Operating System

project 1B

Department: 土木5B
ID: 0511330
Name: 劉紘華
Vedio Link: https://youtu.be/LucTYTZDDg4

Goal

Get familiar with kernel debugging & profiling.
We will learn to set up a **working environment**, **trigger breakpoints** and profile and interpret the results

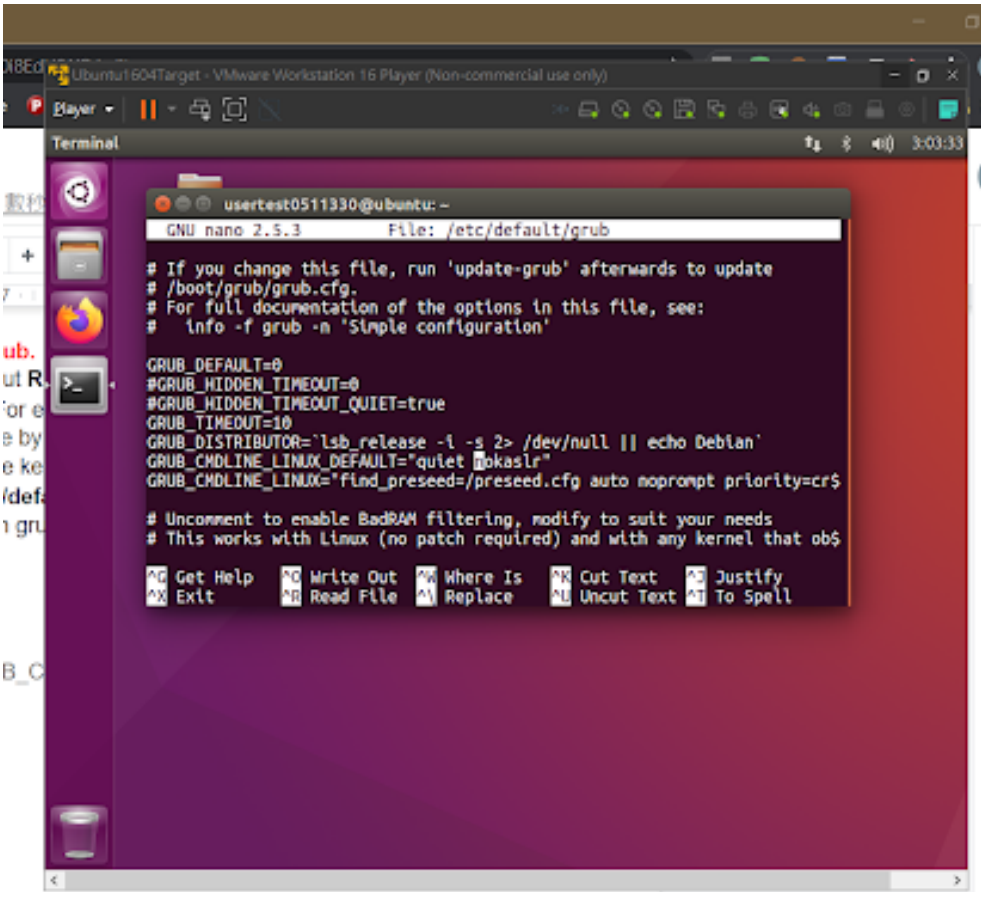
[Section 1-1]

[Screenshot 1] Disable KASLR from grub.

KASLR (**K**ernel **A**ddress **S**pace **L**ayout **R**andomization) is a mechanism that will put the kernel code at random locations, to prevent malicious attacks. For example, the **unprivileged** instruction **SITD** can be used to get the location of kernel code by user program. However, for our convenience, we tend to disable it in order to debug the kernel.

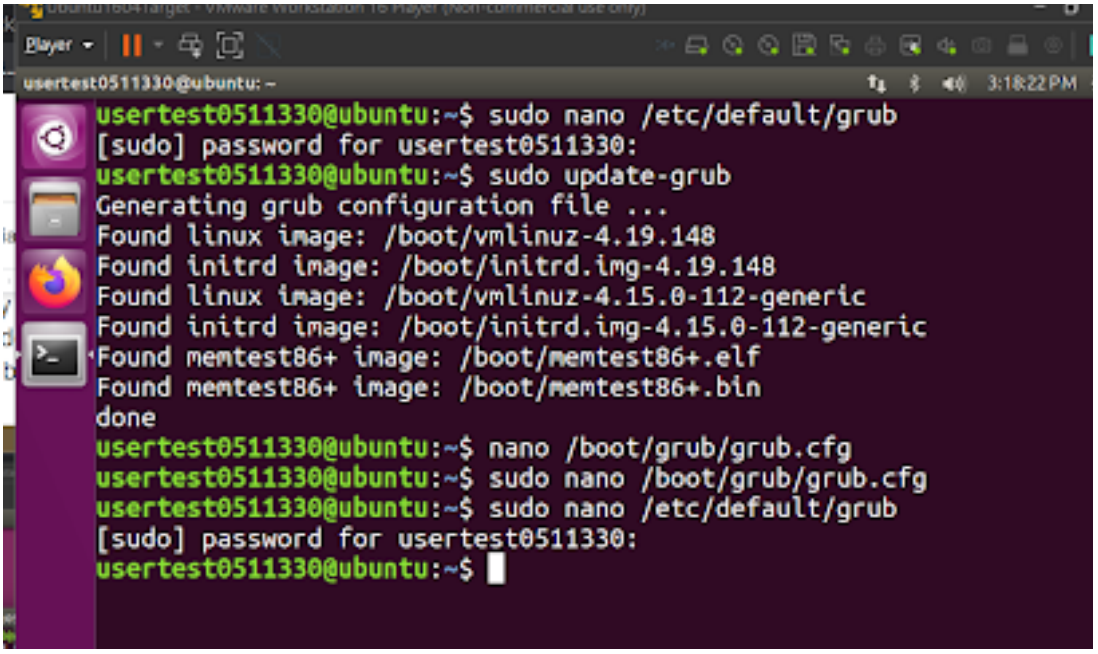
By using command: **$sudo nano /etc/default/grub** to add “nokaslr” right after the line “GRUB_CMDLINE_LINUX_DEFAULT” in grub

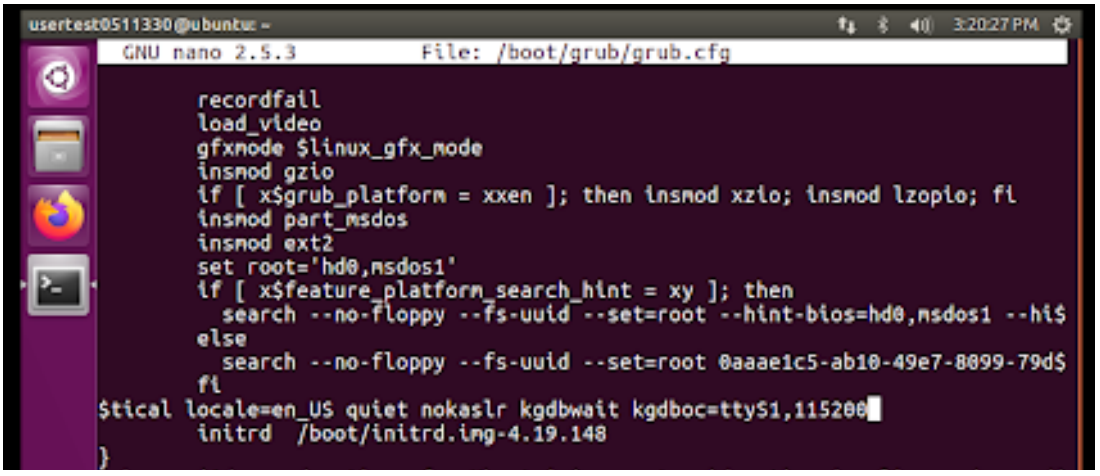
.



