Operating System

project 2

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

Ojective:

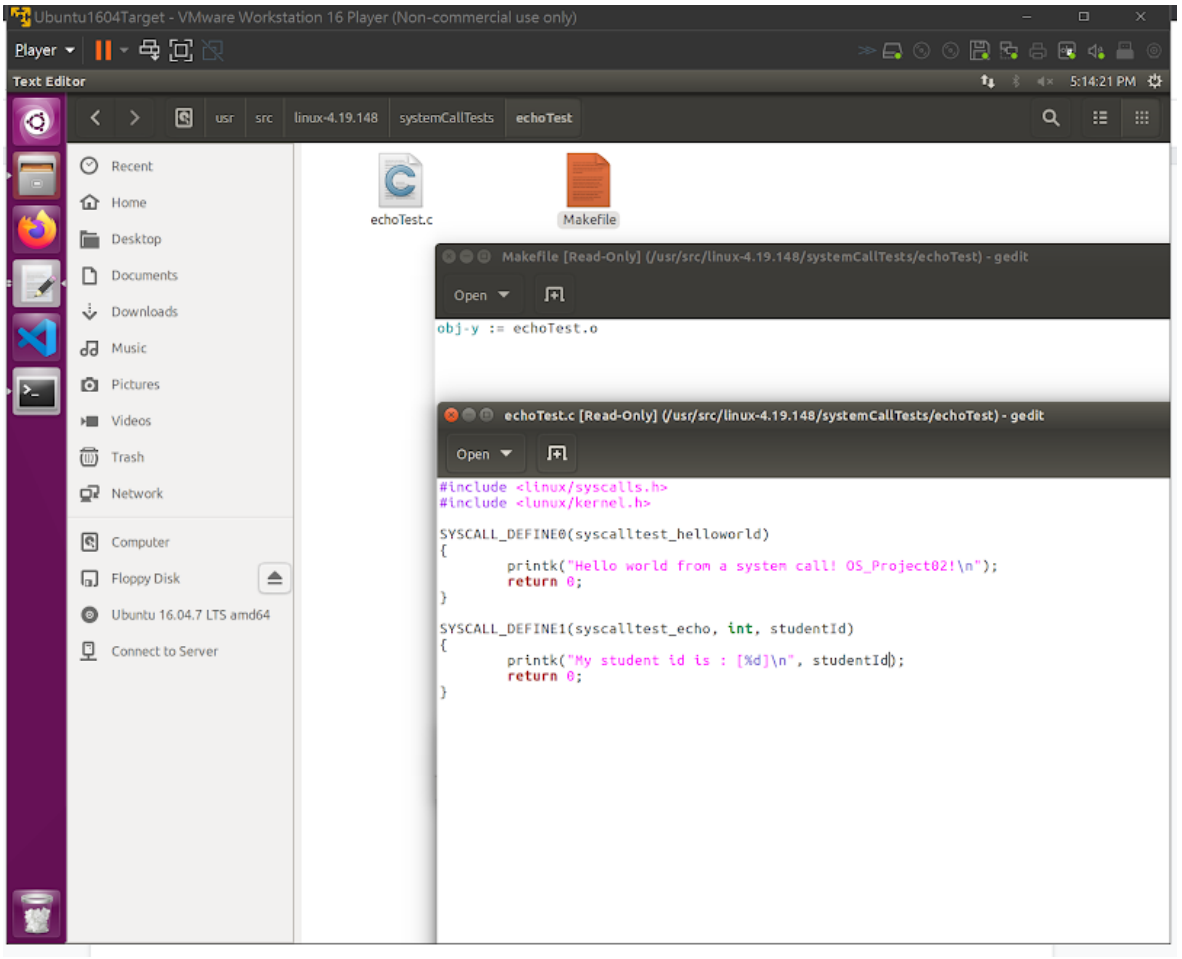
To help the student to get familiar with system calls, including the definition, usage and implementation of a system call.

[Section 1-2]

[Screenshot 1] Defining a new system call.

1. In order to place our custom system call code, create a new folder **systemCallTests** and a subfolder **echoTest**.
(in the kernel folder: **/usr/src/linux-4.19.148**)
[Command: **$sudo mkdir systemCallTests**, and so does echoTest.]

2. Create a C file **echoTest.c** to implement our system calls, and a text file **Makefile** to ensure that our system calls will be compiled.
[Command: **$sudo gedit echtoTest.c**, and so does **Makefile**]



[Note]
We also need to notify the compiler where our c file is, i.e., which folder our files will be. So append the folder name **syscallTests/echoTest/** to the end of the line **core-y += kernel/ certs/ mm/…** in the Makefile, which should be in the kernel folder **/usr/src/linux-4.19.148**.

