

# 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.

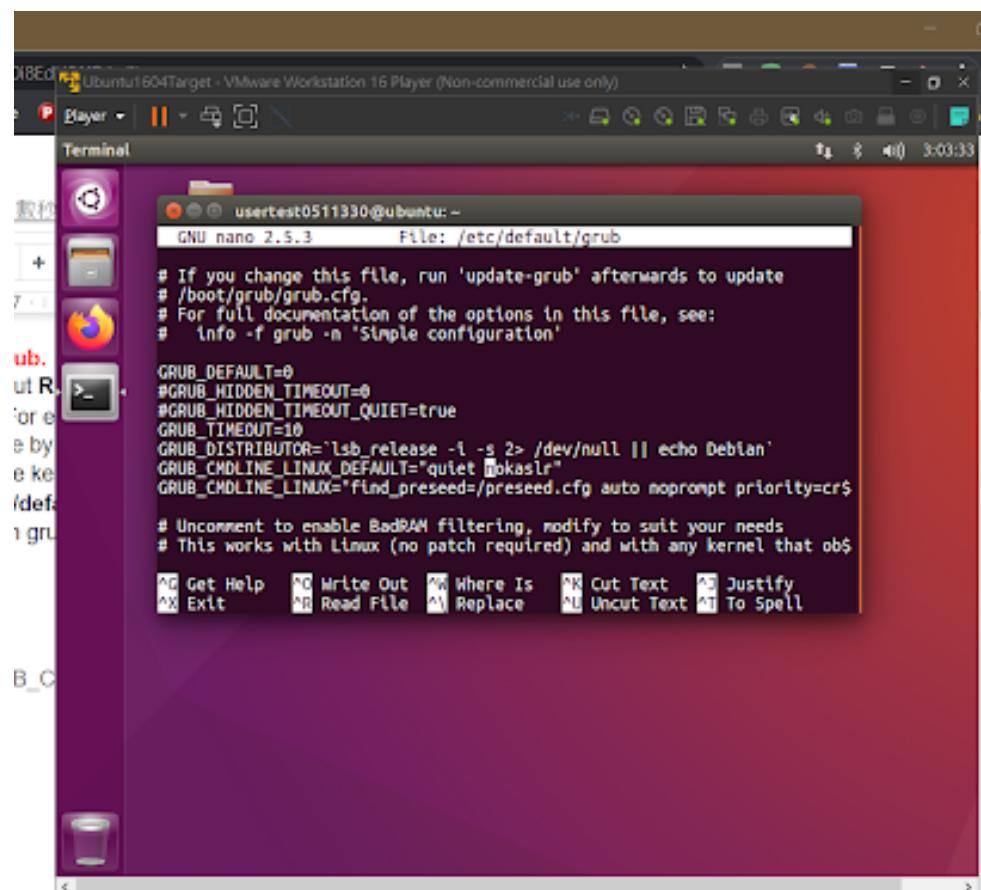
We will learn to set up a **working environment**, **trigger breakpoints** and profile and interpret the results

## [Section 1-1]

### [Screenshot 1] Disable KASLR from grub.

KASLR (Kernel Address Space Layout Randomization) 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`)

The screenshot shows a terminal window titled "Ubuntu 16.04 Target - VMware Workstation Player (Non-commercial use only)". The user is running the command `sudo nano /etc/default/grub`, entering their password. Then, they run `sudo update-grub`. The output shows the system generating the configuration file and finding various Linux images and initrd files. Finally, they run `nano /boot/grub/grub.cfg` and `sudo nano /boot/grub/grub.cfg` again, followed by `sudo nano /etc/default/grub` and entering their password again.

```
user@user:~$ sudo nano /etc/default/grub
[sudo] password for user:
user@user:~$ sudo update-grub
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.19.148
Found initrd image: /boot/initrd.img-4.19.148
Found linux image: /boot/vmlinuz-4.15.0-112-generic
Found initrd image: /boot/initrd.img-4.15.0-112-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
user@user:~$ nano /boot/grub/grub.cfg
user@user:~$ sudo nano /boot/grub/grub.cfg
user@user:~$ sudo nano /etc/default/grub
[sudo] password for user:
user@user:~$
```

The screenshot shows the `GNU nano 2.5.3` editor displaying the `/boot/grub/grub.cfg` file. The file contains GRUB configuration code, including boot loaders for multiple kernels and a specific entry for the `kgdbwait kgdboc=ttyS1,115200` command.

```
recordfail
load_video
gfxmode $linux_gfx_mode
insmod gzio
if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
insmod part_msdos
insmod ext2
set root='hd0,msdos1'
if [ x$feature_platform_search_hint = xy ]; then
    search --no-floppy --fs-uuid --set=root --hint-bios=hd0,msdos1 --hi...
else
    search --no-floppy --fs-uuid --set=root 0aaae1c5-ab10-49e7-8099-79d$...
fi
$static locale=en_US quiet nokaslr kgdbwait kgdboc=ttyS1,115200
initrd /boot/initrd.img-4.19.148
}
```

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

### (i) mkdir

- 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)

The terminal window shows the command `sudo wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.19.148.tar.xz` being run, followed by the contents of the `syscall_64.tbl` file in a text editor. The `mkdir` entry is highlighted in red.

%rax	Name	Entry point	Implementation
83	<b>mkdir</b>	sys_mkdir	<a href="#">fs/namei.c</a>
258	<b>mkdirat</b>	sys_mkdirat	<a href="#">fs/namei.c</a>

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

The terminal window shows the command `cat /usr/src/linux-4.19.148/fs/namei.c` being run, displaying the C code for the `mkdir` and `mkdirat` system calls.

```
SYSCALL_DEFINE3(mkdirat, int, dfd, const char __user *, pathname, umode_t, mode)
{
    return do_mkdirat(dfd, pathname, mode);
}

SYSCALL_DEFINE2(mkdir, const char __user *, pathname, umode_t, mode)
{
    return do_mkdirat(AT_FDCWD, pathname, mode);
}

int vfs_rmdir(struct inode *dir, struct dentry *dentry)
{
    int error = may_delete(dir, dentry, 1);

    if (error)
        return error;

    if (!dir->i_op->rmdir)
        return -EPERM;

    dget(dentry);
    inode_lock(dentry->d_inode);

    error = -EBUSY;
```

%rax	Name	Entry point	Implementation
83	<b>mkdir</b>	sys_mkdir	<a href="#">fs/namei.c</a>

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

Text Editor  
westhost0511330@ubuntu:~\$

Recent Home Desktop Makefile syscall\_32.tbl syscall\_64.tbl

\*syscall\_64.tbl [Read-Only] (/usr/src/linux-4.19.148/arch/x86/entry/syscalls)  
Open Save  
53 common socketpair \_\_x64\_sys\_socketpair  
54 64 setsockopt \_\_x64\_sys\_setssockopt  
55 64 getsockopt \_\_x64\_sys\_getsockopt  
56 common clone \_\_x64\_sys\_clone/pregs  
57 common fork \_\_x64\_sys\_fork/pregs  
58 common vfork \_\_x64\_sys\_vfork/pregs  
59 64 execve \_\_x64\_sys\_execve/pregs  
60 common exit \_\_x64\_sys\_exit  
61 common wait4 \_\_x64\_sys\_wait4  
62 common kill \_\_x64\_sys\_kill  
63 common uname \_\_x64\_sys\_newuname  
64 common senget \_\_x64\_sys\_senget  
65 common senop \_\_x64\_sys\_senop  
66 common senctl \_\_x64\_sys\_senctl  
67 common shndt \_\_x64\_sys\_shndt  
Plain Text Tab Width: 8 Ln 78, Col 59 INS

kernel  
Recent Home Desktop fail\_function.c fork.c freezer.c  
Documents Downloads fork.c  
fork.c [Read-Only] (/usr/src/linux-4.19.148/kernel) - gedit  
Open Save  
#ifdef \_ARCH\_WANT\_SYS\_FORK  
SYSCALL\_DEFINE(fork)  
{  
#ifdef CONFIG\_MMU  
return \_\_do\_fork(SIGCHLD, 0, 0, NULL, 0);  
#else  
/\* can not support in mmu mode \*/  
return -EINVAL;  
#endif  
}  
#endif  
C Tab Width: 8 Ln 227, Filter: fork