[Screenshot 2] Update the grub

1. Update the grub by using command: **$sudo update-grub**
2. **Don't Forget** to add the commands **kgdbwait kgdboc=ttyS1,115200** in grub.cfg every time we update the grub, which was mentioned in the previous project. (Using the command: **$sudo nano /boot/grub/grub.cfg**)

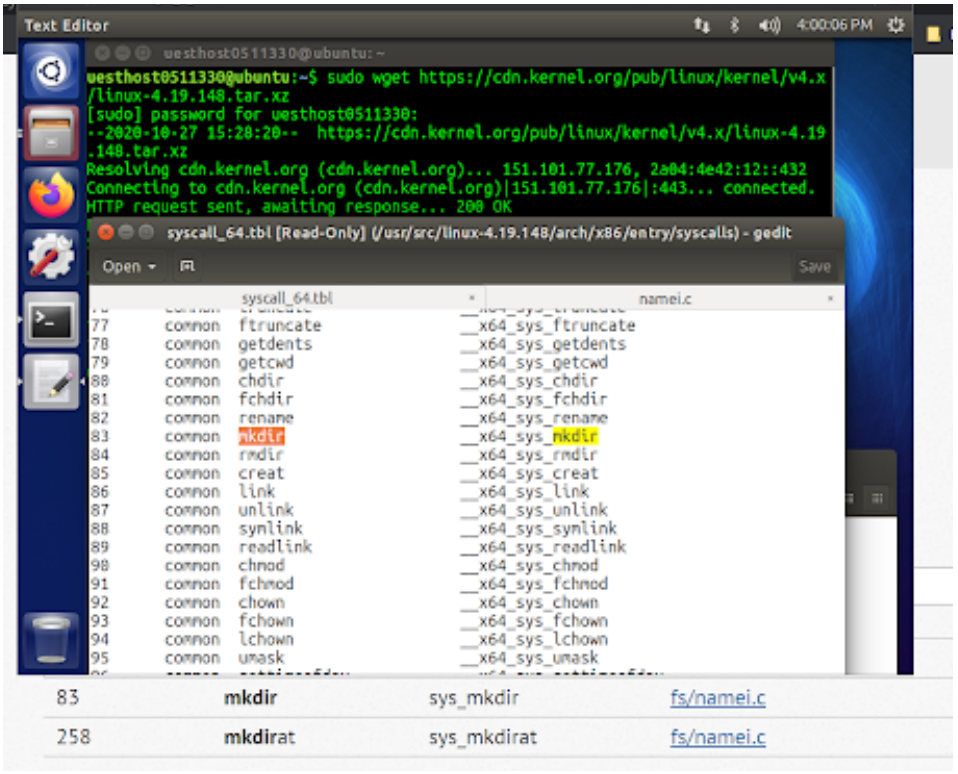




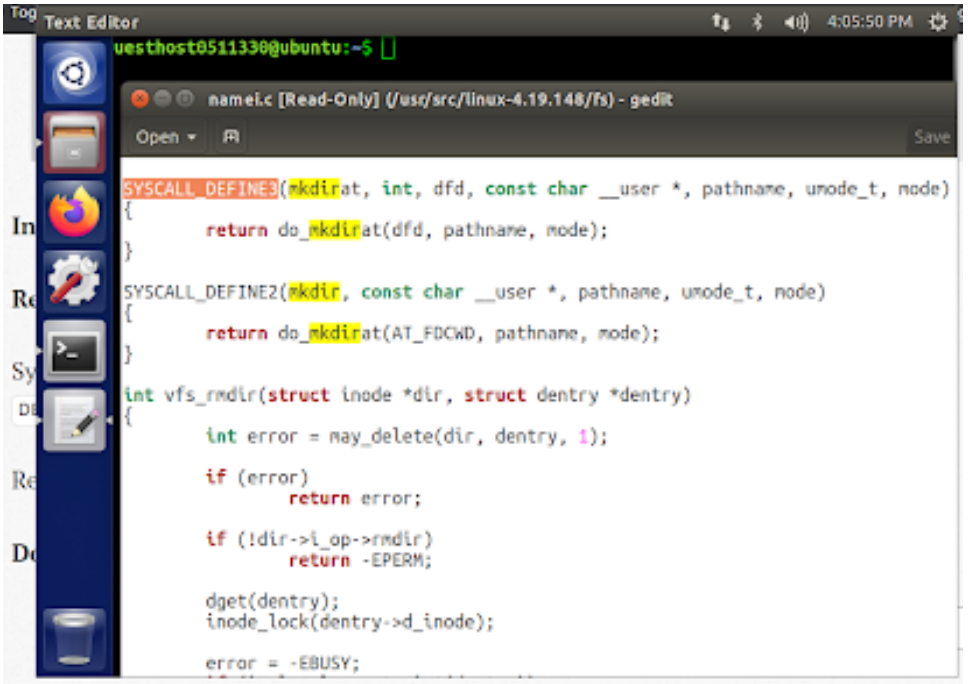
Step 7: FInd the sys.call number of Kernel Function .

(i) mkdir

1. Find the sys.call ”**mkdir**” in the file “ **syscall_64.tbl** ”
   (Path: /usr/src/linux-4.19.148/arch/x86/entry/syscalls/syscall_64.tbl)



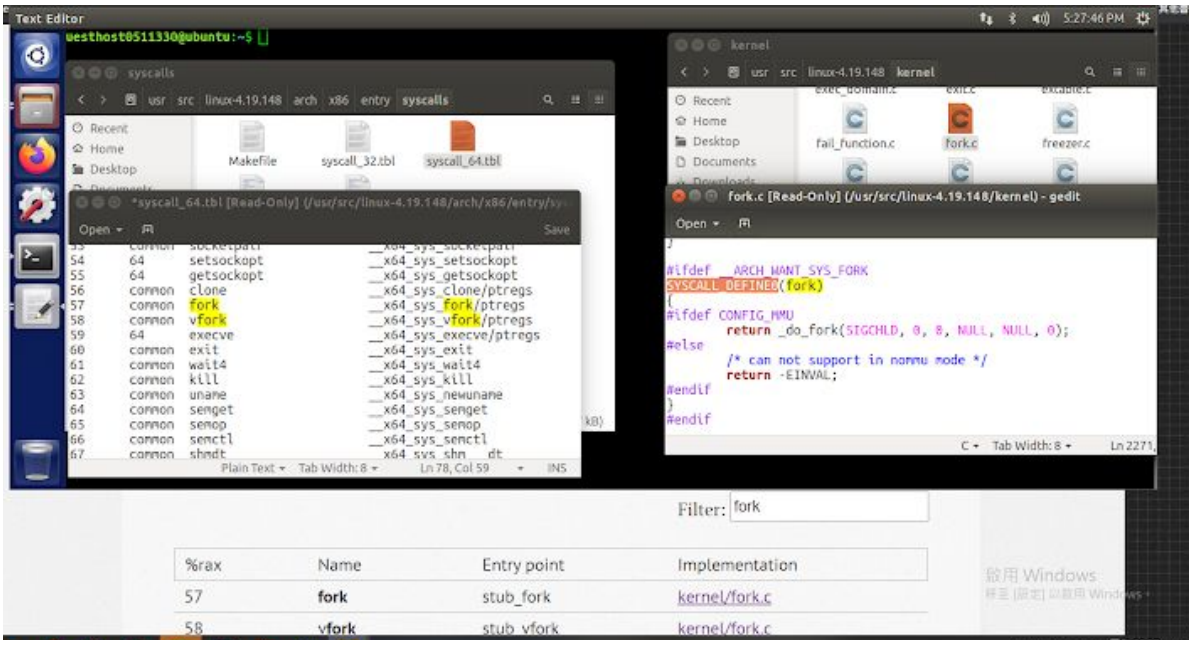
2. Find the Kernel Function implemented in the file “**namei.c**”.
   (Path: /usr/src/linux-4.19.148/fs/namei.c)



Also, we can find fork & getpid implemented files.

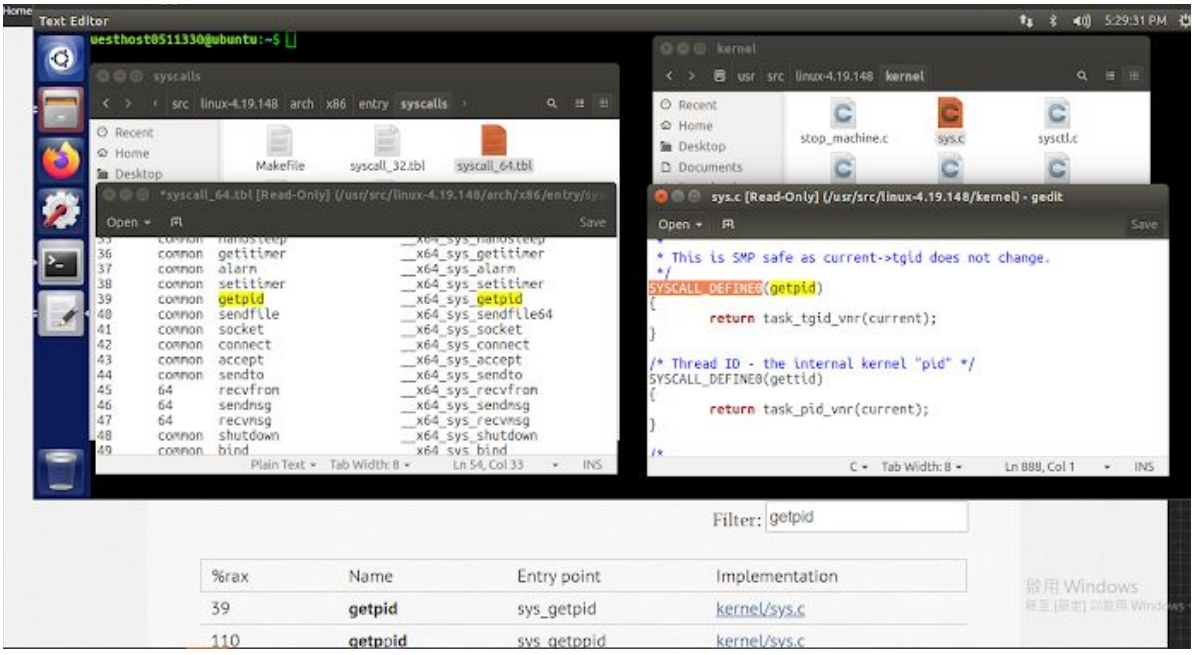
[Screenshot 3]

(ii) fork (parameter: SIGCHLD)
implemented file: fork.c
path: **/usr/src/linux-4.19.148/kernel**



