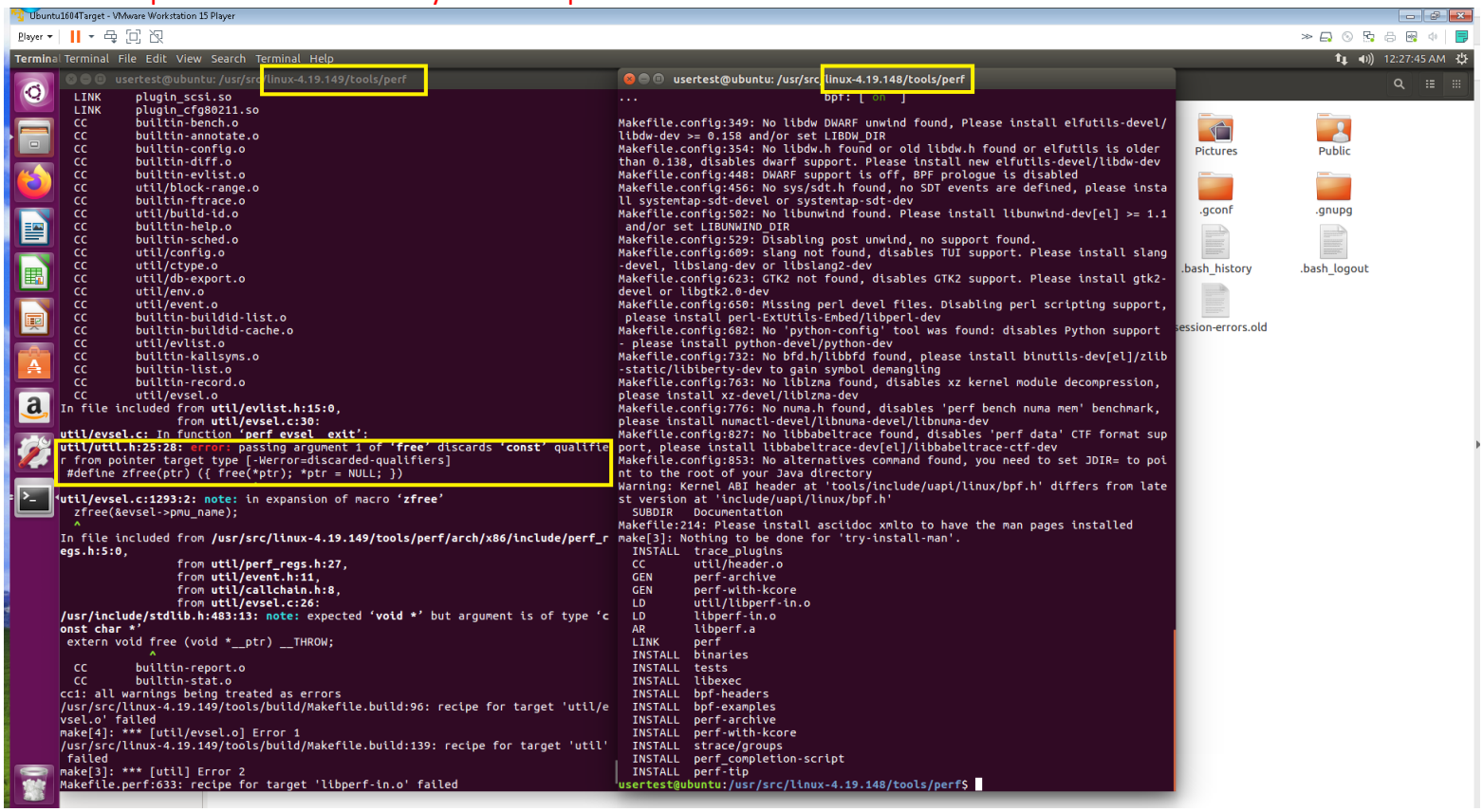
In this section, we will install the required tool to profile the kernel, and proceed to show how to profile and interpret some example functions.

AT THIS POINT, YOU CAN TURN OFF THE HOST MACHINE.

Section 2.0: (IMPORTANT) Perf source code patching

NOTE:

A problem with kernels 4.19.149 – 150 – 151 – 152 source code was recently found that prevents to successfully build the perf tool.



Git comments:

This problem was discovered by the official developers on Sept. 29-30/2020:

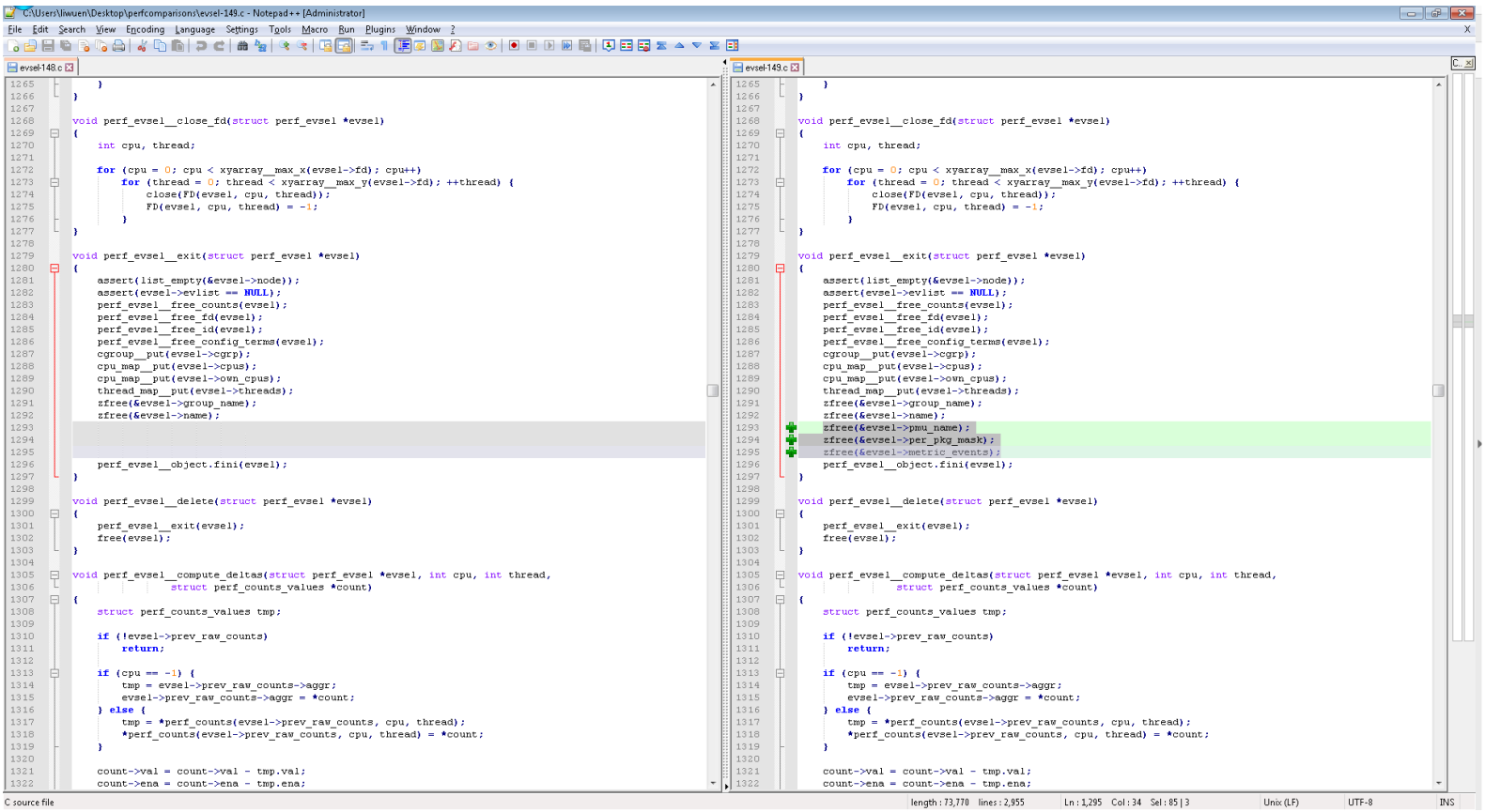
- https://lkml.org/lkml/2020/9/29/2330
- https://lkml.org/lkml/2020/9/30/1024

Solutions:

There are two official solutions for this problem:

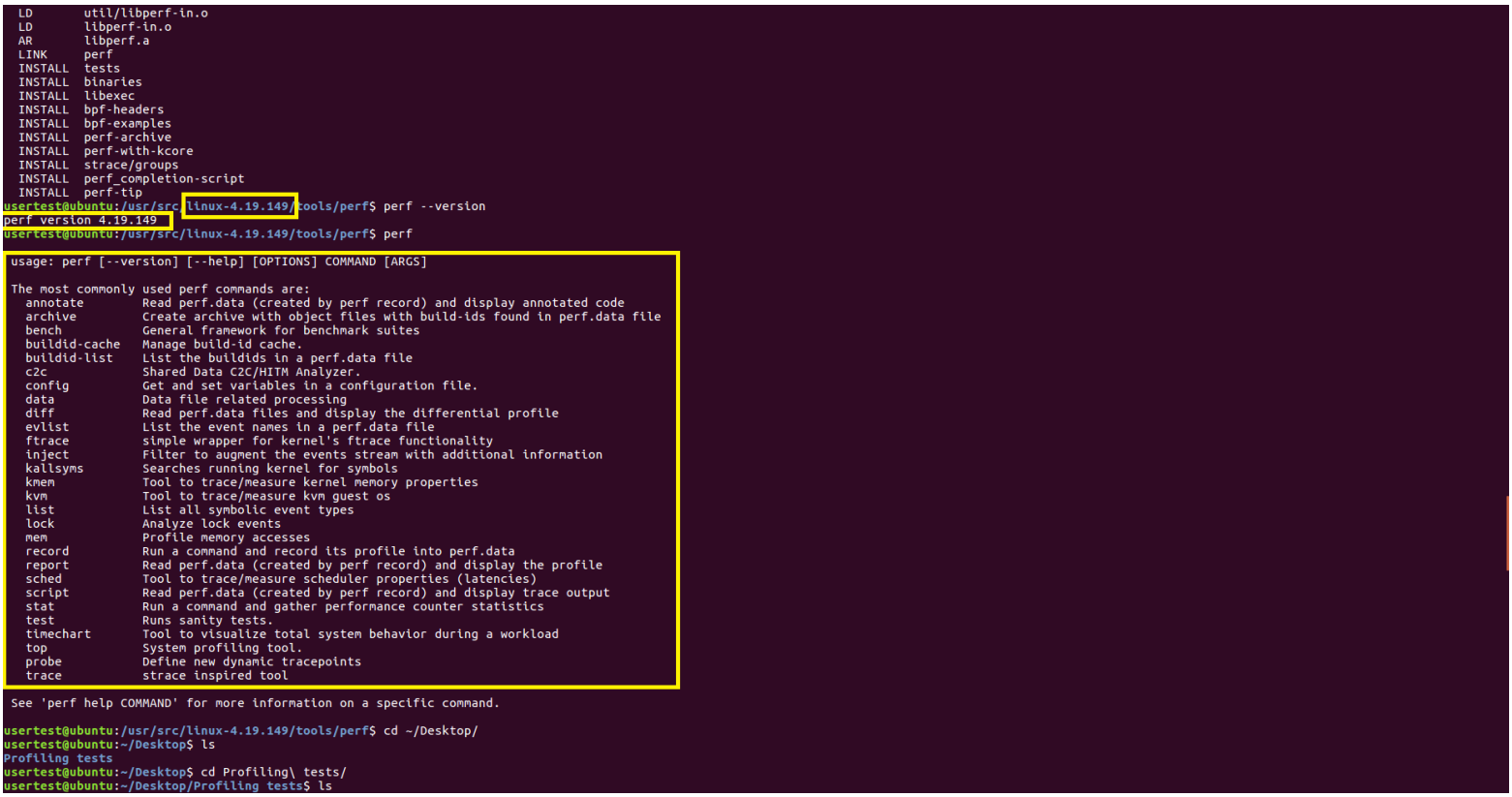
1. Use kernel 4.19.148:
   https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.19.148.tar.xz
   This solution requires Section 2 and Section 3 of project 1A.

2. In the affected kernels, go to **linux-4.19.149/tools/perf/util/evsel.c**.
   Locate the following function:
   void perf_evsel__exit(struct perf_evsel *evsel)
   and comment the lines:
   zfree(&evsel->pmu_name);
   zfree(&evsel->per_pkg_mask);
   zfree(&evsel->metric_events);
   (Around lines 1293, 1294 and 1295).



After commenting these lines, save the evsel.c file, and run the commands
$ sudo make
$ sudo make install
(Please refer to Section 2.1 Project 1B)

After patching this file, the build process should be successful.



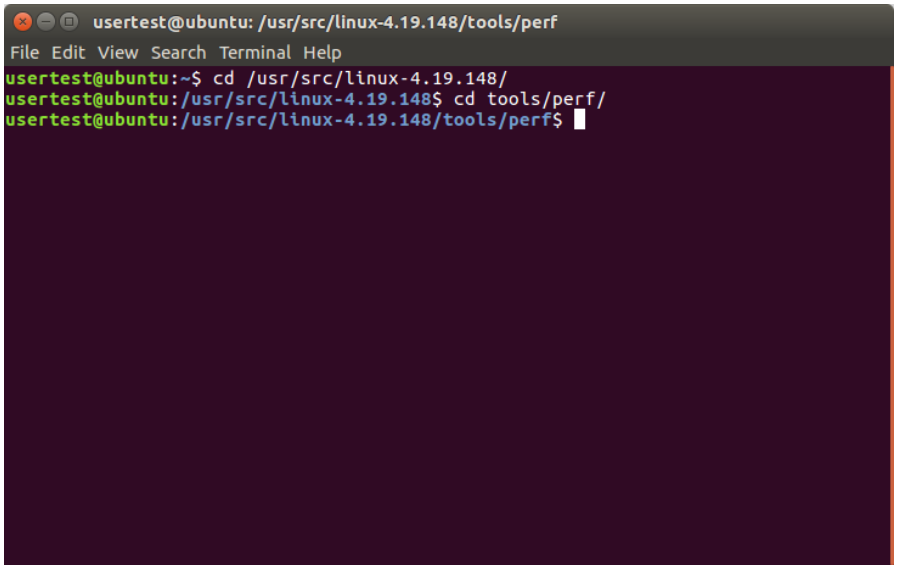
IMPORTANT NOTES:

1. This patch is a **best-effort** solution (so no guarantees that this will solve all the possible problems).
   If more problems are found later in the affected kernels, please use kernel 4.19.148.
2. In order to avoid this situation in the future, **Project 2 and Project 3 will require to use kernel 4.19.148** (so you will have to download and compile kernel 4.19.148)**.**
3. For more information, please check the related post on E3 forum: https://e3new.nctu.edu.tw/mod/forum/discuss.php?d=72257