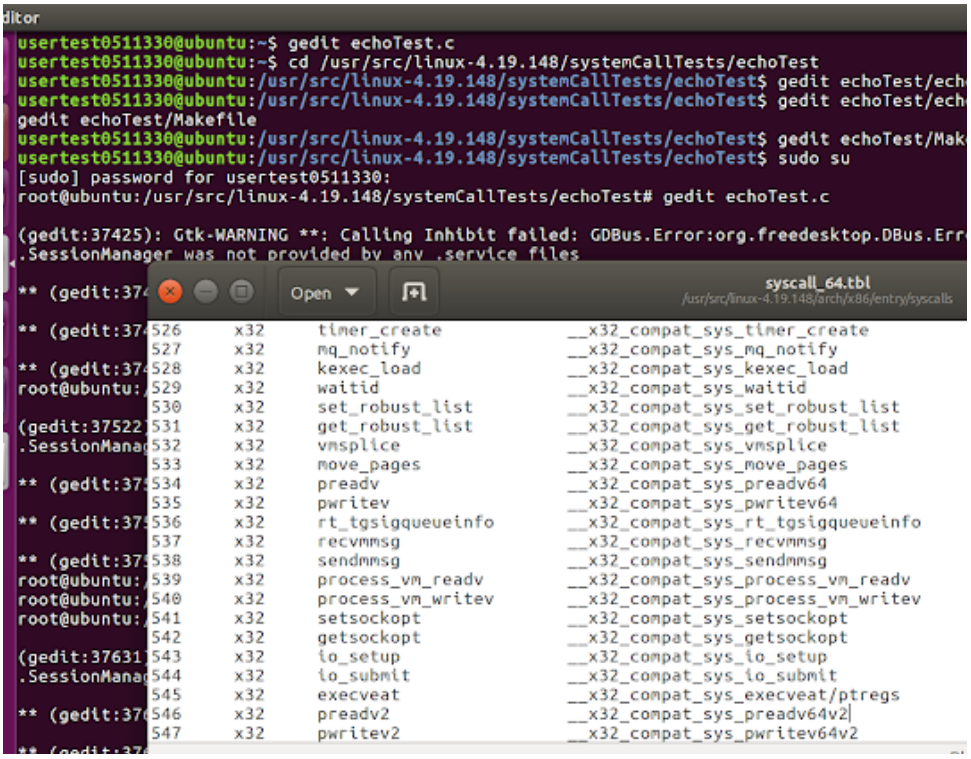
[Screenshot 2, 3] Modify syscall_64.tbl

Adding our system call into the system call table **syscall_64.tbl**, to let the user program can access it.

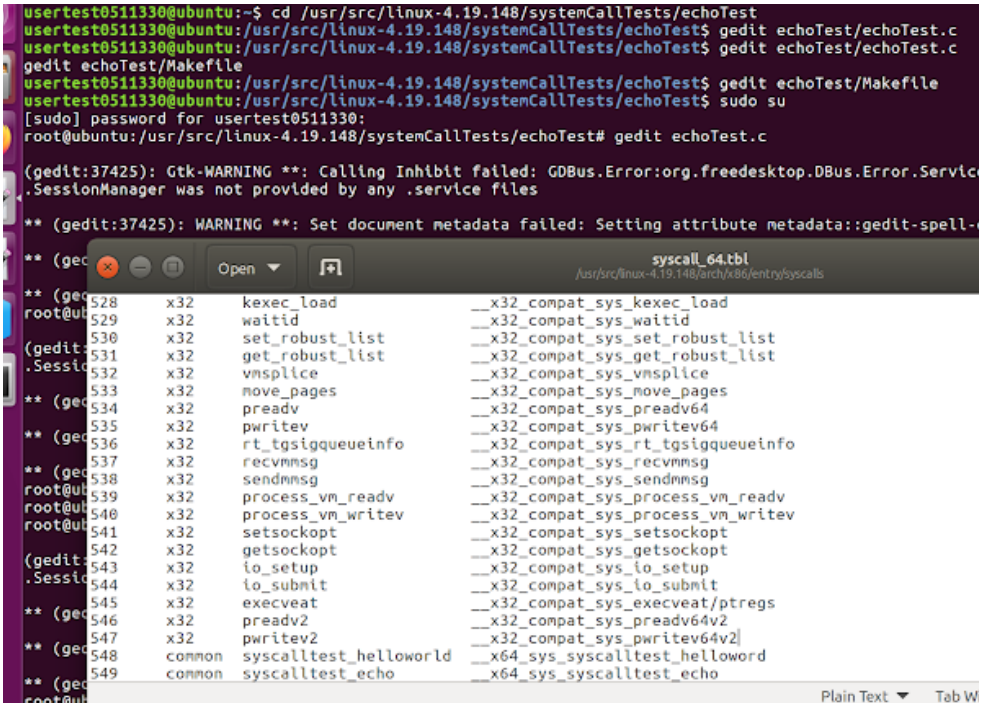
[Command: **$sudo gedit syscall_64.tbl]**

(under the path: **/linux-4.19.148/arch/x86/entry/syscalls**)