[Screenshot 4]

(iii) getpid (parameter: current)
implemented file: sys.c
path: **/usr/src/linux-4.19.148/kernel**



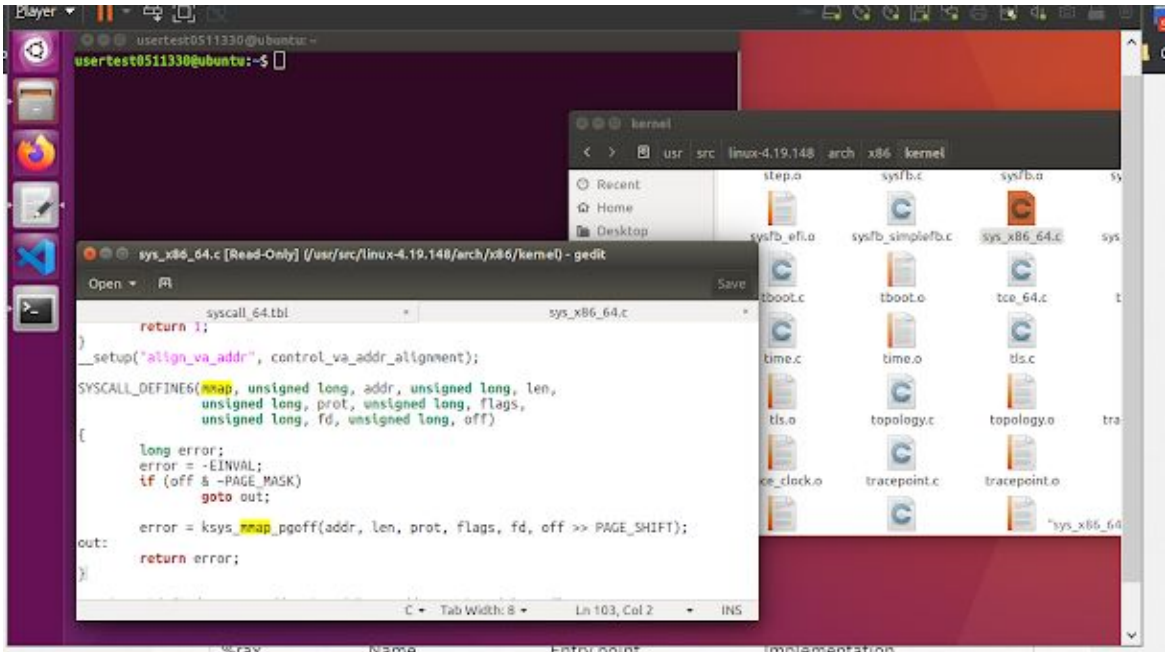
OS: fork() & getpid() !! Our ~~best friends~~ in textbook example XD

[Screenshot 5]

(iii) mmap

implemented file: sys_x86_64.c

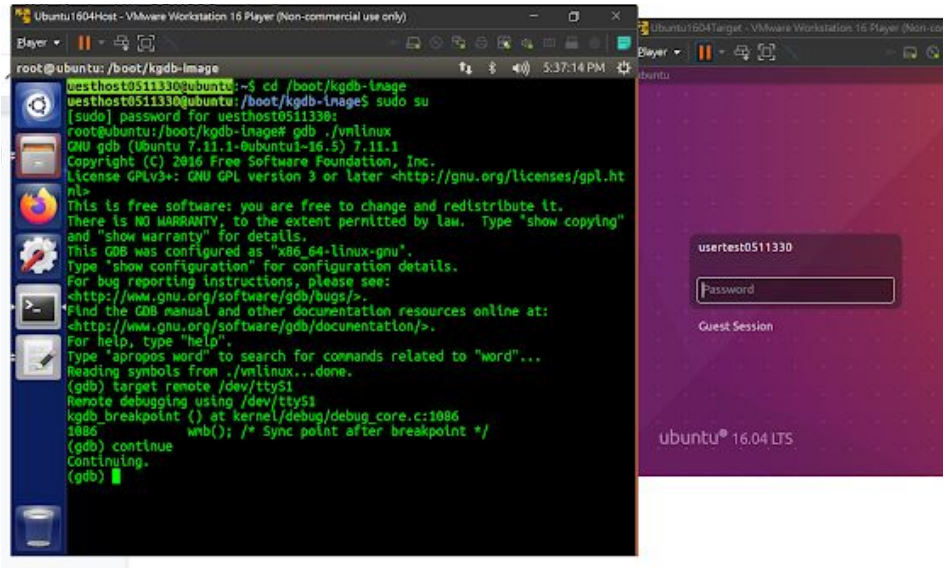
path: **/usr/src/linux-4.19.148/arch/x86/kernel**



[Screenshot 7] Connect GDB to the target

Using command: **#gdb ./vmlinux** and **target remote /dev/ttyS1**

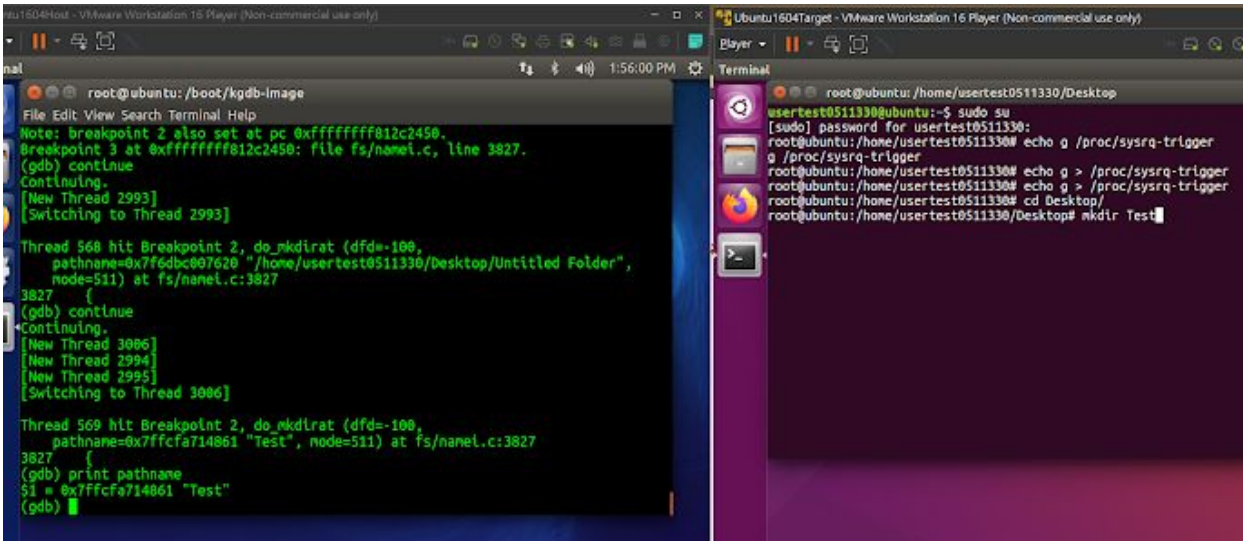
We will connect the target machine to debug it.



[Screenshot 8] Print the parameter of the function

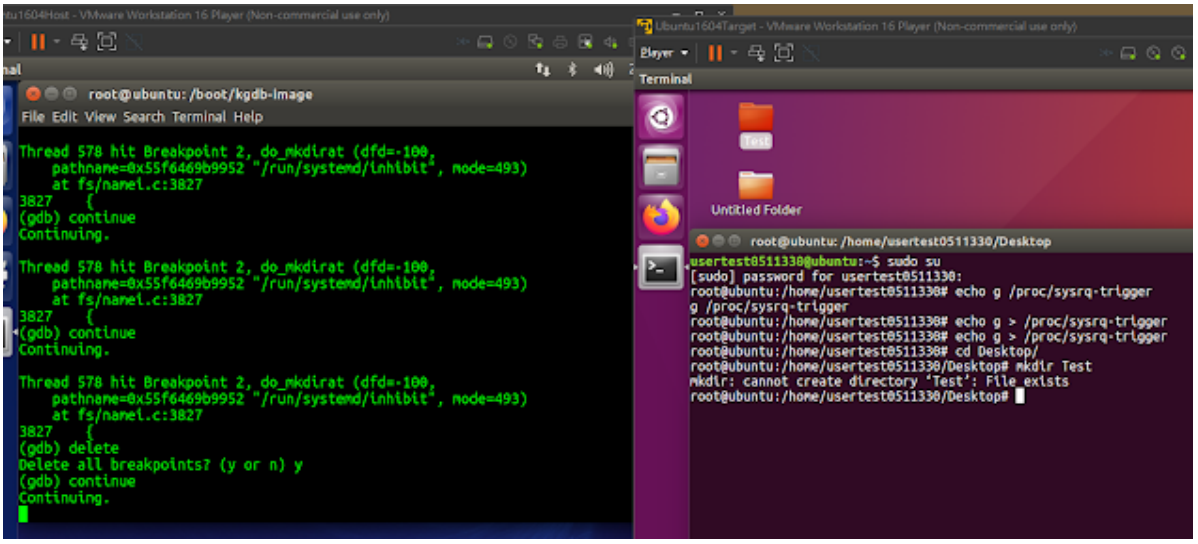
Using the command:**(gdb) print pathname**

Print the pathname, and it shows “Test” on the terminal, which is the folder we create in the target.