%rax	Name	Entry point	Implementation
57	fork	stub_fork	kernel/fork.c
58	vfork	stub_vfork	kernel/fork.c

## [Screenshot 4]

(iii) getpid (parameter: current)

implemented file: sys.c

path: /usr/src/linux-4.19.148/kernel

Text Editor  
westhost0511330@ubuntu:~\$

Recent Home Desktop Makefile syscall\_32.tbl syscall\_64.tbl

\*syscall\_64.tbl [Read-Only] (/usr/src/linux-4.19.148/arch/x86/entry/syscalls)  
Open Save  
53 common nanosleep \_\_x64\_sys\_nanosleep  
54 common gettimer \_\_x64\_sys\_gettime  
55 common alarm \_\_x64\_sys\_alarm  
56 common setitimer \_\_x64\_sys\_setitimer  
57 common getpid \_\_x64\_sys\_getpid  
58 common sendfile \_\_x64\_sys\_sendfile64  
59 common socket \_\_x64\_sys\_socket  
60 common connect \_\_x64\_sys\_connect  
61 common accept \_\_x64\_sys\_accept  
62 common sendto \_\_x64\_sys\_sendto  
63 common recvfrom \_\_x64\_sys\_recvfrom  
64 64 sendmsg \_\_x64\_sys\_sendmsg  
65 64 recvmsg \_\_x64\_sys\_recvmsg  
66 64 shutdown \_\_x64\_sys\_shutdown  
67 common bind \_\_x64\_sys\_bind  
Plain Text Tab Width: 8 Lh 54, Col 33 INS

kernel  
Recent Home Desktop stop\_machine.c sys.c sysctl.c  
Documents Downloads sys.c  
sys.c [Read-Only] (/usr/src/linux-4.19.148/kernel) - gedit  
Open Save  
\* This is SMP safe as current->tgid does not change.  
\*/  
SYSCALL\_DEFINEx(getpid)  
{  
 return task\_tgid\_vnr(current);  
}  
/\* Thread ID - the internal kernel "pid" \*/  
SYSCALL\_DEFINEx(getpid)  
{  
 return task\_pid\_vnr(current);  
}  
/\*  
C Tab Width: 8 Ln 888, Col 1 Filter: getpid

%rax	Name	Entry point	Implementation
39	getpid	sys_getpid	kernel/sys.c
110	getpid	sys_getpid	kernel/sys.c

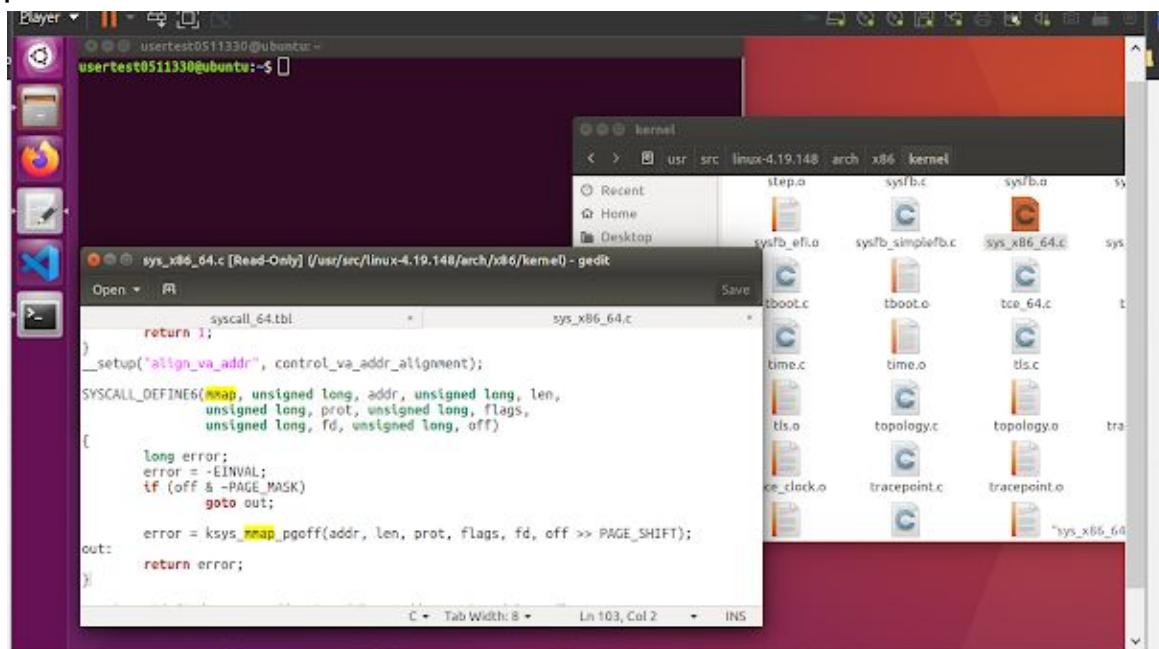
OS: fork() & getpid() !! Our best friends in textbook example XD

## [Screenshot 5]

### (iii) mmap

implemented file: sys\_x86\_64.c

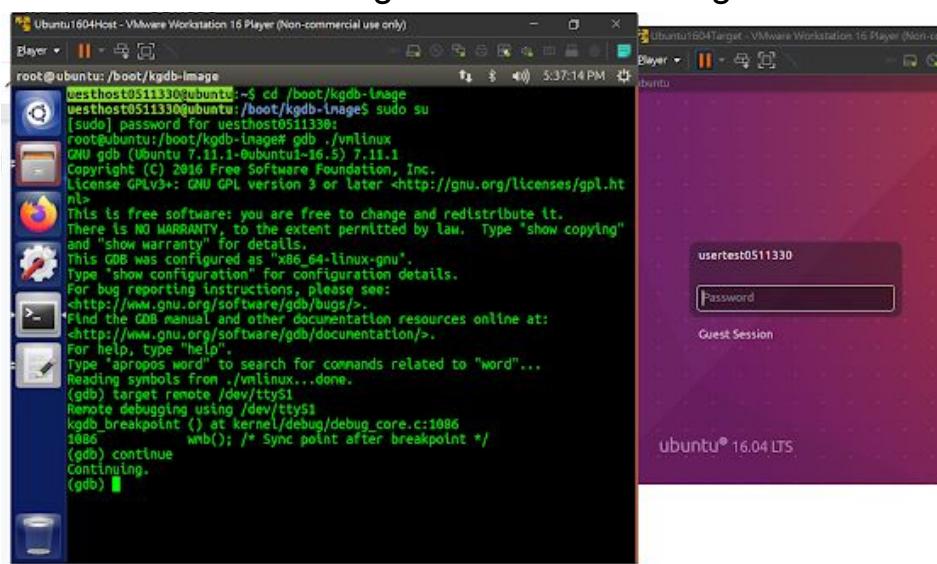
path: /usr/src/linux-4.19.148/arch/x86/kernel



[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.