before



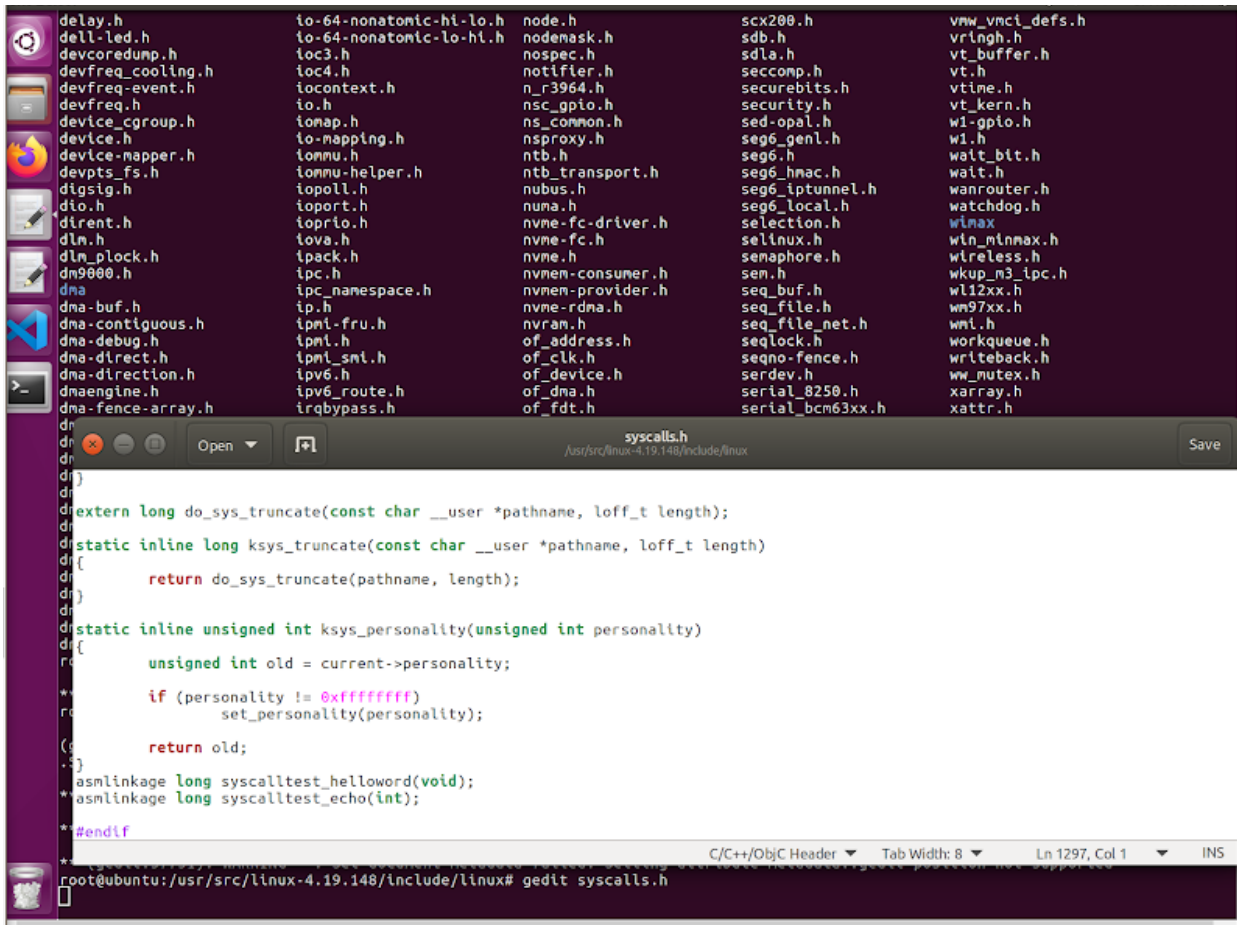
after



[Screenshot 4] syscalls.h

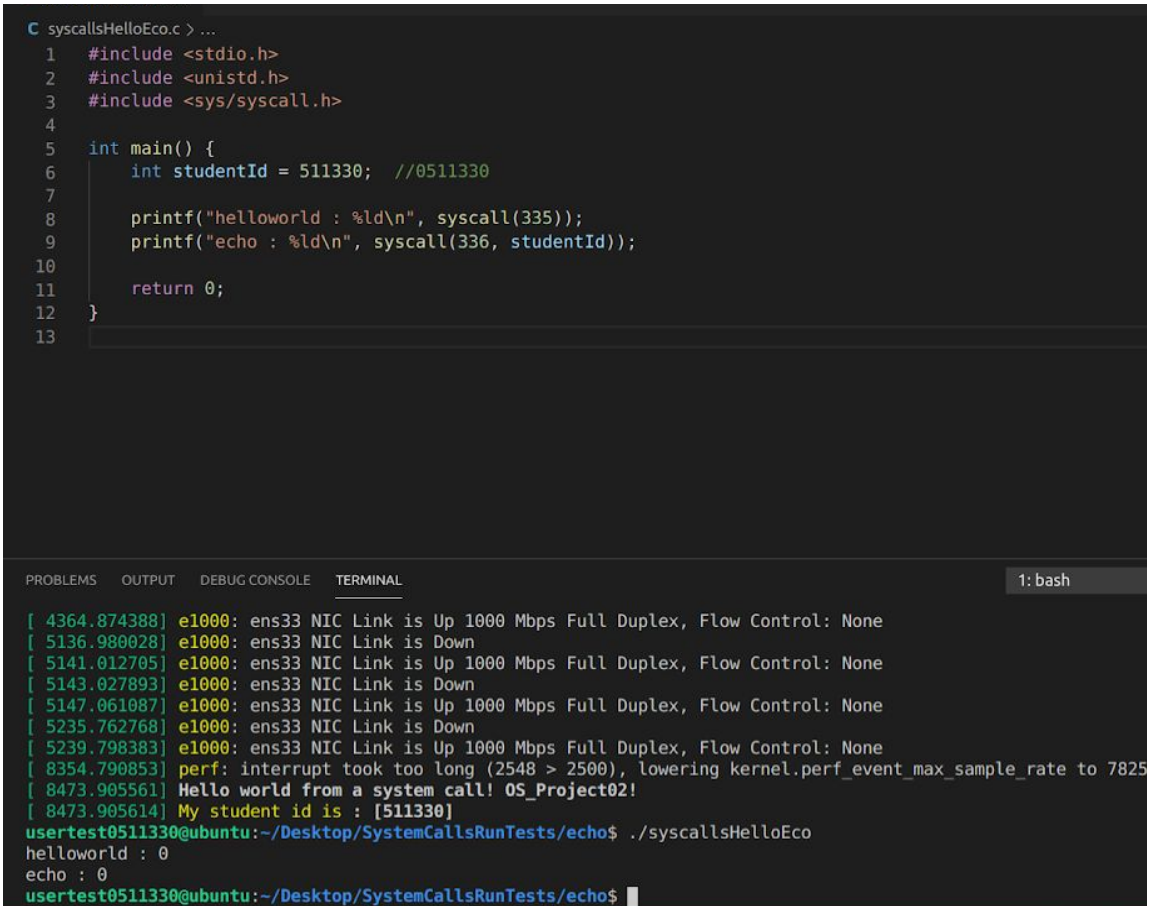
Define the prototype of the function in **syscalls.h**. “asmlinkage” is a keyword used to indicate that all parameters of the function would be available on the stack.