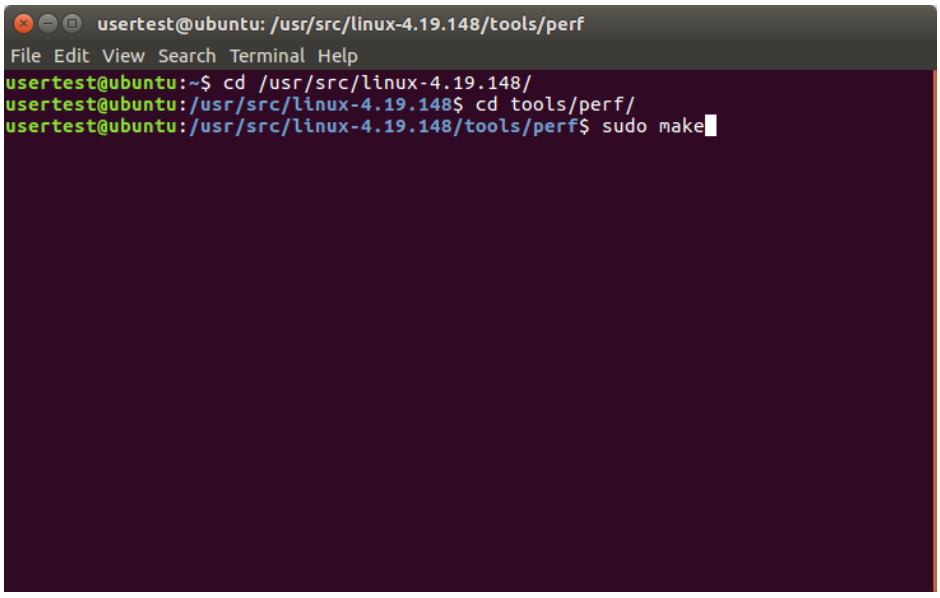
Section 2.1: Profiling tool installation

In this section, we will build the profiling function **perf**.

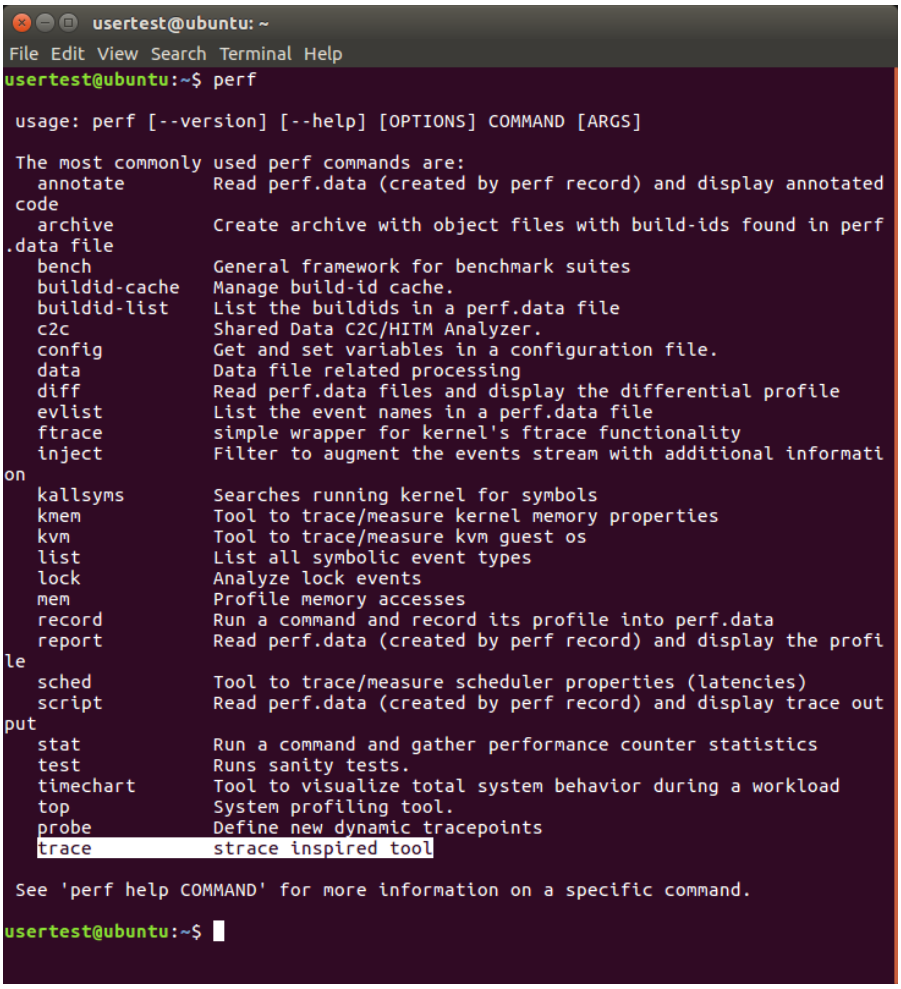
1. Go to the Linux source code folder, and go to **/tools/perf.**