The screenshot shows two terminal windows side-by-side. The left window is a GDB session attached to a kernel (kgdb-image). It displays assembly code and a stack trace, with the instruction at address 3827 being do\_mkdirat. The right window is a standard Linux terminal window for user test0511330. It shows the user running several commands: sudo su, echo g > /proc/sysrq-trigger, echo g > /proc/sysrq-trigger, echo g > /proc/sysrq-trigger, cd Desktop, and mkdir Test. The output of the mkdir command indicates that the directory 'Test' already exists.

```
root@ubuntu:/boot/kgdb-image
File Edit View Search Terminal Help
Note: breakpoint 2 also set at pc 0xffffffff812c2450.
Breakpoint 3 at 0xffffffff812c2450: file fs/namei.c, line 3827.
(gdb) continue
Continuing.
[New Thread 2993]
[Switching to Thread 2993]

Thread 568 hit Breakpoint 2, do_mkdirat (dfd=-100,
    pathname=0x7f6dbc007620 "/home/usertest0511330/Desktop/Untitled Folder",
    mode=511) at fs/namei.c:3827
3827 {
(gdb) continue
Continuing.
[New Thread 3006]
[New Thread 2994]
[New Thread 2995]
[Switching to Thread 3006]

Thread 568 hit Breakpoint 2, do_mkdirat (dfd=-100,
    pathname=0x7ffcfa714861 "Test", mode=511) at fs/namei.c:3827
3827 {
(gdb) print pathname
$1 = 0x7ffcfa714861 "Test"
(gdb) ■

root@ubuntu:/home/usertest0511330/Desktop
[usertest0511330@ubuntu:~]$ sudo su
[sudo] password for usertest0511330:
root@ubuntu:/home/usertest0511330# echo g > /proc/sysrq-trigger
g /proc/sysrq-trigger
root@ubuntu:/home/usertest0511330# echo g > /proc/sysrq-trigger
root@ubuntu:/home/usertest0511330# echo g > /proc/sysrq-trigger
root@ubuntu:/home/usertest0511330# cd Desktop/
root@ubuntu:/home/usertest0511330/Desktop# mkdir Test
root@ubuntu:/home/usertest0511330/Desktop# mkdir: cannot create directory 'Test': File exists
root@ubuntu:/home/usertest0511330/Desktop# ■
```

## [Screenshot 9] Target get control back

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

The screenshot shows two terminal windows side-by-side. The left window is a GDB session attached to a kernel (kgdb-image). It displays assembly code and a stack trace, with the instruction at address 3827 being do\_mkdirat. The right window is a standard Linux terminal window for user test0511330. It shows the user running several commands: sudo su, echo g > /proc/sysrq-trigger, echo g > /proc/sysrq-trigger, echo g > /proc/sysrq-trigger, cd Desktop, and mkdir Test. The output of the mkdir command indicates that the directory 'Test' already exists. In this screenshot, the GDB session has reached a point where it is prompting the user to delete breakpoints, indicating that the target system has been successfully controlled back by the debugger.

```
root@ubuntu:/boot/kgdb-image
File Edit View Search Terminal Help
Thread 578 hit Breakpoint 2, do_mkdirat (dfd=-100,
    pathname=0x55f6469b9952 "/run/systemd/inhibit", mode=493)
    at fs/namei.c:3827
3827 {
(gdb) continue
Continuing.

Thread 578 hit Breakpoint 2, do_mkdirat (dfd=-100,
    pathname=0x55f6469b9952 "/run/systemd/inhibit", mode=493)
    at fs/namei.c:3827
3827 {
(gdb) continue
Continuing.

Thread 578 hit Breakpoint 2, do_mkdirat (dfd=-100,
    pathname=0x55f6469b9952 "/run/systemd/inhibit", mode=493)
    at fs/namei.c:3827
3827 {
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) continue
Continuing.

root@ubuntu:/home/usertest0511330/Desktop
[usertest0511330@ubuntu:~]$ sudo su
[sudo] password for usertest0511330:
root@ubuntu:/home/usertest0511330# echo g > /proc/sysrq-trigger
g /proc/sysrq-trigger
root@ubuntu:/home/usertest0511330# echo g > /proc/sysrq-trigger
root@ubuntu:/home/usertest0511330# echo g > /proc/sysrq-trigger
root@ubuntu:/home/usertest0511330# cd Desktop/
root@ubuntu:/home/usertest0511330/Desktop# mkdir Test
root@ubuntu:/home/usertest0511330/Desktop# mkdir: cannot create directory 'Test': File exists
root@ubuntu:/home/usertest0511330/Desktop# ■
```

## [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)

The screenshot shows a dual-pane code editor. The left pane displays the file `/usr/src/linux-4.19.148/fs/namei.c`, which contains C code for the `rename` function. The right pane displays the file `/usr/src/linux-4.19.148/arch/x86/entry/syscalls/syscall_64.tbl`, which is a table of system call entries. In the right pane, the entry for `sys_renameat` is highlighted. Both panes have tabs for "Plain Text" and "INS". The bottom status bar indicates "Ln 282, Col 12".

## [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.

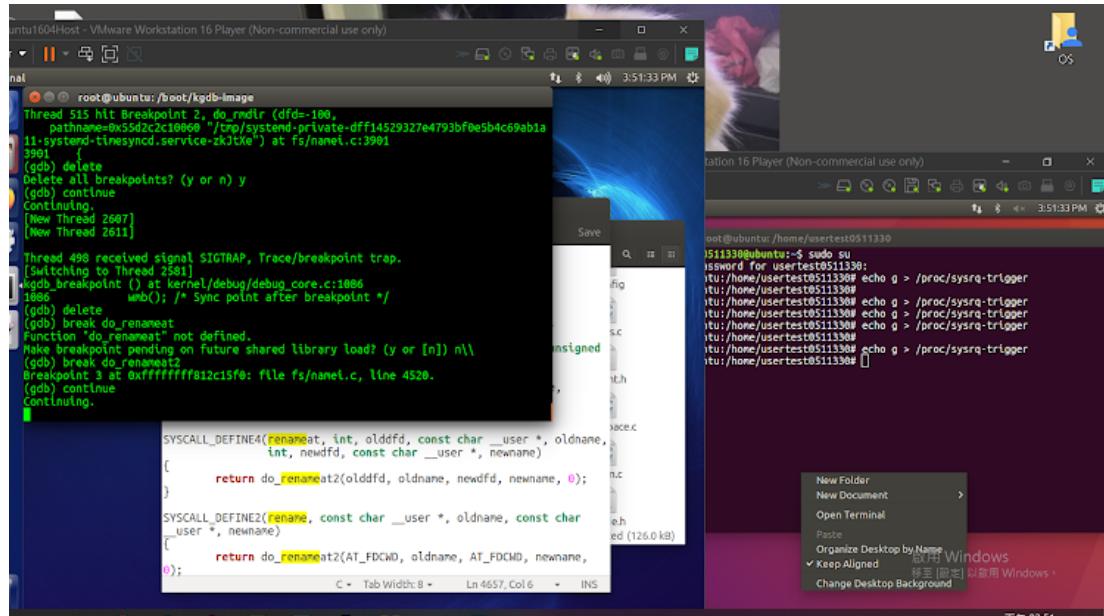
The screenshot shows a terminal window running under GDB. The command `break do_renameat2` is entered, and the response shows that the breakpoint is pending. The background shows a file browser with the file `namei.c` selected. The terminal window also shows other kernel logs and commands like `sudo su`.