[command: **$sudo gedit syscalls.h**, path: **/usr/src/include/linux**]



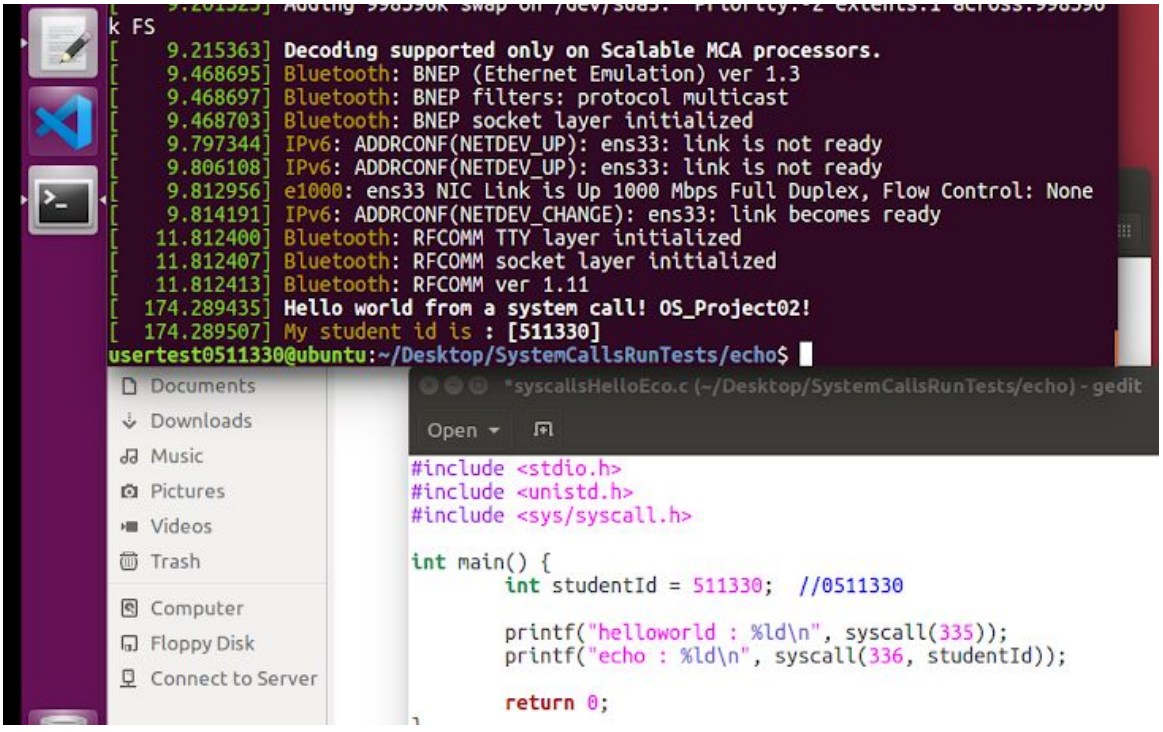
[Screenshot 5] Test syscalls whether it works or not

After recompile and reboot the system, create a C file to test if the system call we made works.



```
C syscallsHelloEco.c > ...
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main() {
    int studentId = 511330;  //0511330

    printf("helloworld : %ld\n", syscall(335));
    printf("echo : %ld\n", syscall(336, studentId));

    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                                1: bash

[ 4364.874388] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 5136.980028] e1000: ens33 NIC Link is Down
[ 5141.012705] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 5143.027893] e1000: ens33 NIC Link is Down
[ 5147.061087] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 5235.762768] e1000: ens33 NIC Link is Down
[ 5239.798383] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 8354.790853] perf: interrupt took too long (2548 > 2500), lowering kernel.perf_event_max_sample_rate to 7825
[ 8473.905561] Hello world from a system call! OS_Project02!
[ 8473.905614] My student id is : [511330]
usertest0511330@ubuntu:~/Desktop/SystemCallsRunTests/echo$ ./syscallsHelloEco
helloworld : 0
echo : 0
usertest0511330@ubuntu:~/Desktop/SystemCallsRunTests/echo$
```



```
k FS
[    9.215363] Decoding supported only on Scalable MCA processors.
[    9.468695] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[    9.468697] Bluetooth: BNEP filters: protocol multicast
[    9.468703] Bluetooth: BNEP socket layer initialized
[    9.797344] IPv6: ADDRCONF(NETDEV_UP): ens33: link is not ready
[    9.806108] IPv6: ADDRCONF(NETDEV_UP): ens33: link is not ready
[    9.812956] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[    9.814191] IPv6: ADDRCONF(NETDEV_CHANGE): ens33: link becomes ready
[   11.812400] Bluetooth: RFCOMM TTY layer initialized
[   11.812407] Bluetooth: RFCOMM socket layer initialized
[   11.812413] Bluetooth: RFCOMM ver 1.11
[  174.289435] Hello world from a system call! OS_Project02!
[  174.289507] My student id is : [511330]
usertest0511330@ubuntu:~/Desktop/SystemCallsRunTests/echo$
```

Documents
Downloads
Music
Pictures
Videos
Trash
Computer
Floppy Disk
Connect to Server

*syscallsHelloEco.c (~/Desktop/SystemCallsRunTests/echo) - gedit

Open

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main() {
        int studentId = 511330;  //0511330

        printf("helloworld : %ld\n", syscall(335));
        printf("echo : %ld\n", syscall(336, studentId));