[Screenshot 9] Target get control back

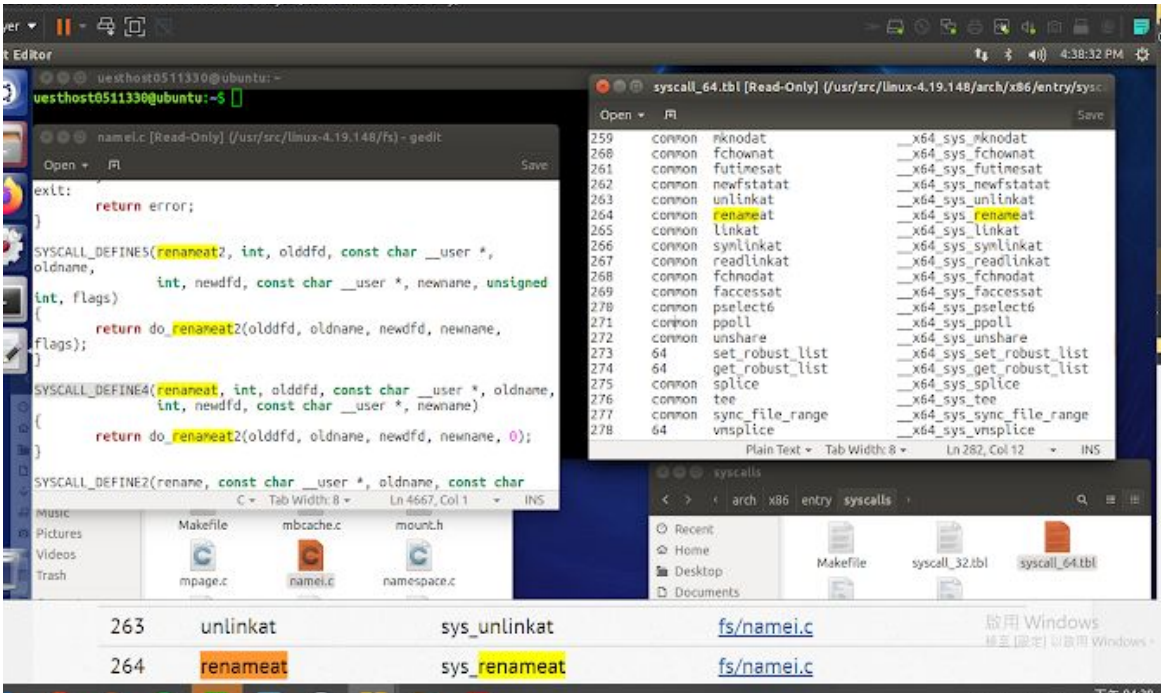
Using the command:**(gdb) continue** to resume the target system.



[Screenshot 10] Select “rename” function

a. I pick up the “**rename”** function.

b. it is implemented in the file “**namei.c**”,
(path: **/usr/src/linux-4.19.148/fs**)

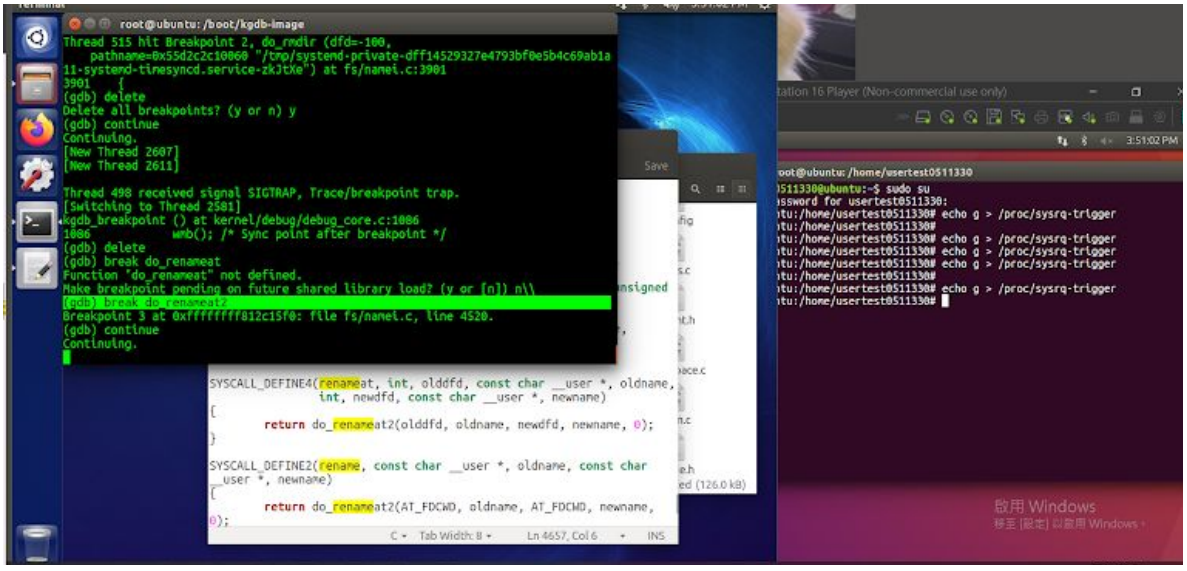


[Screenshot 11] Set the breakpoint

c. kernel function: do_renameat2

d. parameter: (olddfd, oldname newdfd, newname, flags)

using command: **break do_renameat2** to set the breakpoint to it.

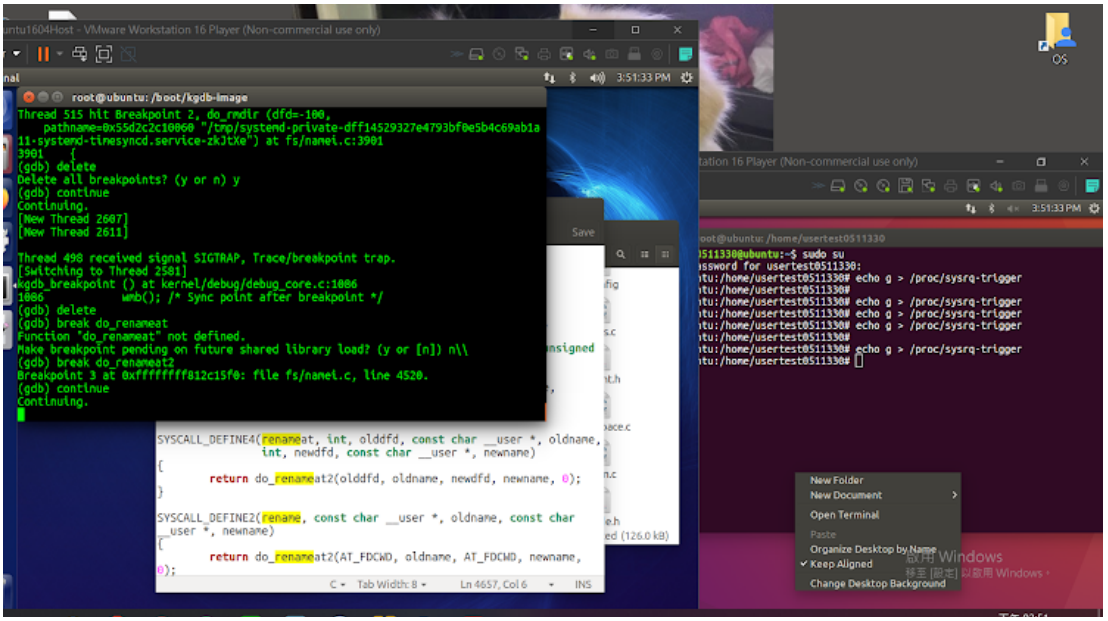


e. trigger-event - (1)create a new folder and (2)delete it.

(1) create a new folder

[Screenshot 12] Machine is still responsive

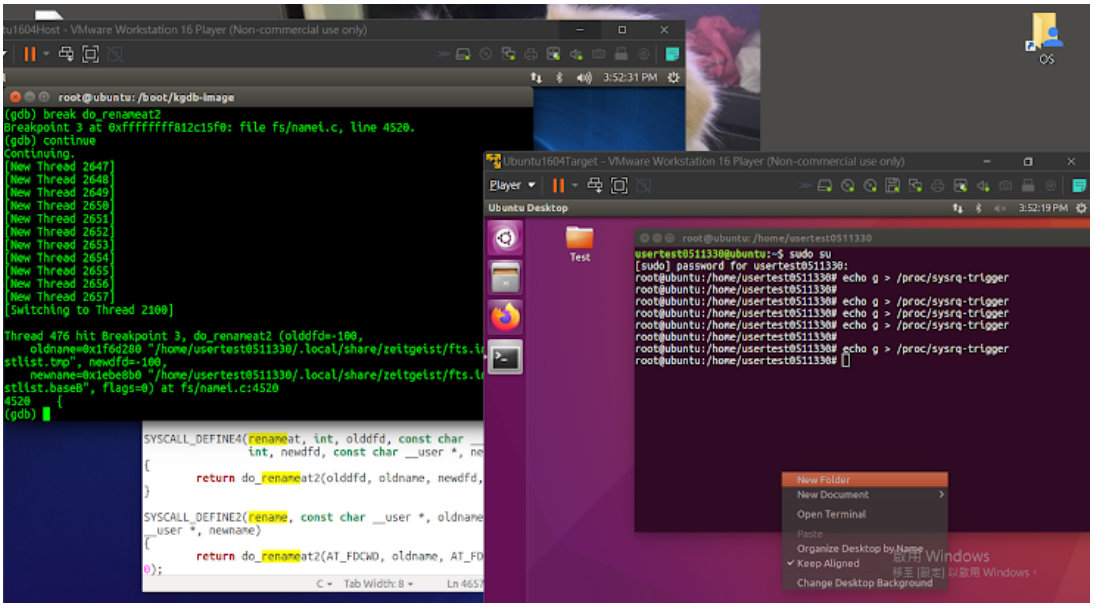
Right click on the target's desktop and the system is **still alive.**