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

### (1) create a new folder

#### [Screenshot 12] Machine is still responsive

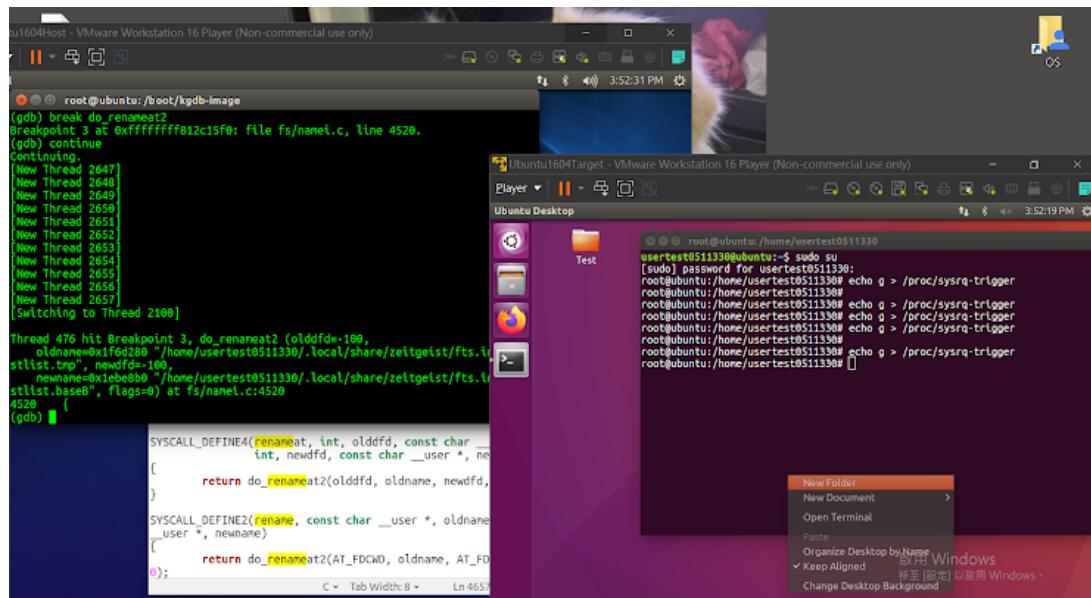
Right click on the target's desktop and the system is **still alive**.



(The clock showing in both machines were consistent)

#### [Screenshot 13] trigger-event

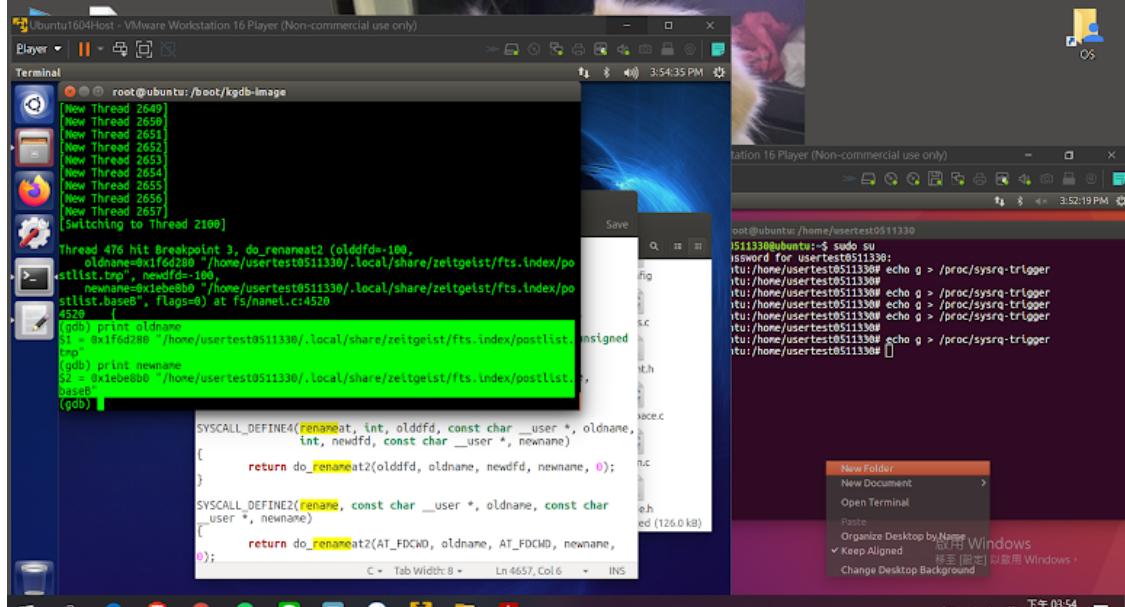
However, when we create a new folder, it gets freezed immediately.



(The target's clock was stopped!)

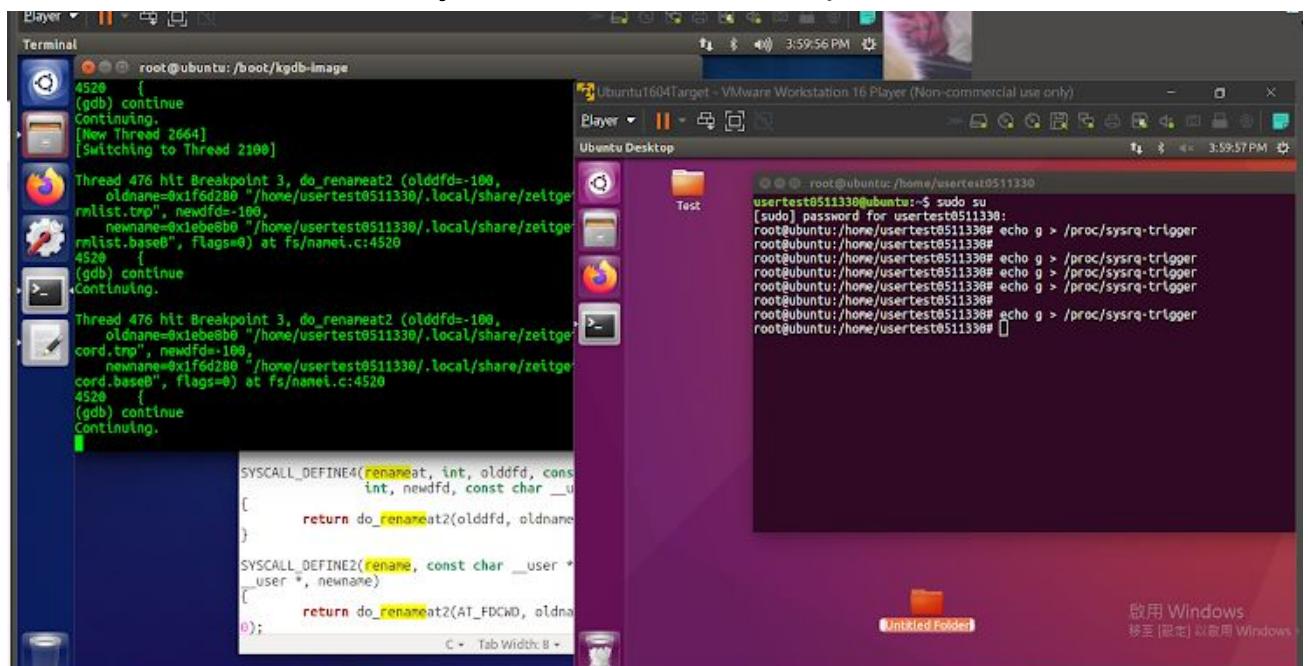
## [Screenshot 14] print the parameters of function

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



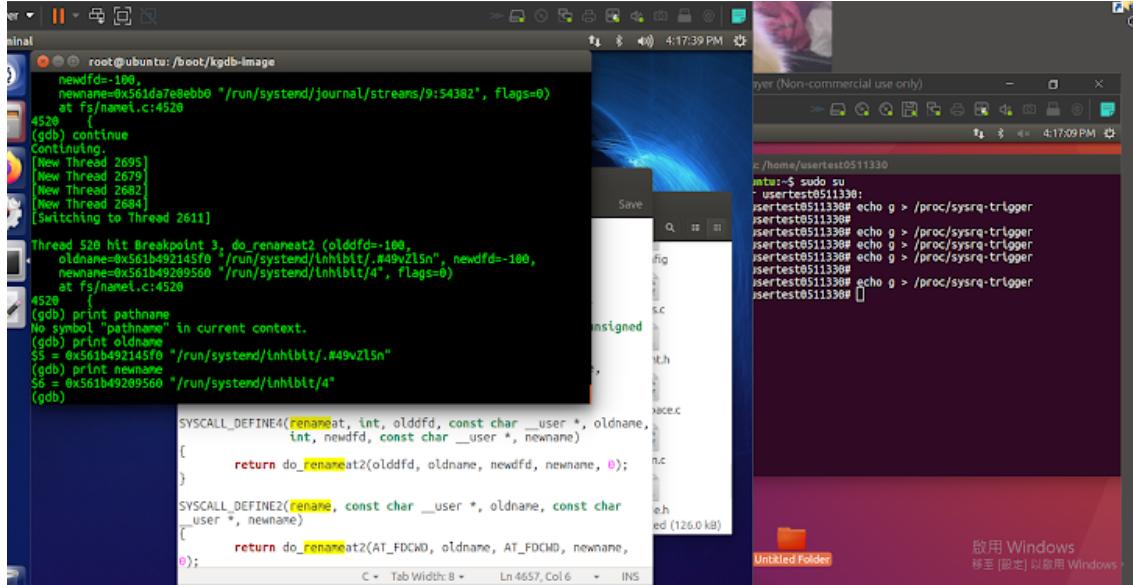
## [Screenshot 15] System resumes.