        return 0;
```

[Section 1-3]

[Screenshot 6] numericalTest.c (syscall)

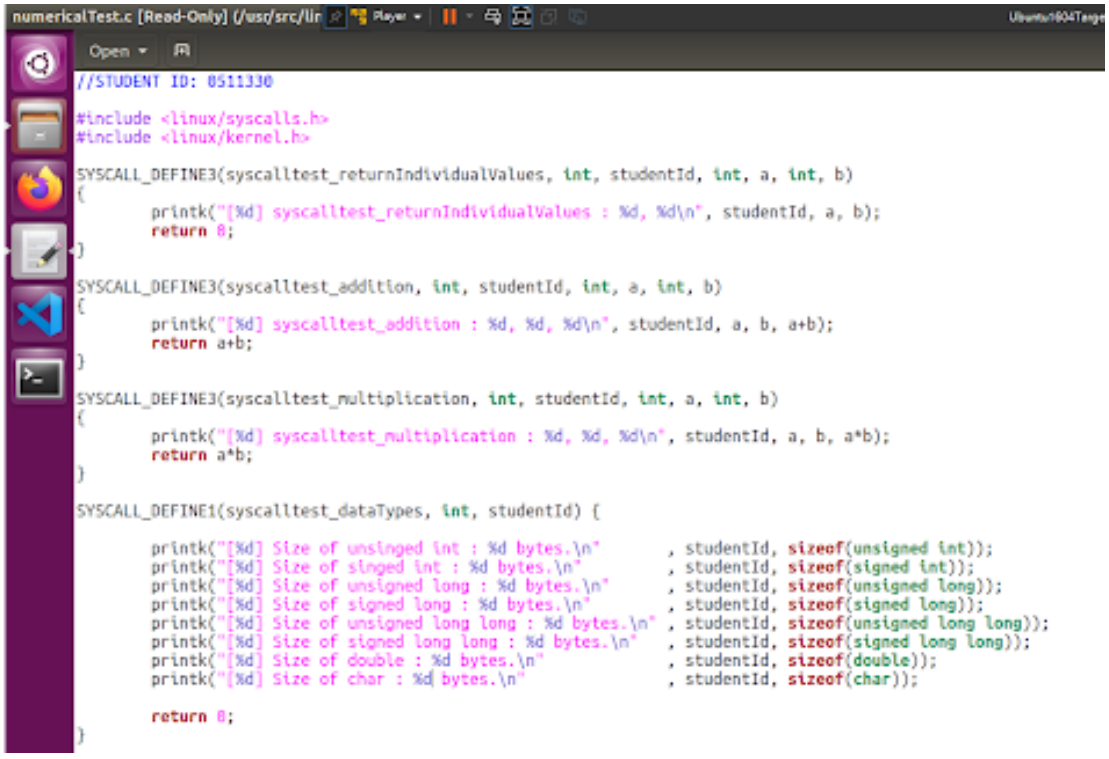
The number in the function name **SYSCALL_DEFINE3** means this system call contains **3** parameters.

1. **syscalltest_returnIndividualValues**
   It contains 3 integer parameters: **studentId**, **a** and **b**, and does the following things:
   (i) Print **studentId**, **a**, and **b** in the kernel ring buffer.
   (ii) Return value 0 to where it was called.

2. **syscalltest_addition**
   It contains 3 integer parameters: **studentId**, **a** and **b**, and does the following things:
   (i) Print **studentId, a, b** and **a+b** in the kernel ring buffer.
   (ii) Return value **a+b** to where it was called.

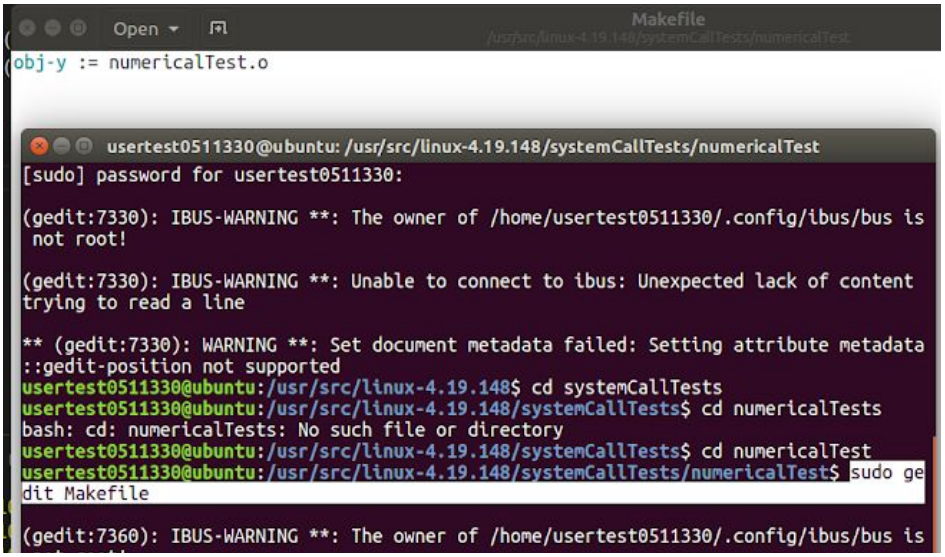
3. **syscalltest_muliplication**
   It contains 3 integer parameters: **studentId**, **a** and **b**, and does the following things:
   (i) Print **studentId, a, b** and **a*b** in the kernel ring buffer.
   (ii) Return value **a*b** to where it was called. (refer to **[Note]**)
4. **syscalltest_dataTypes**
   It contains 1 integer parameters: **studentId**, and does the following things:
   (i) Print the size of basic data type in the kernel ring buffer.
   (by **sizeof** operator)
   (ii) Return value 0 to where it was called.

   [Note]
   If a system call **returns a negative value**, it's treated as an **error**. A special error handling code invoked in libc. So it returns **-1** instead of our expectation when it returns a negative value.
   We need to handle this situation by passing a pointer containing our result to a kernel function **copy_to_user**, and it will copy the pointer in the kernel area to user space, then we can get the correct return value in user space which is negative.



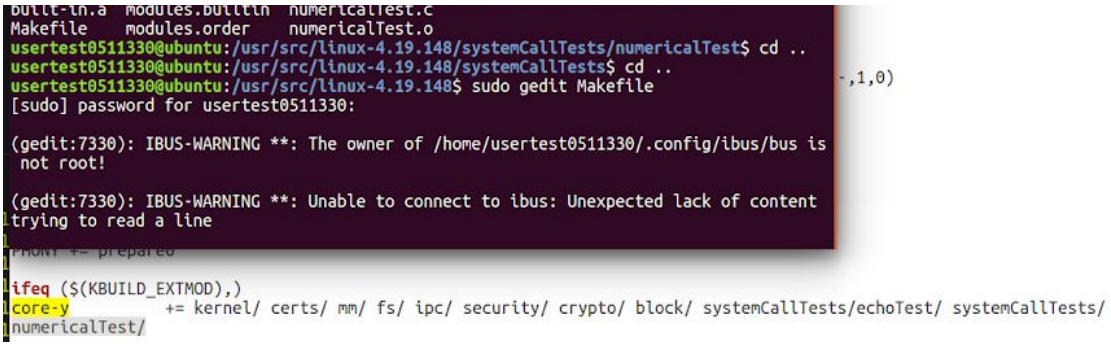
[Screenshot 7] Create the Makefile for our syscalls.

Create a Makefile and fill it with line: **obj-y := numericalTest.o**