(The clock showing in both machines were consistent)

[Screenshot 13] trigger-event

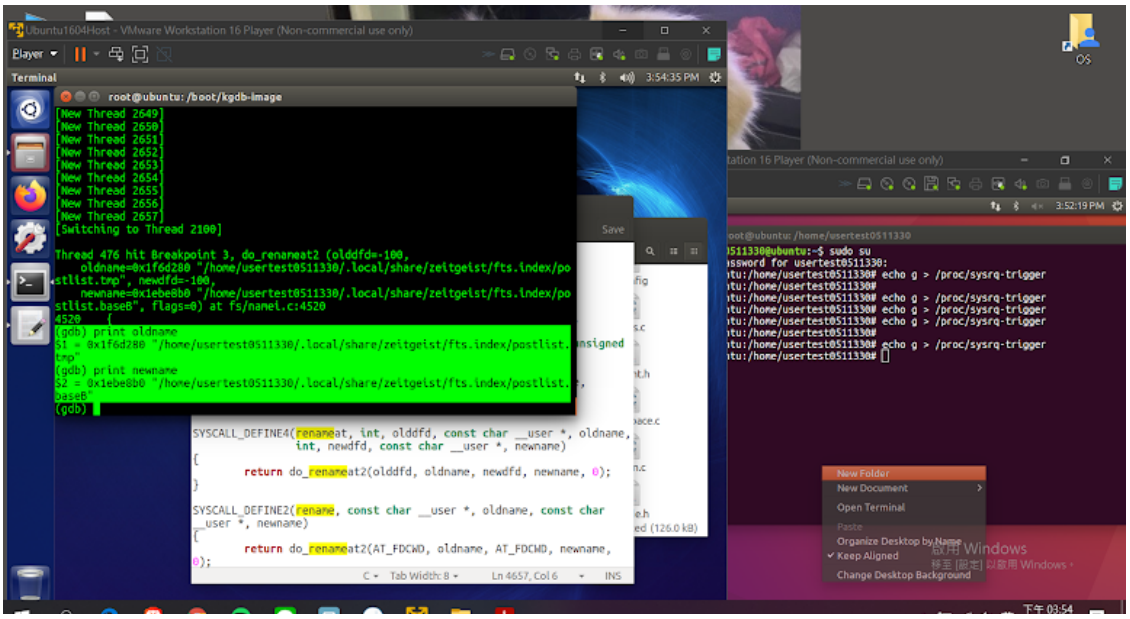
However, when we create a new folder, it gets freezed immediately.



(The target's clock was stopped!)

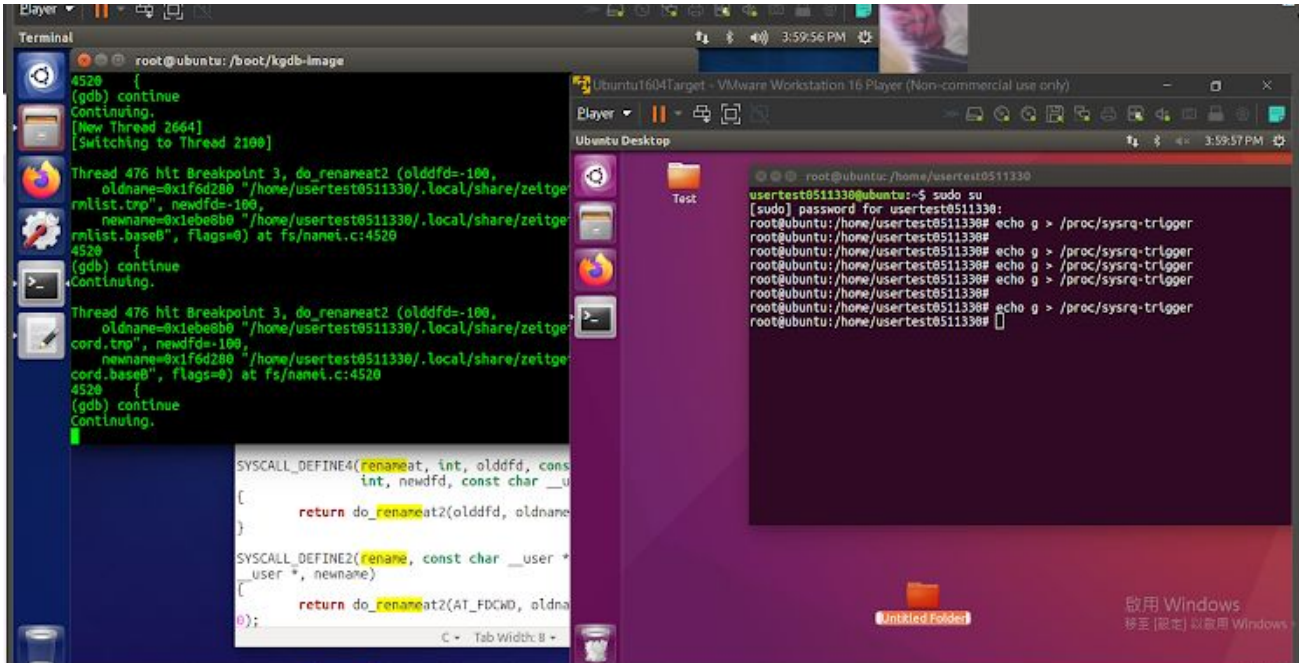
[Screenshot 14] print the parameters of function

Using gdb in the host to print the parameter - “oldname” & “newname”



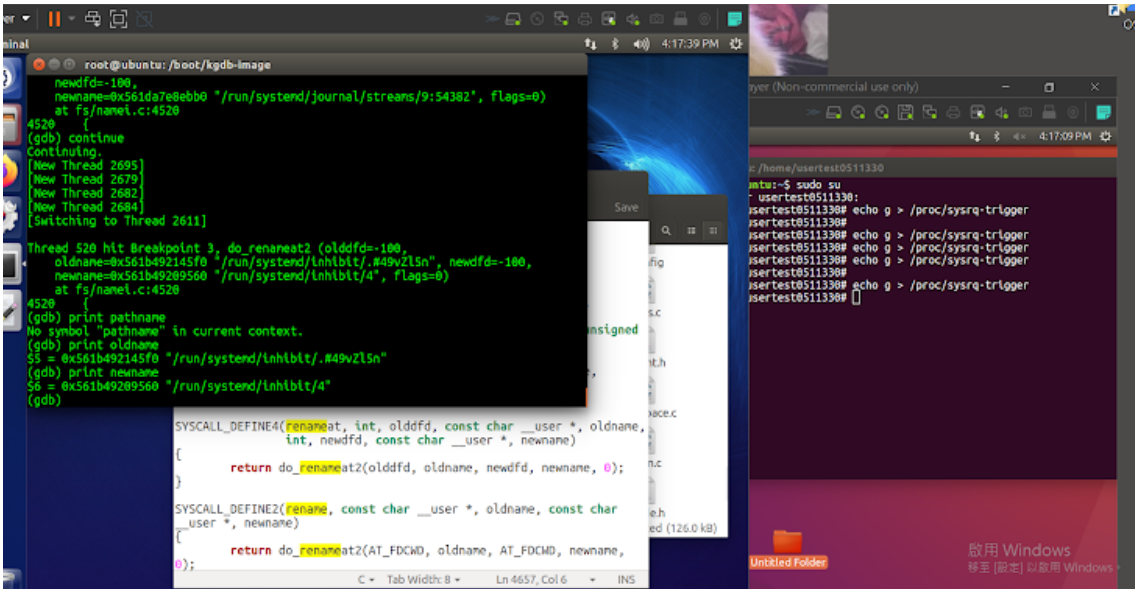
[Screenshot 15] System resumes.

With a series of "continue” commands input, the target finally resumes again. It seems that when we create a new folder, the system will do a lot of tasks rather than we just seen on the desktop.