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.

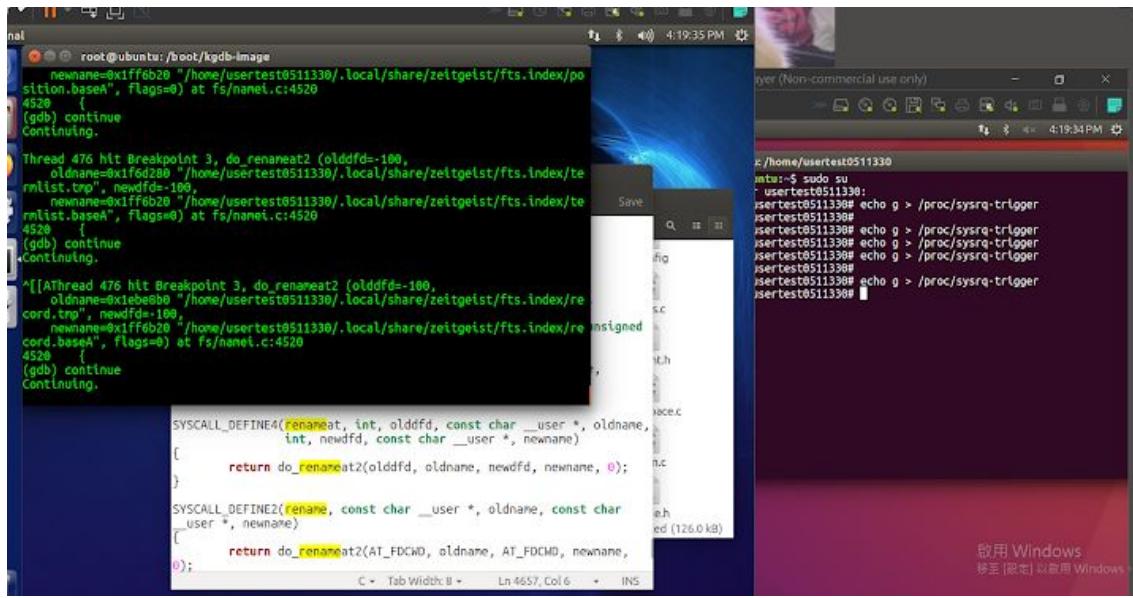


## (2)delete folder

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

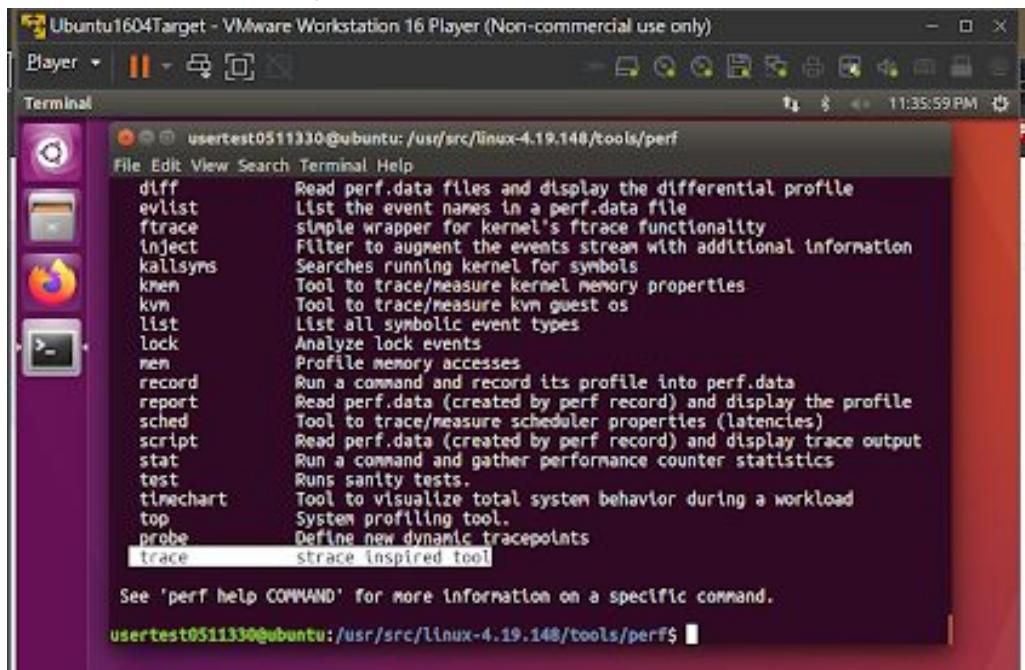


After many times of “continue” commands input on the host, the folder on the target desktop finally gets deleted .



## [Screenshot 16] Installing profiling tool

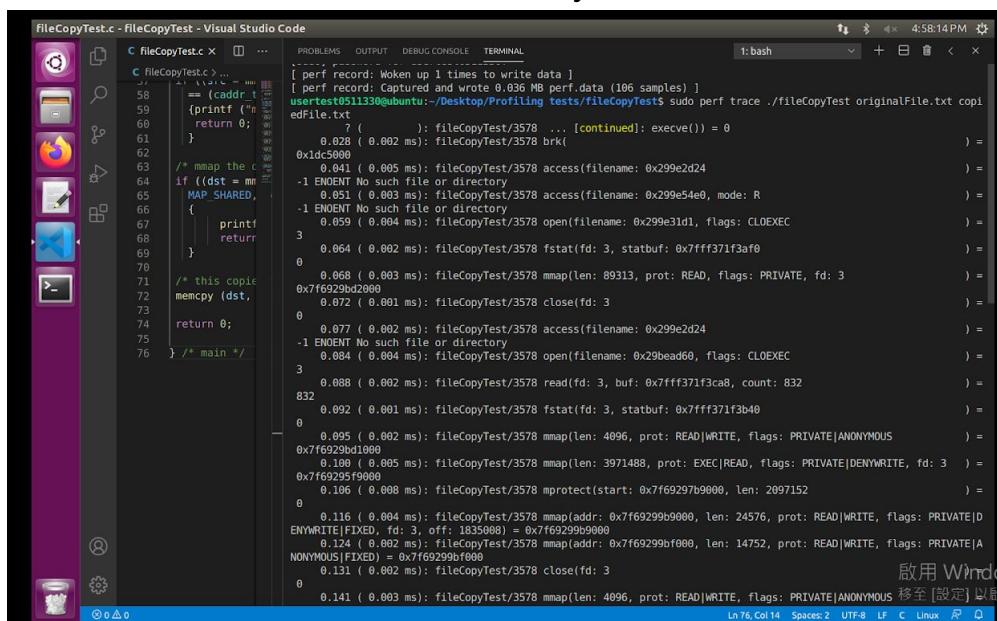
using `$sudo make` & `$sudo make install` to install perf tool under the `/tools/perf/` directory.



The screenshot shows a terminal window titled "Ubuntu1604Target - VMware Workstation 16 Player (Non-commercial use only)". The user is at the prompt `user0511330@ubuntu:/usr/src/linux-4.19.148/tools/perf$`. The terminal displays the help menu for the perf tool, listing various commands such as dlf, evlist, ftrace, inject, kallsyms, kvm, list, lock, men, record, report, sched, script, stat, test, timechart, top, probe, and trace. Each command is followed by a brief description. At the bottom of the help menu, it says "See 'perf help COMMAND' for more information on a specific command".