As mentioned, This is to make sure that our file (**numericalTest.c**) will be compiled.
[Command: **$sudo gedit Makefile]**
[folder: **/linux-4.19.148/sysCallTests/numericalTest**]



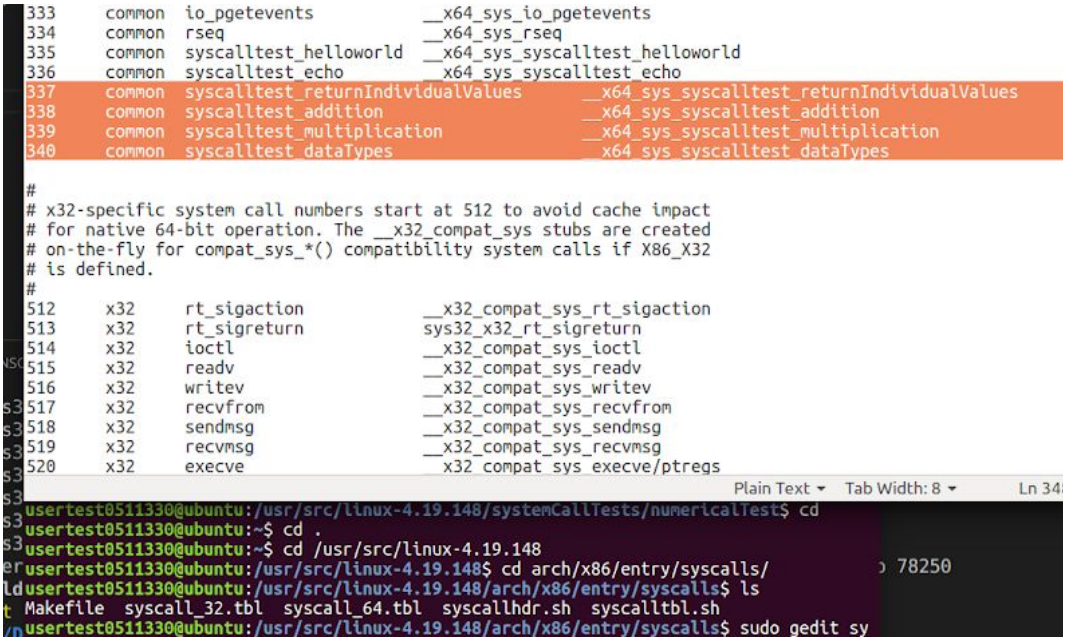
[Screenshot 8] Appending the directory to linux Makefile

We also need to tell the compiler where the syscalls we defined.
Append the directory **numericalTest/** to Makefile, which is lying in the kernel folder.
[Command: **$sudo gedit Makefile]**
[folder: **/linux-4.19.148**]



[Screenshot 9]

For calling system calls from a user program in userspace, label our function to the syscalls table **syscalls_64.tbl**.
[Command: **$sudo gedit syscalls_64.tbl**]
[in folder: **/linux-4.19.148/arch/x86/entry/syscalls**]

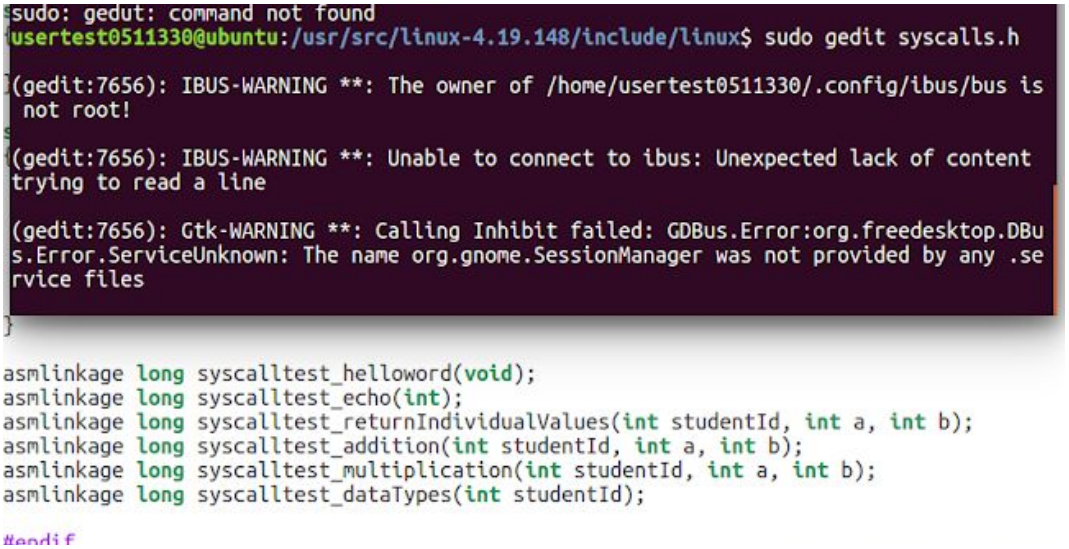


[Screenshot 10]

As the previous section mentioned, we also need to add the prototype of the function to **syscalls.h**. “asmlinkage” is a keyword used to indicate that all parameters of the function would be available on the stack.

[Command: **$sudo gedit syscalls.h**]

[folder: **/linux-4.19.148/include/linux**]



Finally, we can recompile the kernel. run the following commands:

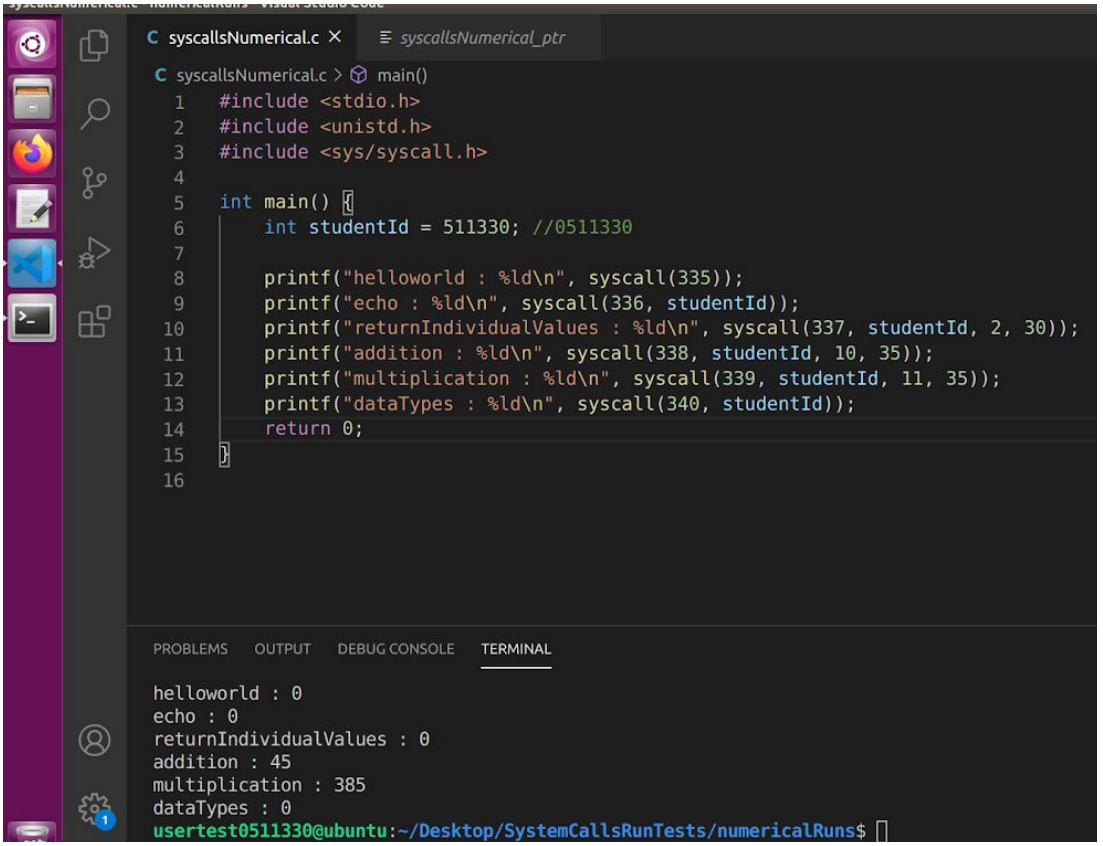
$sudo su
make menuconfig
make -j $(nproc)
make modules_install
make install

[Screenshot 11] Source code of syscallsNumerical.c

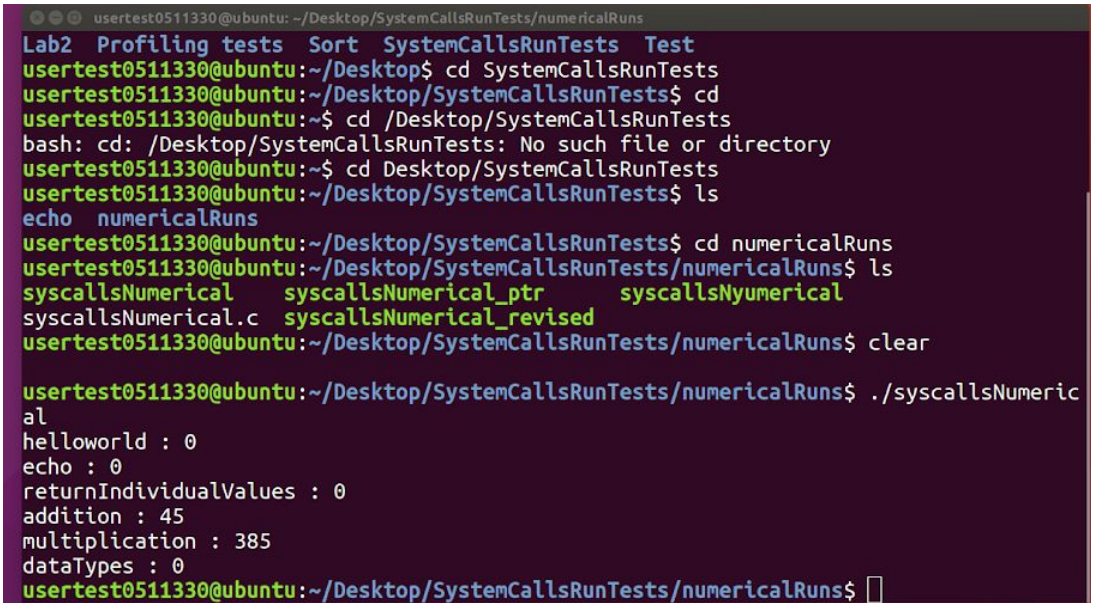
The source code to test syscalls as shown.

[Remind]

When you expect to receive a **negative value return** from syscalls, It's treated as an error and always returns **-1** instead of your expectation.

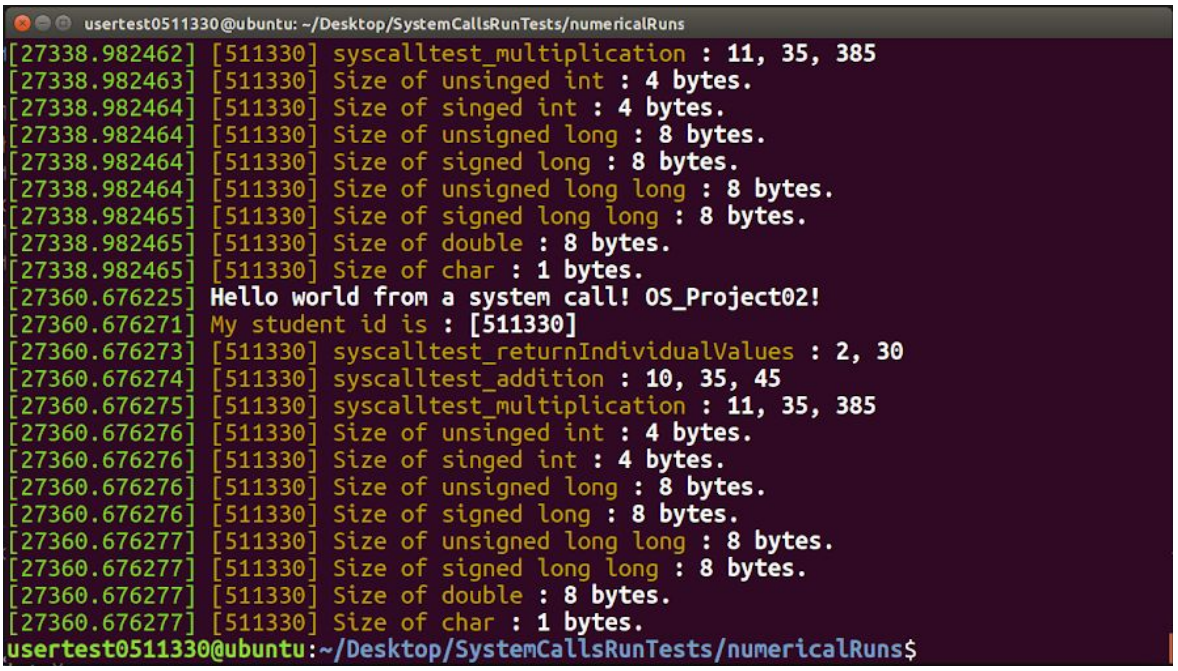


[Screenshot 12] Results of calling custom syscall



[Screenshot 13] Messages printed on the kernel ring buffer

[Command: **dmesg**] to see the messages printed in the ring buffer.



[Quetions]

1. What is Kernel space and User space? What's the difference between them?