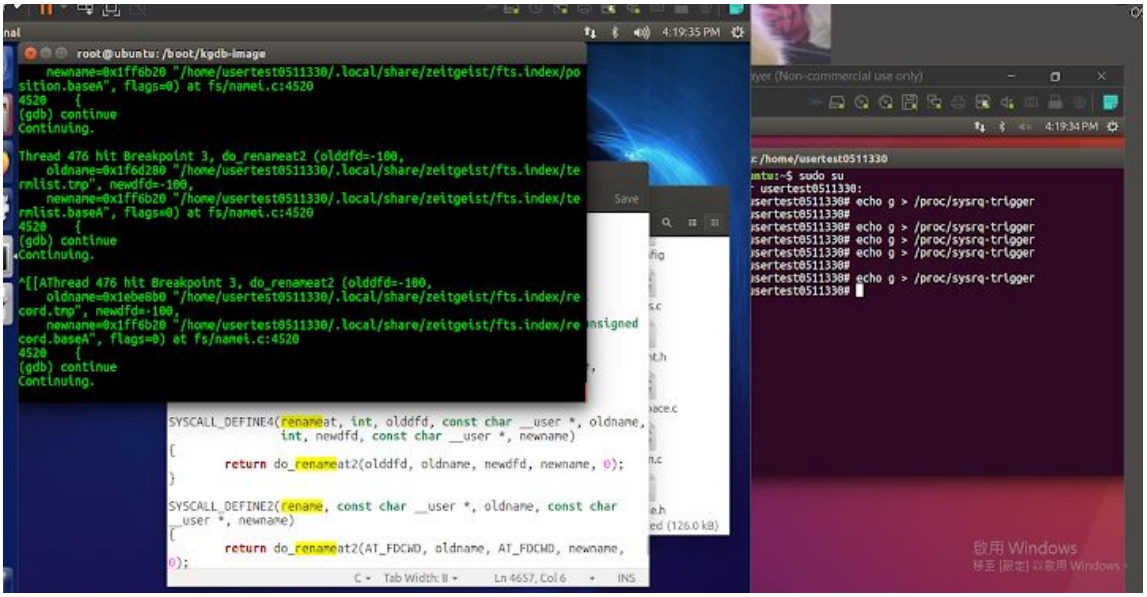


(2)delete folder

then I decide to delete the folder just created, the target freezes again. Parameters “oldname” and “newname” as shown in the figure.

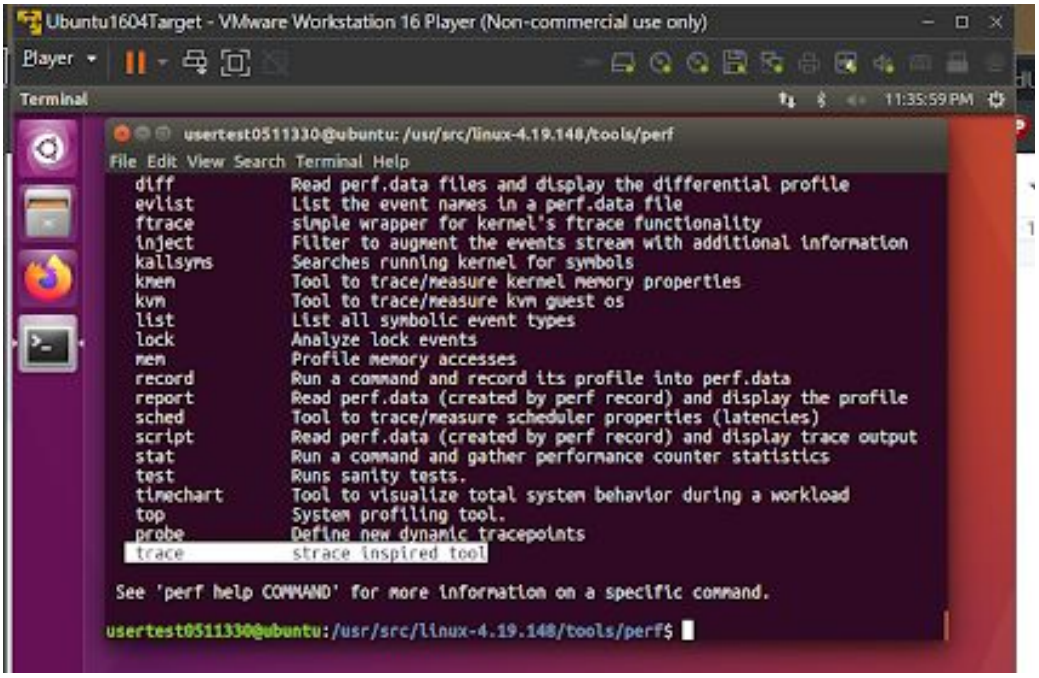


After many times of “continue”commands input on the host, the folder on the target desktop finally gets deleted .



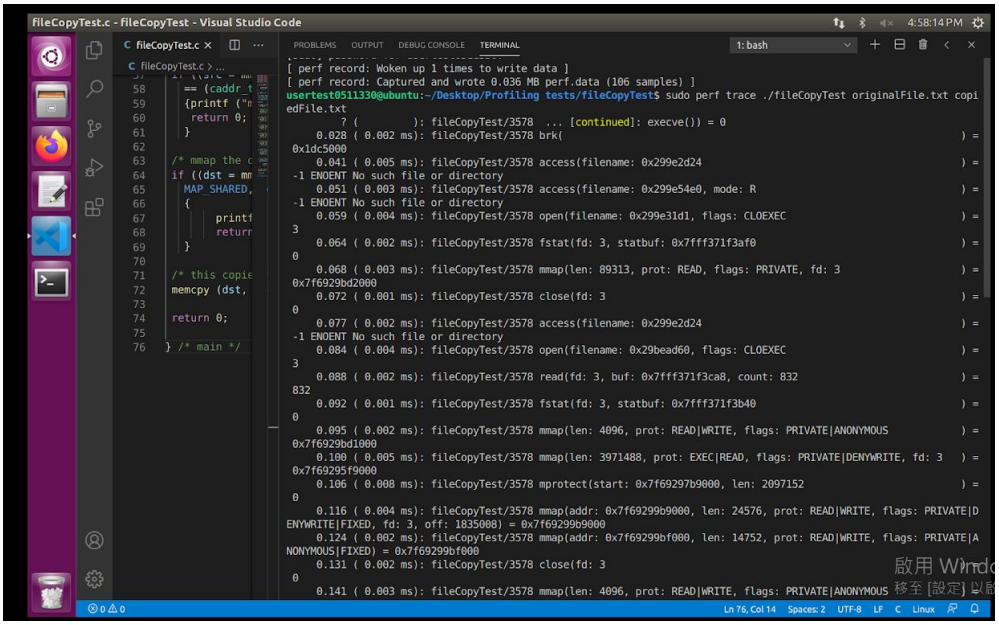
[Screenshot 16] Installing profiling tool

using **$sudo make** & **$sudo make install** to install perf tool under the /tools/perf/ directory.



[Screenshot 17] fileCopyTest results

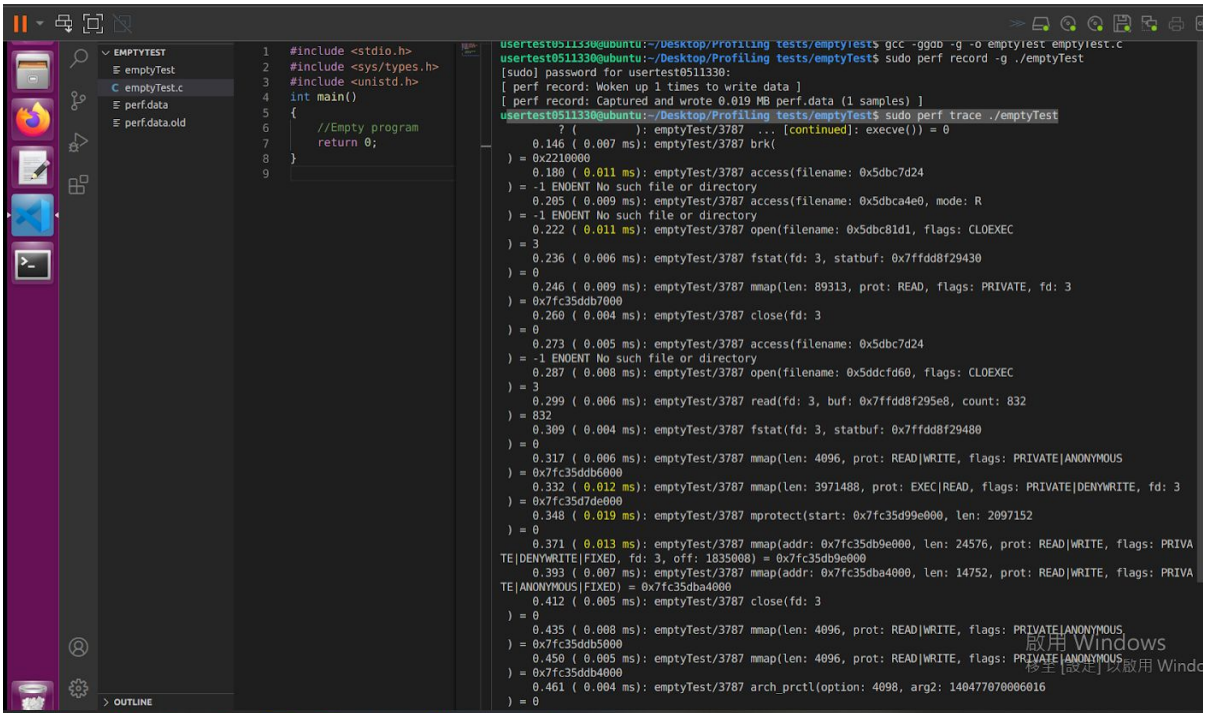
We trace the perfdata of fileCoptyTest and it shows some useful information for us to realize what syscalls has been called.