## [Screenshot 17] fileCopyTest results

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



The screenshot shows a Visual Studio Code interface with a terminal tab open. The terminal shows the output of a perf trace session for the fileCopyTest program. The output lists various system calls and their times, such as fileCopyTest/3578 mmap, access, and write operations. The terminal window title is "FileCopyTest.c - FileCopyTest - Visual Studio Code".

## [Screenshot 18] emptyTest results

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

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

```
user@test0511330:~/Desktop/Profiling tests/emptyTest$ gcc -ggdb -g -o emptyTest emptyTest.c
user@test0511330:~/Desktop/Profiling tests/emptyTest$ sudo perf record -g ./emptyTest
[perf] password for user@test0511330:
[perf record] Woken up 1 times to write data
[perf record] Captured and wrote 0.019 MB perf.data (1 samples)
user@test0511330:~/Desktop/Profiling tests/emptyTest$ sudo perf trace ./emptyTest
    ?
    0.146 ( 0.007 ms): emptyTest/3787 [continued]: execve() = 0
    ) = 0x22109000
    0.180 ( 0.011 ms): emptyTest/3787 access(filename: 0x5dbc7d24
    ) = -1 ENOENT No such file or directory
    0.285 ( 0.009 ms): emptyTest/3787 access(filename: 0x5dbc4e0, mode: R
    ) = -1 ENOENT No such file or directory
    0.222 ( 0.011 ms): emptyTest/3787 open(filename: 0x5dbc81d1, flags: CLOEXEC
    ) = 3
    0.236 ( 0.006 ms): emptyTest/3787 fstat(fd: 3, statbuf: 0x7ffffd8f29430
    ) = 0
    0.246 ( 0.009 ms): emptyTest/3787 mmap(len: 89313, prot: READ, flags: PRIVATE, fd: 3
    ) = 0x7fc35dd6b7000
    0.260 ( 0.004 ms): emptyTest/3787 close(fd: 3
    ) = 0
    0.273 ( 0.005 ms): emptyTest/3787 access(filename: 0x5dbc7d24
    ) = -1 ENOENT No such file or directory
    0.287 ( 0.008 ms): emptyTest/3787 open(filename: 0x5dbc81d1, flags: CLOEXEC
    ) = 3
    0.299 ( 0.006 ms): emptyTest/3787 mmap(len: 89313, prot: READ, flags: PRIVATE, fd: 3
    ) = 0
    0.309 ( 0.004 ms): emptyTest/3787 read(fd: 3, buf: 0x7ffffd8f295e8, count: 832
    ) = 832
    0.317 ( 0.006 ms): emptyTest/3787 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|ANONYMOUS
    ) = 0x7fc35dd6b000
    0.332 ( 0.012 ms): emptyTest/3787 mmap(len: 3971488, prot: EXEC|READ, flags: PRIVATE|DENY|WRITE, fd: 3
    ) = 0x7fc35d7de000
    0.343 ( 0.019 ms): emptyTest/3787 mprotect(start: 0x7fc35d9e000, len: 2097152
    ) = 0
    0.371 ( 0.013 ms): emptyTest/3787 mmap(addr: 0x7fc35db9e000, len: 24576, prot: READ|WRITE, flags: PRIVATE|DENY|WRITE|FIXED, fd: 3, off: 1835008) = 0x7fc35db9e000
    0.393 ( 0.007 ms): emptyTest/3787 mmap(addr: 0x7fc35dba4000, len: 14752, prot: READ|WRITE, flags: PRIVATE|ANONYMOUS|FIXED) = 0x7fc35dba4000
    0.412 ( 0.005 ms): emptyTest/3787 close(fd: 3
    ) = 0
    0.435 ( 0.008 ms): emptyTest/3787 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|ANONYMOUS
    ) = 0x7fc35dd6b5000
    0.450 ( 0.005 ms): emptyTest/3787 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|ANONYMOUS
    ) = 0x7fc35dd6b4000
    0.461 ( 0.004 ms): emptyTest/3787 arch_prctl(option: 4098, arg2: 140477070006016
    ) = 0
```

## [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.

```
user@test0511330:~/Desktop/Profiling tests/fileCopyTest$ sudo perf trace ./fileCopyTest
    ?
    0.040 ( 0.002 ms): fileCopyTest/41353 ... [continued]: execve() = 0
    0.055 ( 0.006 ms): fileCopyTest/41353 brk(
    0.065 ( 0.005 ms): fileCopyTest/41353 access(filename: 0x2a2c0d24
    0.075 ( 0.003 ms): fileCopyTest/41353 access(filename: 0x2a2c2d4e, mode: R
    0.081 ( 0.003 ms): fileCopyTest/41353 open(fd: 3, statbuf: 0x7ffcc2ff19b0
    0.084 ( 0.003 ms): fileCopyTest/41353 mmap(len: 89313, prot: READ, flags: PRIVATE
    0.089 ( 0.002 ms): fileCopyTest/41353 close(fd: 3
    0.095 ( 0.002 ms): fileCopyTest/41353 access(filename: 0x2a2c0d24
    0.105 ( 0.003 ms): fileCopyTest/41353 open(fd: 3, statbuf: 0x7ffcc2ff19b0, flags: CLOEXEC
    0.105 ( 0.003 ms): fileCopyTest/41353 read(fd: 3, buf: 0x7ffcc2ff19b0, count: 1433
    0.110 ( 0.001 ms): fileCopyTest/41353 fstat(fd: 3, statbuf: 0x7ffcc2ff19b0
    0.112 ( 0.002 ms): fileCopyTest/41353 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|DENY|WRITE
    0.118 ( 0.019 ms): fileCopyTest/41353 mmap(len: 3971488, prot: EXEC|READ, flag
    0.120 ( 0.003 ms): fileCopyTest/41353 mprotect(start: 0x7f26a2a00000, len: 2097152
    0.129 ( 0.005 ms): fileCopyTest/41353 mmap(addr: 0x7f26a2c9e000, len: 24576, p
    0.149 ( 0.005 ms): fileCopyTest/41353 mmap(addr: 0x7f26a2c9e000, len: 14752, p
    0.159 ( 0.002 ms): fileCopyTest/41353 close(fd: 3
    0.166 ( 0.002 ms): fileCopyTest/41353 access(fd: 3
    0.175 ( 0.003 ms): fileCopyTest/41353 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|DENY|WRITE
    0.181 ( 0.003 ms): fileCopyTest/41353 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|DENY|WRITE
    0.185 ( 0.002 ms): fileCopyTest/41353 arch_prctl(option: 4098, arg2: 139803916
    0.191 ( 0.003 ms): fileCopyTest/41353 mprotect(start: 0x7f26a2a00000, len: 1433
    0.242 ( 0.003 ms): fileCopyTest/41353 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|DENY|WRITE
    0.248 ( 0.004 ms): fileCopyTest/41353 mprotect(start: 0x7f26a2a00000, len: 4096
    0.253 ( 0.004 ms): fileCopyTest/41353 mmap(len: 89313, prot: READ, flags: PRIVATE|DENY|WRITE
    0.287 ( 0.132 ms): fileCopyTest/41353 clone_flags: CHILD_CLEARTID|CHILD_D
    0.432 ( 0.005 ms): fileCopyTest/41353 open(filename: 0x2ff92835
    0.438 ( 0.253 ms): fileCopyTest/41353 open(filename: 0x2ff92846, flags: RWWRIC
    0.696 ( 0.003 ms): fileCopyTest/41353 fstat(fd: 3, statbuf: 0x7ffcc2ff92360
    0.699 ( 0.002 ms): fileCopyTest/41353 lseek(fd: 4, offset: 9946, whence: SET
    0.703 ( 0.713 ms): fileCopyTest/41353 write(fd: 4, buf: 0x400b77, count: 1
    1.423 ( 0.018 ms): fileCopyTest/41353 mmap(len: 9947, prot: READ|WRITE, flags: PRIVATE|DENY|WRITE
    1.434 ( 0.008 ms): fileCopyTest/41353 mmap(len: 9947, prot: READ|WRITE, flags: PRIVATE|DENY|WRITE
    1.499 ( 0.001 ms): fileCopyTest/41353 exit_group(
```

## [Questions]

### 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?

**KASLR** (Kernel Address Space Layout Randomization) 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 from 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) Create a graph of file size vs. execution times of the four functions.  
How is the behavior of each function?