2. Run the commands
   $sudo make
   $sudo make install



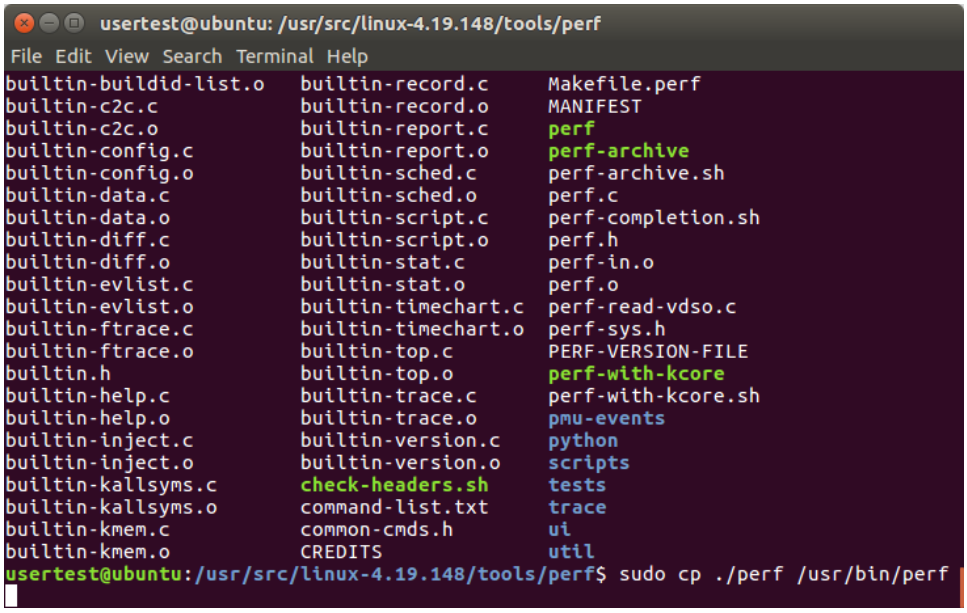
3. At this point, the perf tool should be created. Type **$perf** to verify that the tool was built successfully.



IMPORTANT NOTE: There is a chance that the command **trace** was not generated. If that happens, re-do step 2 and now it should appear in the list of commands.

[Screenshot # 16: Create a screenshot showing the perf tool commands like in the previous screenshot. Your student ID must be visible.]

4. We still need to move the compiled perf tool to a location where all the system can access it. Run the command
   $ sudo cp ./perf /usr/bin/perf



At this point, we are ready to profile kernel functions.

Useful links:

- https://blog.csdn.net/tang05505622334/article/details/103057179

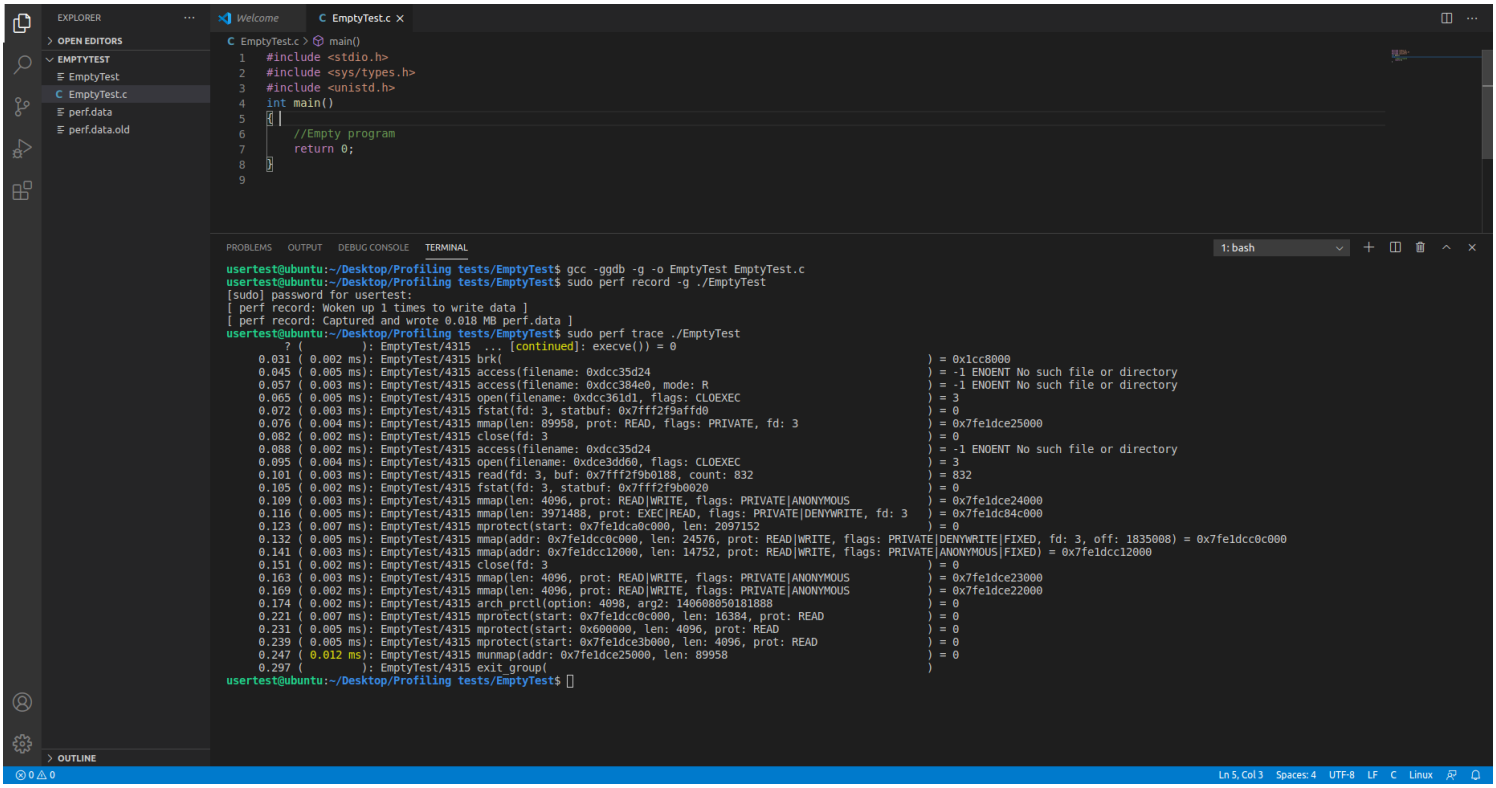
Section 2.2: Profiling functions

In this section, we will show how to use **perf**.

Notes:

- Perf will be used to profile c programs that use system calls.
- We need to use special headers in our programs (they depend on each system call).
- Perf must be executed as root (so sudo is necessary).
- The programs used in this section are available in E3.

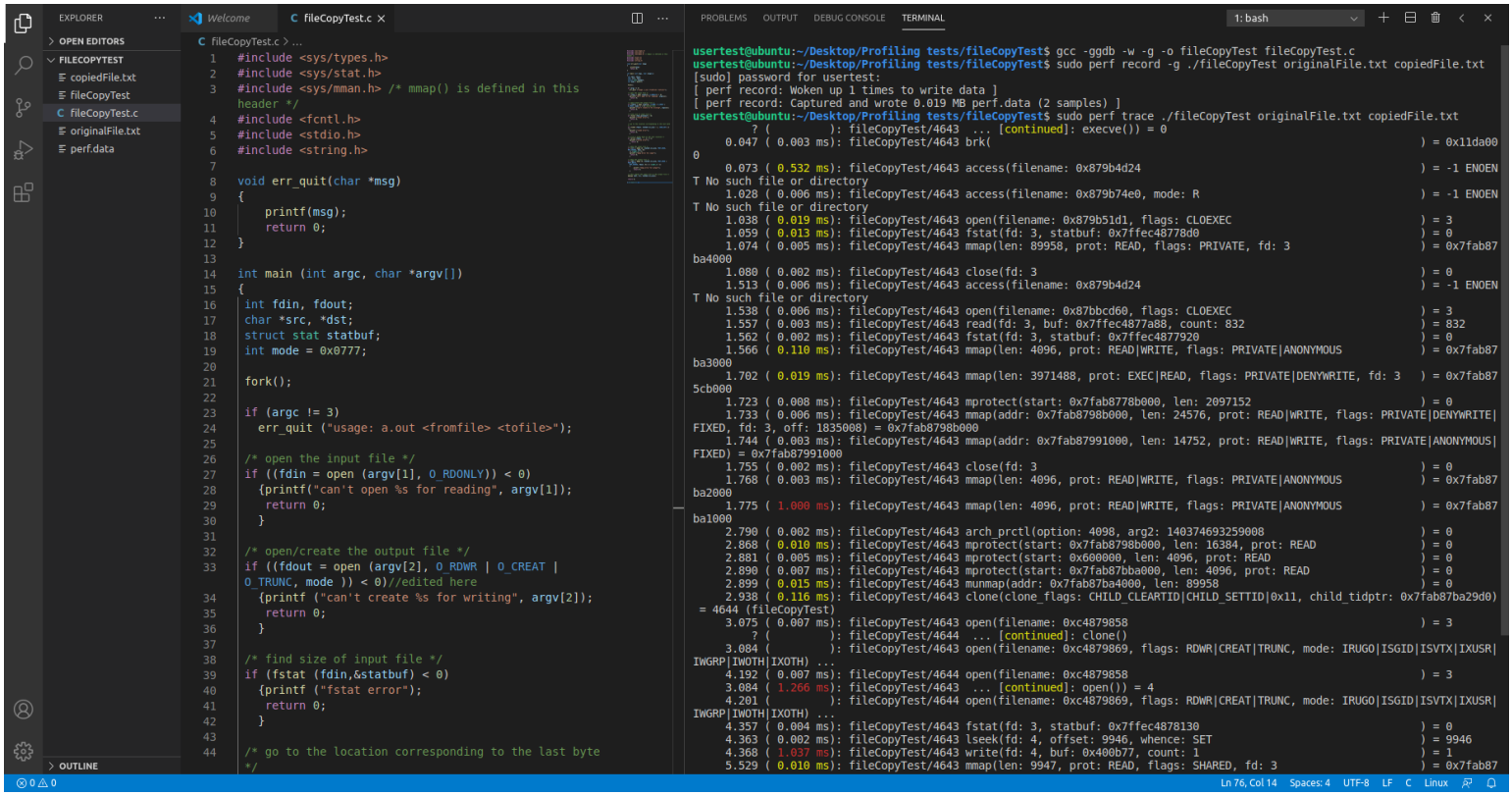
1. Create a folder called **Profiling tests** in your Desktop.
2. Create two folders: **emptyTest** and **fileCopyTest**.
3. For **emptyTest**, create the following program and call it **emptyTest.c**



4. Run the commands
   $ gcc -ggdb -g -o emptyTest emptyTest.c
   $ sudo perf record -g ./emptyTest
   $ sudo perf trace ./emptyTest
5. Copy the result of the trace command, and save it in a text file (call it **emptyTest.txt**).