[Screenshot 18] emptyTest results

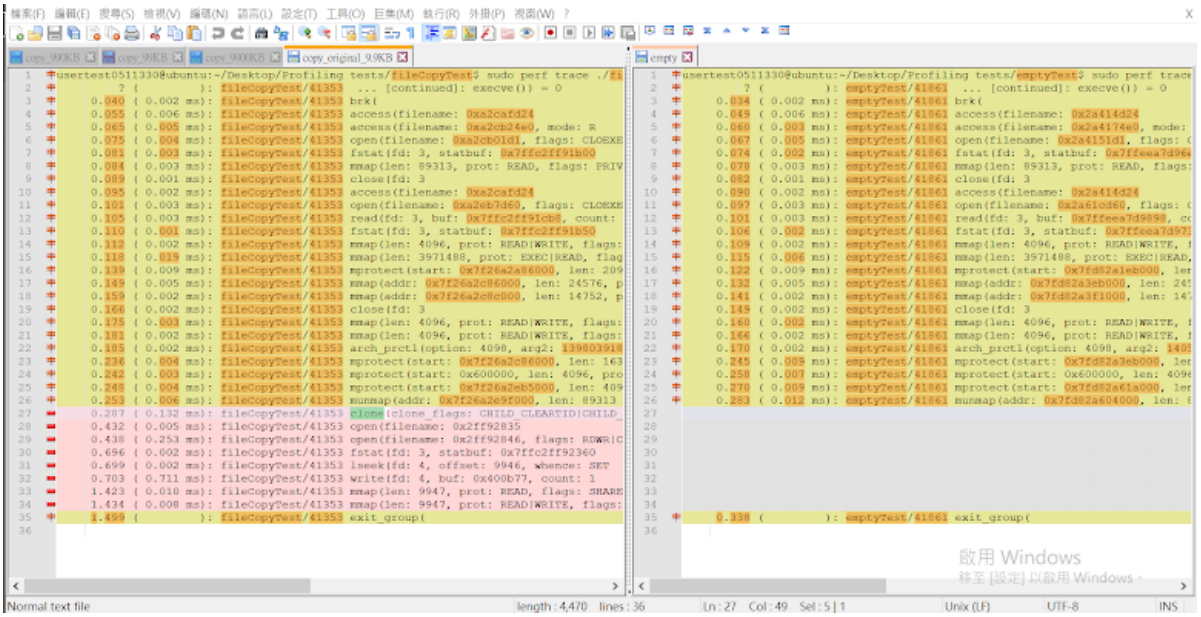
Using the command:**$sudo perf trace ./emptyTest**

We trace the perfdata of emptyTest, and it shows some useful information for us to realize what syscalls has been called.



[Screenshot 19] Comparing trace records - empty vs. copy

We compare two results above by Notepad++. There's an additional syscall in the copyFileTest. We will discuss it in the Q&A part.



[Quetions]

1. What is a kernel function? What is a system call?
   (i) Kernel function: A function who is a member in the group system call. When we invoke a kernel function, the OS jumps to kernel mode and does some tasks for requests from the user program.

   (ii) System call Provide the means for the OS on the user program to perform tasks reserved for the OS on the user program's behalf. A system call usually takes the form of a trap to a specific location in the interrupt vector. For example, When we create a process, we probably will invoke the **fork** system call (UNIX, LINUX OS).

2. What is KASLR? What is it for?
   KASKR (**K**ernel **A**ddress **S**pace **L**ayout **R**andomization) is a mechanism that will put the kernel code at random locations, to prevent malicious attacks. For example, the **unprivileged** instruction **SITD** can be used to get the location of kernel code called by a user program. However, for our convenience, we tend to disable it in order to debug the kernel.

3. What are GDB's non-stop and all-stop modes?
   (i) For some multi-threaded targets, a mode of operation in which you can examine stopped program threads in the debugger while **other threads continue to execute freely**. This is referred to as **non-stop mode.**

   (ii) In contrast to stop mode, whenever your program stops under GDB for any reason, ***all* threads of execution stop**, not just the current thread.This is referred to as **stop mode**. This allows you to examine the overall state of the program, including switching between threads, without worrying that things may change underfoot.

4. Explain what the command **echo g > /proc/sysrq-trigger** does.
   This command is used to send a signal to the host, in order to take control back in GDB.

5. (i) What are these functions: **clone**, **mmap**, **write** and **open**?

(1) **clone** is a system call whose functionality is similar to **fork**, clone will create a new process or thread.

(2) **mmap** stands for memory-mapped files. It's a way to **read** and **write** files which is **faster** than the traditional way. The file happens to be mapped into the virtual address and then we can revise the file directly in memory, instead of invoking additional syscalls that cause overhead for entering kernel mode.

(3) **write** will write data form a buffer declared by the user to a given device.

(4) A program initializes access to a file using the **open** syscall, and opens the files specified by pathname.

(ii) Why is there no fork system call? What is the difference between fork and clone?

(1) **fork** makes a full copy of the parent's memory. Copying the whole memory was pretty expensive since a **fork** was probably followed by an **exec** to do other tasks. (Nowadays **fork** don't copy the memory)

(2) **clone** is the syscall used by **fork**. It creates a new process or a thread, allowing the child process to share parts of its information (e.g. Memory space, files descriptors etc.).

The difference between them is just which data structures are shared or not.

(iii) Will the functions' execution time be longer if the file is bigger?

(1) Only the **open** will execute longer when the file size becomes bigger.

(2) But I found some interesting results. When file size is big enough, the times that function was called are double, The result is shown in Fig(b) & (c) below. So actually all functions will be longer if the file size is bigger enough.

(iv) <u>Create a graph of file size vs. execution times of the four functions. How is the behavior of each function?</u>

(1) Shown as below.
(2) Sorting from fastest to slowest: **mmap**, **write**, **clone**, **open**

Figure (a) - Syscall Execution Time (ms) vs. File Size (KB)

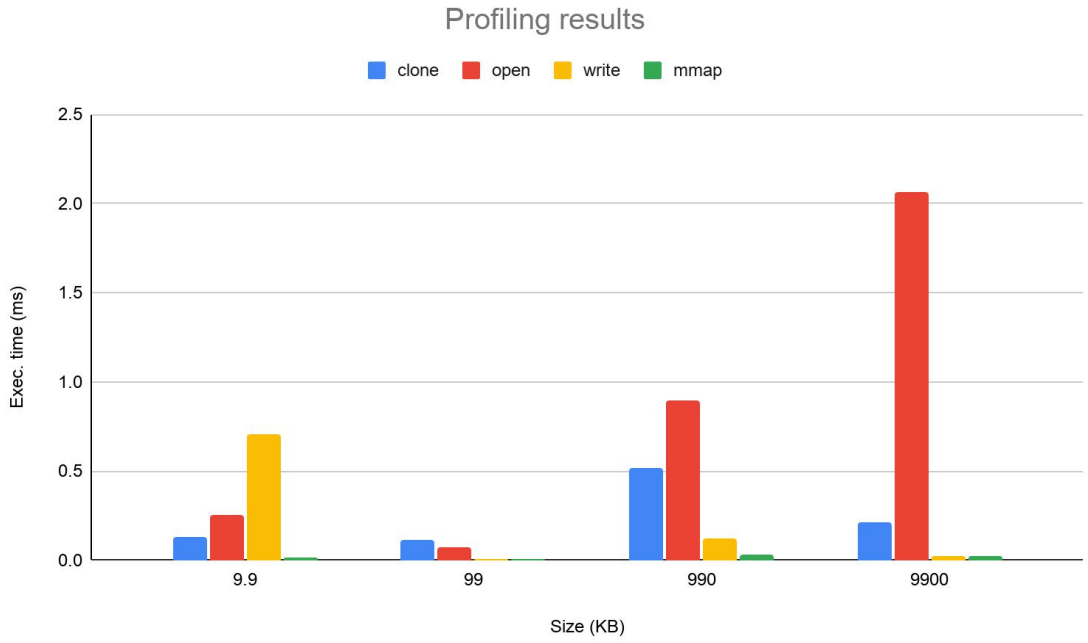
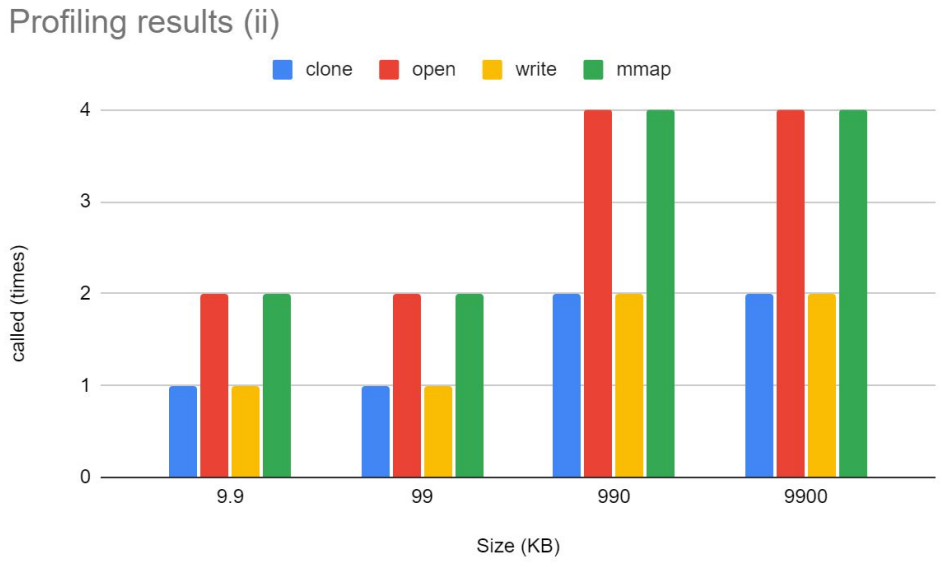
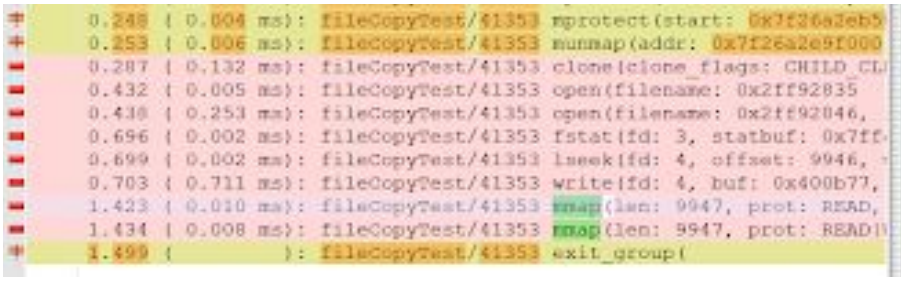