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

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

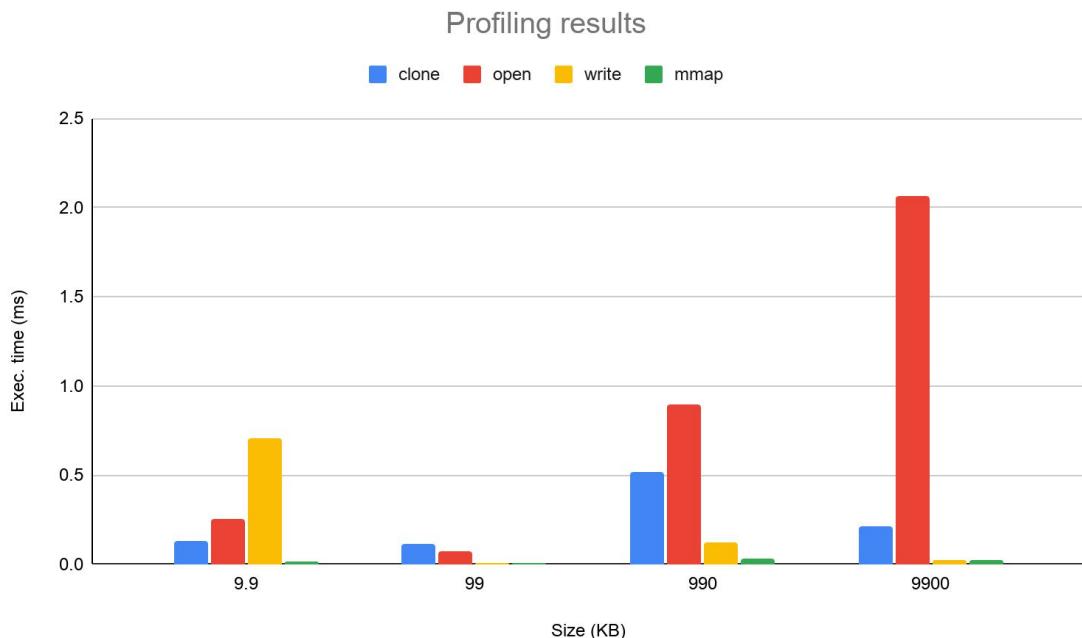
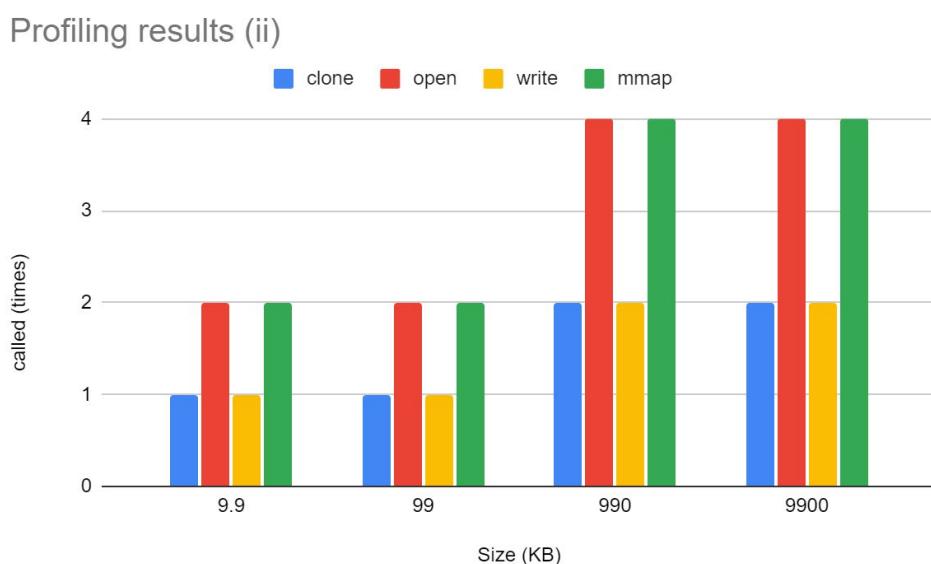


Figure (b) - times that function was called (times) vs. File Size(KB)



Fig(c) - mmap function calling times

9.9KB (2 times)

```
# 0.248 { 0.004 ms}: fileCopyTest/41353 mprotect(start: 0x7f26a2eb5, size: 0x1000, prot: 0x40)
# 0.253 { 0.006 ms}: fileCopyTest/41353 munmap(addr: 0x7f26a2e9f000, size: 0x1000)
# 0.287 { 0.132 ms}: fileCopyTest/41353 clone(clone_flags: CHILD_CLOEXEC)
# 0.432 { 0.005 ms}: fileCopyTest/41353 open(filename: 0x2ff92035, fd: 3, flags: 0x100)
# 0.438 { 0.253 ms}: fileCopyTest/41353 open(filename: 0x2ff92046, fd: 3, flags: 0x100)
# 0.696 { 0.002 ms}: fileCopyTest/41353 fstat(fd: 3, statbuf: 0x7ff1699)
# 0.699 { 0.002 ms}: fileCopyTest/41353 lseek(fd: 4, offset: 9946, whence: SEEK_SET)
# 0.703 { 0.711 ms}: fileCopyTest/41353 write(fd: 4, buf: 0x400b77, count: 0x1)
# 1.423 { 0.010 ms}: fileCopyTest/41353 mmap(len: 9947, prot: READ, fd: 3, offset: 0x1000)
# 1.434 { 0.008 ms}: fileCopyTest/41353 mmap(len: 9947, prot: READ)
# 1.493 { 0.001 ms}: fileCopyTest/41353 exit_group()
```

99KB (2 times)

```
# 0.242 { 0.007 ms}: fileCopyTest/41494 munmap(addr: 0x71289991a000, size: 0x1000)
# 0.270 { 0.116 ms}: fileCopyTest/41494 clone(clone_flags: CHILD_CLOEXEC)
# 0.401 { 0.005 ms}: fileCopyTest/41494 open(filename: 0xc1934835, fd: 3, flags: 0x100)
# 0.407 { 0.066 ms}: fileCopyTest/41494 open(filename: 0xc1934846, fd: 3, flags: 0x100)
# 0.475 { 0.002 ms}: fileCopyTest/41494 fstat(fd: 3, statbuf: 0x7ff1647)
# 0.479 { 0.002 ms}: fileCopyTest/41494 lseek(fd: 4, offset: 99470, whence: SEEK_SET)
# 0.483 { 0.011 ms}: fileCopyTest/41494 write(fd: 4, buf: 0x400b77, count: 0x1)
# 0.496 { 0.003 ms}: fileCopyTest/41494 mmap(len: 99471, prot: READ, fd: 3, offset: 0x1000)
# 0.501 { 0.003 ms}: fileCopyTest/41494 mmap(len: 99471, prot: READ)
# 0.593 { 0.001 ms}: fileCopyTest/41494 exit_group()
```