   (i) Kernel Space: Kernel space is strictly reserved for running a privileged instruction of the system kernel, kernel extensions, and device drivers.

   (ii) User Space: User space is the memory area where application software and some drivers execute.

   The separation serves to provide memory the mechanism of OS such as memory protection and hardware protection from malicious or errant software behaviour. The user processes can only access a small part of the kernel via and interface - **the system calls**

2. What are protection rings and how many? What are Ring 0 and Ring 1?
   Protection ring is the mechanism to protect data and functionality from faults or malicious behavior.It's mainly to implement the idea of hierarchical protection domains.Typically in x86 architecture, there are 4 rings of protection.
   - Ring 0 is kernel mode, the level with the most privileges and interacts most directly with hardware. (The most privileged one)
   - Ring 1 may be used by virtual machine hypervisors or drivers.

3. What is a system call? How many are there? What are the differences?
   The interface between kernel space and user space. When a user process needs to access the kernel, a system call will be called.
   There are 547 system calls in linux version 4.19.148. Somes are used to handle process interrupts, somes serve user process the I/O requests, or deal with messages passing between processes.

4. For the custom kernel built, where is the list of system calls.
   File name: syscall_64.tbl
   Directory: /usr/src/linux-4.19.148/arch/x86/entry/syscalls/syscall_64.tbl

5. What is the system call ID?
   System call ID is the unique number to each system call. All system calls will be placed in a system call table. When invoking a system call, the OS will find the system call in the table by system call ID, treated as **array index** to speed up.

6. (i) What do the reserved words “asmlinkage” and “printk” mean?