From the previous image, you can see that perf record generates several lines of code for an empty file. These lines are common in any profiled compiled file. We will show how to ignore them.

6. For **fileCopyTest**, create the following program and call it **fileCopyTest.c**



This program copies one text file into another file. It uses the functions **fork, mmap, write,** and **printf.**

7. Download the **originalFile.txt** from E3 and place it in the same folder as **fileCopyTest.c**.
8. Create a new text file and call it **copiedFile.txt.**
9. Run the following commands

   $ gcc -ggdb -w -g -o fileCopyTest fileCopyTest.c

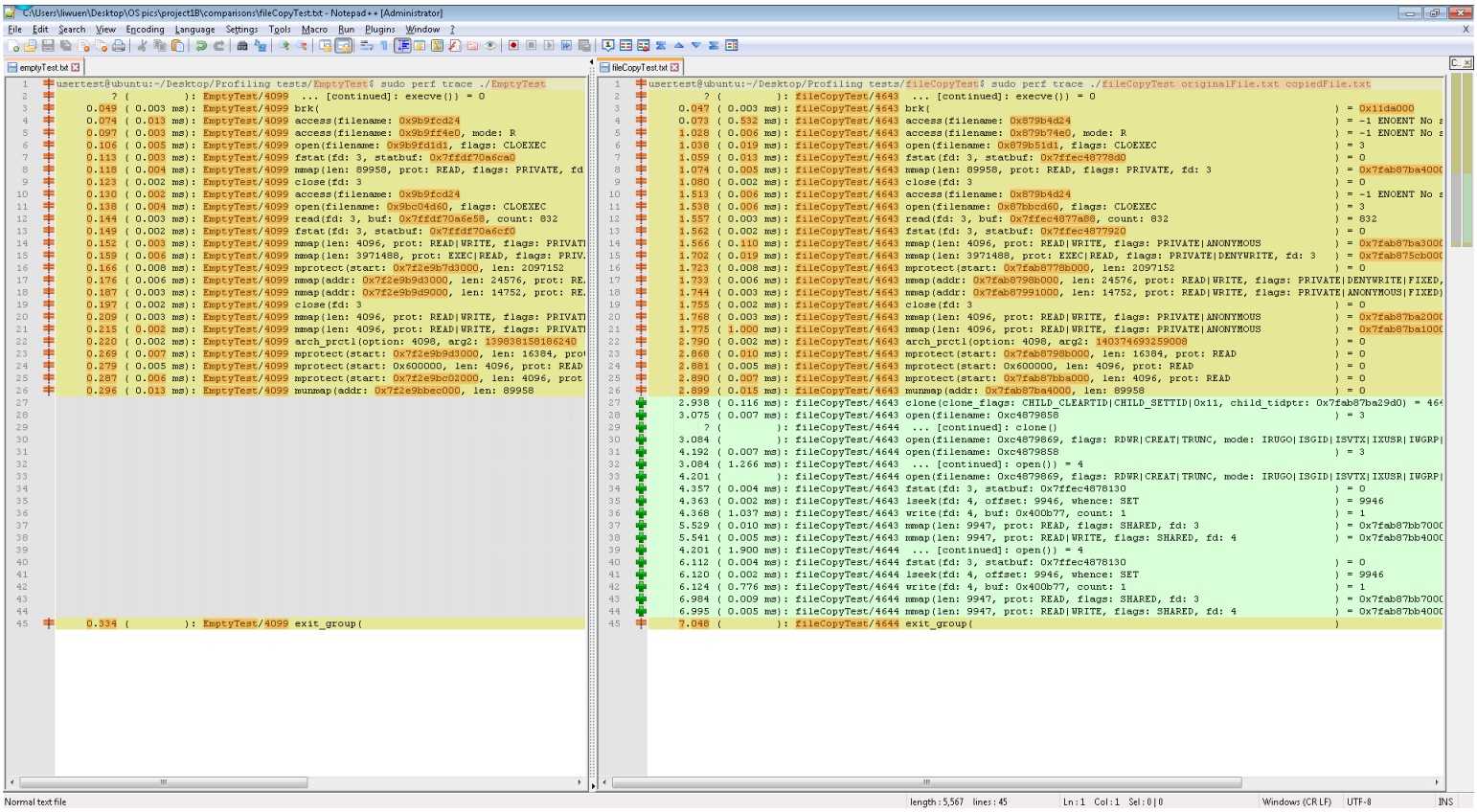
   $ sudo perf record -g ./fileCopyTest originalFile.txt copiedFile.txt

   $ sudo perf trace ./fileCopyTest originalFile.txt copiedFile.txt

[Screenshot # 17 and #18: Create two screenshots showing these files and the trace results, as shown above. Your student ID must be visible.]