990KB (4 times)

```
26 # 0.644 { 0.012 ms}: fileCopyTest/41560 munmap(addr: 0x7ffb03d2a0, size: 0x1000)
27 # 0.710 { 0.519 ms}: fileCopyTest/41560 clone(clone_flags: CHILD_CLOEXEC)
28 # 1.329 { 0.008 ms}: fileCopyTest/41560 open(filename: 0xe294183, fd: 3, flags: 0x100)
29 # 1.338 { 0.383 ms}: fileCopyTest/41560 open(filename: 0xe294184, fd: 3, flags: 0x100)
30 # 1.728 { 0.003 ms}: fileCopyTest/41560 fstat(fd: 3, statbuf: 0x7ff1699)
31 # 1.733 { 0.002 ms}: fileCopyTest/41560 lseek(fd: 4, offset: 99470, whence: SEEK_SET)
32 # 1.738 { 0.046 ms}: fileCopyTest/41560 write(fd: 4, buf: 0x400b77, count: 0x1)
33 # 1.788 { 0.009 ms}: fileCopyTest/41560 mmap(len: 994701, prot: READ, fd: 3, offset: 0x1000)
34 # 1.799 { 0.004 ms}: fileCopyTest/41560 mmap(len: 994701, prot: READ)
35 # 3.100 { 0.001 ms}: fileCopyTest/41560 exit_group()
36 # ? { 0.001 ms}: fileCopyTest/41561 ... [continued]: clone()
37 # 3.655 { 0.012 ms}: fileCopyTest/41561 open(filename: 0xe294183, fd: 3, flags: 0x100)
38 # 3.692 { 0.497 ms}: fileCopyTest/41561 open(filename: 0xe294184, fd: 3, flags: 0x100)
39 # 4.223 { 0.003 ms}: fileCopyTest/41561 fstat(fd: 3, statbuf: 0x7ff1699)
40 # 4.261 { 0.002 ms}: fileCopyTest/41561 lseek(fd: 4, offset: 994701, whence: SEEK_SET)
41 # 4.280 { 0.076 ms}: fileCopyTest/41561 write(fd: 4, buf: 0x400b77, count: 0x1)
42 # 4.386 { 0.014 ms}: fileCopyTest/41561 mmap(len: 994701, prot: READ, fd: 3, offset: 0x1000)
43 # 4.411 { 0.009 ms}: fileCopyTest/41561 mmap(len: 994701, prot: READ)
44 # 5.317 { 0.001 ms}: fileCopyTest/41561 exit_group()
45
```

9900KB (4 times)

```
# 0.248 { 0.047 ms}: fileCopyTest/41636 munmap(addr: 0x13d3b831, size: 0x1000)
# 0.599 { 0.213 ms}: fileCopyTest/41636 clone(clone_flags: CHILD_CLOEXEC)
# 0.841 { 0.012 ms}: fileCopyTest/41636 open(filename: 0x13d3b831, fd: 3, flags: 0x100)
# 0.857 { 1.145 ms}: fileCopyTest/41636 open(filename: 0x13d3b841, fd: 3, flags: 0x100)
# 2.008 { 0.003 ms}: fileCopyTest/41636 fstat(fd: 3, statbuf: 0x7ff1699)
# 2.013 { 0.002 ms}: fileCopyTest/41636 lseek(fd: 4, offset: 994701, whence: SEEK_SET)
# 2.017 { 0.013 ms}: fileCopyTest/41636 write(fd: 4, buf: 0x400b77, count: 0x1)
# 2.032 { 0.007 ms}: fileCopyTest/41636 mmap(len: 994701, prot: READ, fd: 3, offset: 0x1000)
# 2.040 { 0.003 ms}: fileCopyTest/41636 mmap(len: 994701, prot: READ)
# ? { 0.001 ms}: fileCopyTest/41637 ... [continued]: clone()
# 2.812 { 0.011 ms}: fileCopyTest/41637 open(filename: 0x13d3b831, fd: 3, flags: 0x100)
# 2.847 { 0.009 ms}: fileCopyTest/41637 open(filename: 0x13d3b841, fd: 3, flags: 0x100)
# 3.765 { 0.003 ms}: fileCopyTest/41637 fstat(fd: 3, statbuf: 0x7ff1699)
# 3.773 { 0.002 ms}: fileCopyTest/41637 lseek(fd: 4, offset: 994701, whence: SEEK_SET)
# 3.780 { 0.013 ms}: fileCopyTest/41637 write(fd: 4, buf: 0x400b77, count: 0x1)
# 3.794 { 0.008 ms}: fileCopyTest/41637 mmap(len: 994701, prot: READ, fd: 3, offset: 0x1000)
# 3.804 { 0.003 ms}: fileCopyTest/41637 mmap(len: 994701, prot: READ)
# 14.086 { 0.001 ms}: fileCopyTest/41638 exit_group()
```

- Select another function from the `stcall_64.tbl` table, look for where it is implemented and create your own scenario to trigger it.
- Shown in ScreenShot10 ~ 15 above.**

- (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

```

user@0511330@ubuntu:~/Desktop/Profiling tests/fileCopyTest$ gcc -ggdb -w -g -o fileCopyTest fileCopyTest.c
user@0511330@ubuntu:~/Desktop/Profiling tests/fileCopyTest$ sudo perf record -g ./fileCopyTest origin
[sudo] password for user@0511330:
Lowering default frequency rate to 2750.
Please consider tweaking /proc/sys/kernel/perf_event_max_sample_rate.
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.026 MB perf.data (50 samples) ]
Bus error (core dumped)
user@0511330@ubuntu:~/Desktop/Profiling tests/fileCopyTest$ sudo perf report
# To display the perf.data header info, please use --header/-h options.
#
# Total Lost Samples: 0
#
# Samples: 50 of event 'cycles'
# Event count (approx.): 14074899
#
# Children      Self   Command      Shared Object      Symbol
# .....          .....
#       96.39%    0.00%  fileCopyTest [unknown]          [k] 0x039e258d4c544155
|   ...-0x39e258d4c544155
|   ...- libc_start_main
|   ...--93.05%-- memcpy_avx_unaligned
|   ...--8.25%--page_fault
|   ...--do_page_fault
|   ...--handle_mm_fault
|   ...--handle_mm_fault
|   ...--do_fault
|   ...ext4_filemap_fault
|   ...filemap_fault
|

```

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 implies 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).

```
test.c - FileCopyTest - Visual Studio Code
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL
1:bash
gcc: error: fileCopyTest_stat: No such file or directory
user@ubuntu:~/Desktop/Profiling tests/fileCopyTest$ gcc -o fileCopyTest -g fileCopyTest.c
gcc: error: o: No such file or directory
user@ubuntu:~/Desktop/Profiling tests/fileCopyTest$ gcc -o fileCopyTest -g fileCopyTest.c
gcc: error: o: No such file or directory
gcc: error: g: No such file or directory
user@ubuntu:~/Desktop/Profiling tests/fileCopyTest$ gcc -ggdb -w -g -o fileCopyTest fileCopyTest.c
user@ubuntu:~/Desktop/Profiling tests/fileCopyTest$ perf stat ./fileCopyTest
usage: a.out <fromfile> <tofile>can't open (null) for reading
usage: a.out <fromfile> <tofile>can't open (null) for reading
Performance counter stats for './fileCopyTest':
      0.97 msec task-clock:u          #    0.512 CPUs utilized
          0 context-switches:u        #    0.000 K/sec
          0 cpu-migrations:u         #    0.000 K/sec
          81 page-faults:u           #    0.083 M/sec
<not counted>    cycles:u                                (0.00%)
<not counted>    stalled-cycles-frontend:u                (0.00%)
<not counted>    stalled-cycles-backend:u                (0.00%)
<not counted>    instructions:u                         (0.00%)
<not counted>    branches:u                            (0.00%)
<not counted>    branch-misses:u                      (0.00%)
      0.001898626 seconds time elapsed
      0.001568000 seconds user
      0.000000000 seconds sys

Some events weren't counted. Try disabling the NMI watchdog:
echo 0 > /proc/sys/kernel/nmi_watchdog
perf stat ...
echo 1 > /proc/sys/kernel/nmi_watchdog
user@ubuntu:~/Desktop/Profiling tests/fileCopyTest$
```