   asmlinkage is a keyword to indicate all parameters of function would be available on the stack instead of passed by register.

   (ii) What does the command **update-initramfs** do?
   It will generate an initramfs image, managing your initramfs image on your local box

7. How do you use printk and read the messages printed by it?
   Simply include the <linux/kernel.h> header to use it. We can use the command “dmesg” to see the logs which were printed in the kernel ring buffer.

8. What is the kernel ring buffer? How do you read its contents?
   The kernel ring buffer is a data structure that records messages related to the operation of the kernel.
   dmesg will output all kernel messages currently in the ring buffer to a file called "kernel_msgs.txt”.

9. What is a function signature?
   A function signature defines input and output of functions, including: parameters & their datatype, return value & their datatype, and other advanced information (e.g. **static**).

10. What does SYSCALL_DEFINE[n] mean? What is n?
    It means that there will be **n** parameters in self-defined system call.

11. For a system call wrapper, how does its function signature look like when it has 0 inputs as parameters? 1, 2 or 3 integer numbers as input?
    0 input : SYSCALL_DEFINE0(*syscall name***)**
    1 input : **SYSCALL_DEFINE1(***syscall name, parameter 1*)
    n inputs: **SYSCALL_DEFINEn(***syscall name, parameter 1, …, parameter n*)
    (n<6)

12. Does the function signature of a SYSCALL_DEFINE change depending on the type of element returned?
    No, the function signature of **SYSCALL_DEFINE** is independent of their return type.

13. What is #include<linux/kernel.h> and #include<linux/syscalls.h>?
    <linux/kernel.h> is a header which gets used for module builds. It's part of the kernel source and intended for user processes.
    <linux/syscalls.h> includes the system calls which are exposed to userspace.