Figure (b) - times that function was called (times) vs. File Size(KB)

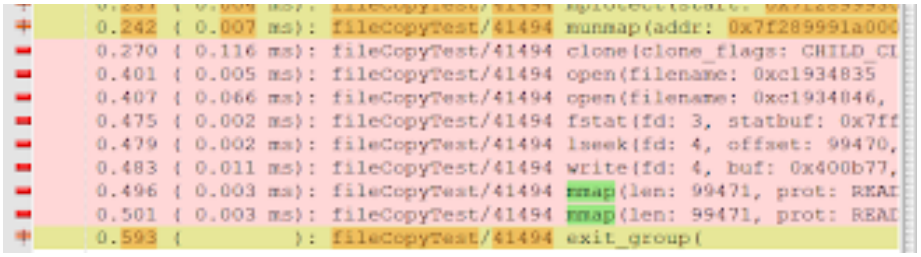


Fig(c) - mmap function calling times

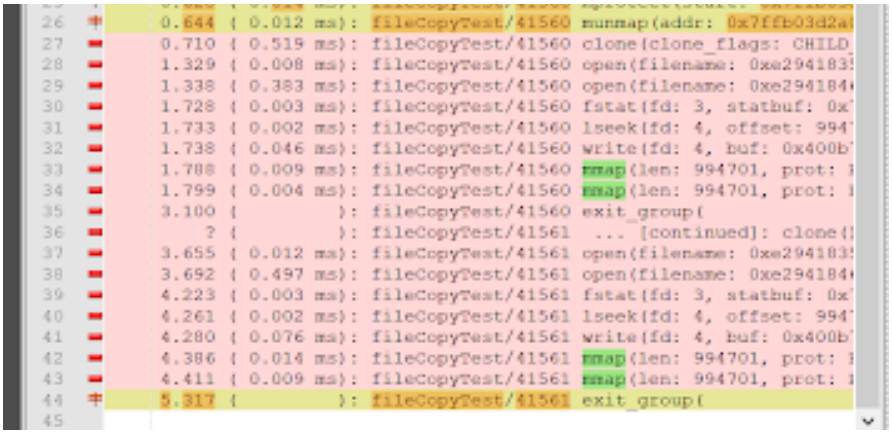
9.9KB (2 times)



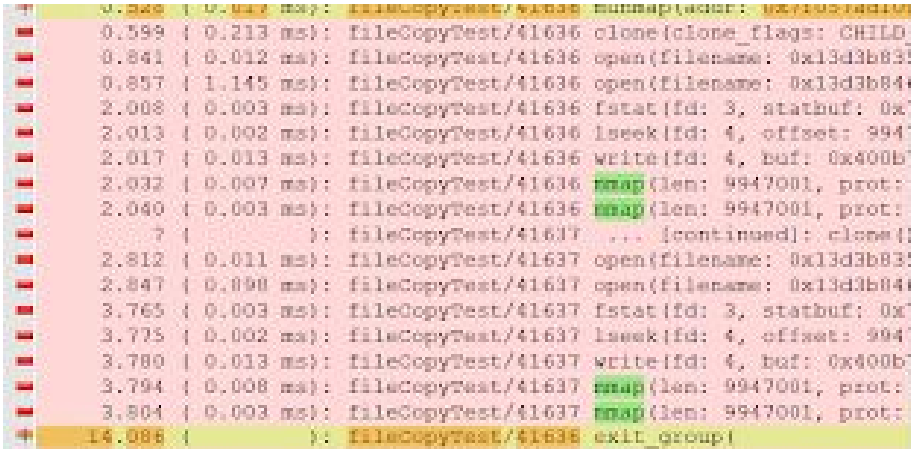
99KB (2 times)



990KB (4 times)



9900KB (4 times)

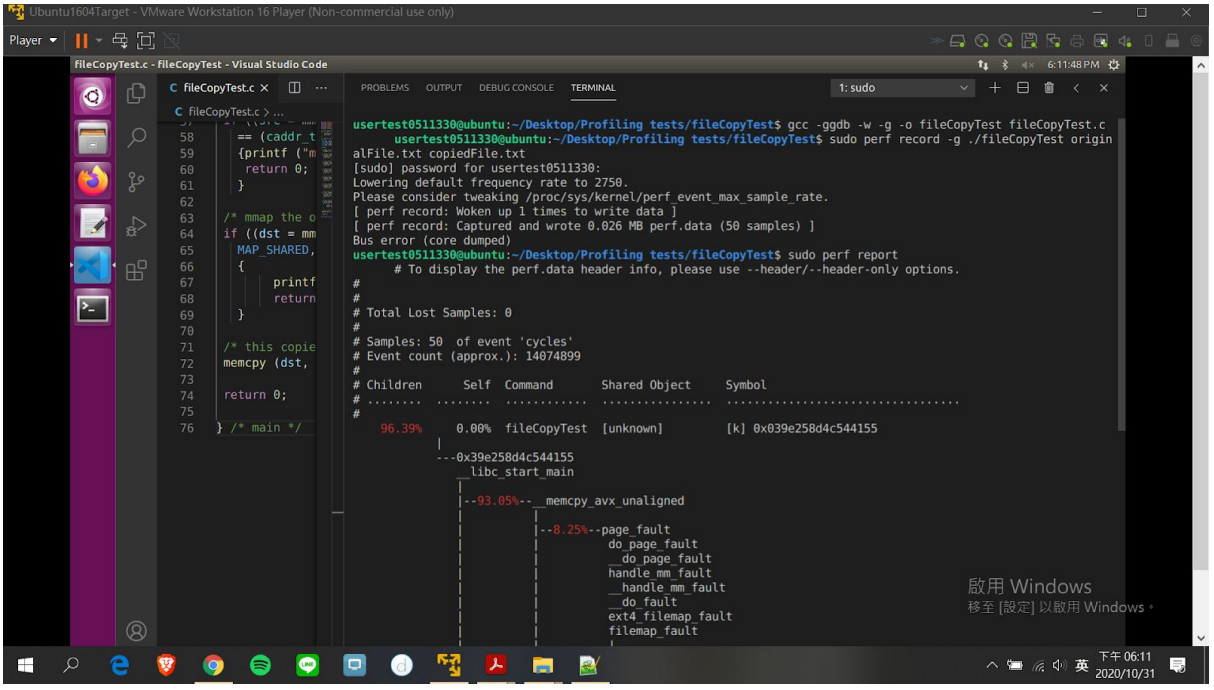


6. Select another function from the stscall_64.tbl table, look for where it is implemented and create your own scenario to trigger it.
   Shown in ScreenShot10 ~ 15 above.

7. (a) What does **$sudo perf report** mean for?
   It can show the statistic record by graph representation(e.g. tree structure, nested event structure), instead of listing merely all the text information. Programmers can easily figure out where the bottleneck is, and focus on the performance-improving part effortlessly.

   (b) For fileCopyTest, show and interpret the results.
   We found that the part **memcpy_avx_unaligned** takes 92.7% event counts. I guess function **memcpy** is the bottleneck of this program. In function **memcpy,** 51.42% **page_fault** occurs. So It should be an **IO-bound** process .

[Screenshot 21] Perf report



8. Perf has more commands. Select another command and explain it.

[Screenshot 22] $perf stat

We usually **$perf stat ./fileCopyTest** to clarify which part is the bottleneck of our program. Some useful information is provided.
For example, High branch-misses impies high the miss rate of branch prediction which will reduce CPI and performance. So we can verify which part of code should improve to increase our performance (loop, if-else part).