Note: In the previous screenshots, we used **visual code** to display the code and the execution results. You are encouraged to use it, but if you wish to use any other developing tool feel free to do so. Just be sure that your student ID is visible in the screenshots.

10. Using Notepad++, compare both files.



Note: In this step we used Notepad++ and its **compare** plugin. In case you cannot install Notepad++ or wish to use another tool that displays the differences between two text files, please feel free to do so, but be sure that the differences are clear enough.

If you want to use Notepad++, here it is explained how to turn on the compare plugin:

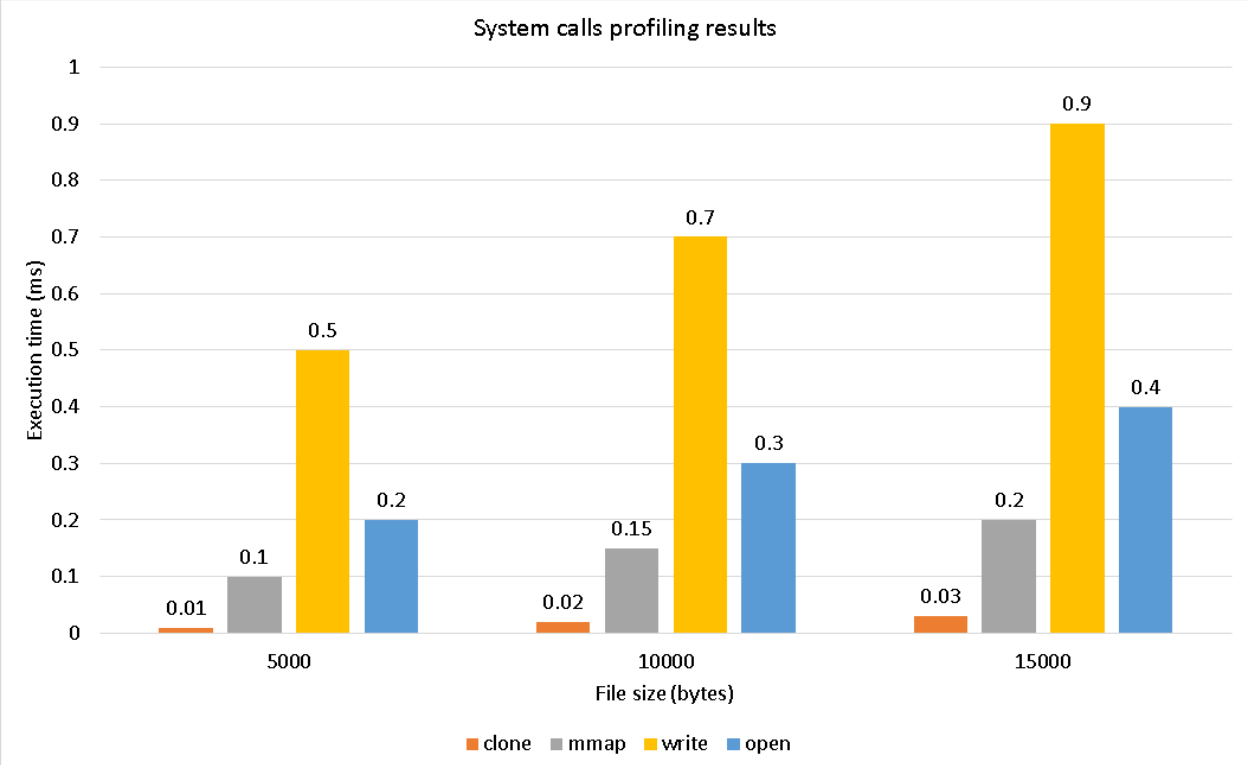
http://www.technicaloverload.com/compare-two-files-using-notepad/

[Screenshot # 19: Create a screenshot showing the differences between these files.]

We can see that **fileCopyTest.txt** has some extra lines which are invocations of the clone, mmap, write and open system calls. By using these lines we can calculate the average execution time of each function for this scenario.

Questions to answer (in both the report and video):

1. What are these functions: **clone**, **mmap**, **write** and **open?** **[Screenshot # 19]**
2. Why is there no **fork** system call? What is the difference between **fork** and **clone?** **[Screenshot # 19]**
3. Will the functions' execution time be longer if the file is bigger? **[Screenshot # 19]**
4. **Create a graph of file size (in bytes) vs. execution time (ms) of these four functions, using 3 different file sizes.**
   How is the behavior of each function? Sort them from slowest to fastest.
   (Example -from fastest to slowest- : clone, mmap, open, write).
   [Screenshot # 20]
   An example of the expected graph is shown below



(This is a dummy graph that only shows what is expected: **file size** vs. **execution time.** The real behavior of the functions is not reflected in this example.)

[Do it yourself] In the previous example we showed a comparisons between **fork**, **mmap**, **open,** and **write.**

Answer the following questions in both your report and video:

1. Perf also has the **report** command:
   $ **sudo perf report**
   Explain:
   a. What is it for? **[Screenshot # 21]**
   b. For **fileCopyTest**, show and interpret the results. **[Screenshot # 21]**
2. Perf has more commands (please refer to Section 2.1 step 3). Select another command (besides report, trace and record), explain what is it for and show how to use it. **[Screenshot # 22]**