Introduction to Operating Systems

Project 2: Linux kernel system calls
(To be lectured on 2021-11-05)

Prof: Ying-Dar Lin (林盈達）
TA: Ricardo Pontaza

Deadline: 2020-11-26 (Fri) at **23:50:00**

Q&A:
If you have any questions, please post it on the E3 discussion board, and it will be answered within two days.

Deliverables:
1. **Demo video** (5 minutes – upload to YouTube – add link in the first page of the report).
2. **Report** (pdf file with file name of the form **OS_Project02_StudentID.pdf**)
   - Screenshots + one small explanation paragraph per screenshot section.
   - Answers to questions below.
   - In both the demo video and report, for each screenshot, explain what has been done, and the reasoning behind the steps.
3. **Files: (To be uploaded to E3)**
   a. K4_numericalTest.c and K5_numericalTest.c
   b. K4_syscall_64.tbl and K5_syscall_64.tbl
   c. K4_syscalls.h and K5_syscalls.h
   d. syscallsNumerical.c
4. In the C code, keep correct indentation, clean code and clear comments.

K4_numericalTest.c
K4_syscall_64.tbl
K4_syscalls.h
K5_numericalTest.c
K5_syscall_64.tbl
K5_syscalls.h
OS_Project02_012345.pdf
syscallsNumerical.c
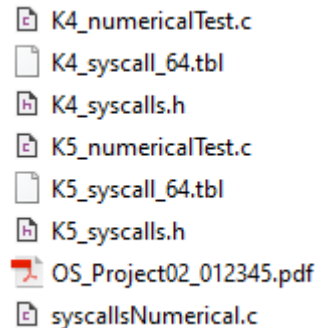
Objective:
The objective of this project is to help the student to get familiar with system calls. The student will learn the definition, usage and how to implement custom system calls.

Scope:
Understand the concept of system call, and implement custom calls.

Projects Distribution:
1. Project 01: Linux Kernel Download, Patch, Compilation, Debugging and Profiling
   (Ubuntu 16.04.7 **Server** - Kernel **4.4.101** + Ubuntu 16.04.7 **Desktop** – Kernel **4.15.0-142**)
2. **Project 02: Linux Kernel System Calls**
   (Ubuntu **20.04.3 Desktop** – Kernel **4.19.148** + Kernel **5.13.19**)
3. Project 03: Dynamically-Loadable Kernel Modules (DLKMs)
   (Ubuntu **20.04.3 Desktop** – Kernel **4.19.148** + Kernel **5.13.19**)

Table of contents:
- Section 1: Kernels Download / Build
- Section 2: System Calls

Questions to be answered in the report:

1. What is Kernel space? What is user space? What are the differences between them?
2. What are protection rings? How many are them? What is Ring 0? What is Ring 1?
3. What is a system call? How many types are they in total? What are the differences between all the types?
4. For the custom kernel built in project 01, where is the list of system calls? (Give the file name and path)
5. What is the system call ID?
6. What do the reserved words "*asmlinkage*" and "*printk*" mean?
7. How do you use **printk**? How do you read the messages printed by **printk**?
8. What is the **kernel ring buffer**? How do you read its contents?
9. What is a function signature?
10. What does **SYSCALL_DEFINE[n]** mean? What is **n**?
11. For a system call wrapper (SYSCALL_DEFINE), how does its function signature look like when it has 0 inputs as parameters? 1 integer number as input? 2 integer numbers as inputs? 3 integer numbers as inputs?
12. Why the function signature of a SYSCALL_DEFINE wrapper doesn't change depending on the type of element returned?
13. What is **#include <linux/kernel.h>**? What is **#include <linux/syscalls.h>**?

Use the project report check list as your first page of your report. (Next page)

Operating Systems Project Report

Project Number (01 / 02 / 03):	
Name:	
Student ID:	
YouTube link (Format youtube.com/watch?v=[key]):	https://www.youtube.com/watch?v=
Date (YYYY-MM-DD):	
Names of the files uploaded to E3:	
Physical Machine Total RAM (Example: 8.0 GB):	
Physical Machine CPU (Example: Intel i7-2600K):	

Checklist	
Yes/No	**Item**
	The report name follows the format "OS_Project02_StudentID.pdf".
	The report was uploaded to E3 before the deadline.
	The YouTube video is public, and anyone with the link can watch it.
	The audio of the video has a good volume.
	The pictures in your report and video have a good quality.
	All the questions and exercises were answered inside the report.
	I understand that late submission is late submission, regardless of the time uploaded.
	I understand that any cheating in my report / video / code will not be tolerated.

SECTION 1:

KERNELS DOWNLOAD / BUILD

<p align="center">**Section 1: Kernel Download / Build**</p>

In this section we will download and install Ubuntu 20.04 and kernels 4.19.148 And kernel 5.13.19.

Section 1.1: OS Installation

1. Download the **Ubuntu 20.04 Desktop** iso: https://releases.ubuntu.com/20.04/
2. [Project 01-Section 01] Inside the "VHD-VMWarePlayer" folder, create a new subfolder called **Ubuntu20.04.**
3. Create a virtual machine with the following characteristics:
    a. OS: **Ubuntu 20.04 Desktop**
    b. RAM: 3,072 Mb
    c. HD: 100 Gb (**Split in multiple files**, and **not preallocated**).
    d. Username: **usertest+StudentID** (Example: **usertest12345**).
    e. Do a <u>manual</u> partition table, and create the following partitions:
        i. Primary (ext4) – 100Gb – Mount root (/)
        ii. Logical – 7.4Gb – Swap
4. After you installed the OS, update and upgrade it

> **$ sudo apt update**
> **$ sudo apt upgrade**

and install the needed build software by running the command

> **$ sudo apt-get install build-essential libncurses-dev bison flex libssl-dev libelf-dev**

Section 1.2: Download and Build Kernels source code

In this section we download and install the source code of Kernels 4.19.148 and 5.13.19. If you have any questions about how to perform any step, check Project 01 – Section 2 and Section 3.

1. Inside the **/usr/src** folder, download and decompress the code for kernel 4.19.148:
   https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.19.148.tar.xz
   and kernel 5.13.19:
   https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.13.19.tar.xz

```
usertest@usertest-vm:/usr/src$ ls
linux-4.19.148        linux-headers-5.11.0-27-generic
linux-4.19.148.tar    linux-headers-5.11.0-38-generic
linux-5.13.19         linux-hwe-5.11-headers-5.11.0-27
linux-5.13.19.tar     linux-hwe-5.11-headers-5.11.0-38
usertest@usertest-vm:/usr/src$
```

<p align="center">**[Screenshot # 1: Create a screenshot showing this folder, and both kernel folders.]**</p>

2. For both kernels, create their respective **.config** file (check Project 01 – Section 3.1). <u>You don't need KGDB for this project.</u> You should end with a .config file inside both folders.

```
usertest@usertest-vm:/usr/src$ ls linux-4.19.148/.config
linux-4.19.148/.config
usertest@usertest-vm:/usr/src$ ls linux-5.13.19/.config
linux-5.13.19/.config
```

<p align="center">**[Screenshot # 2: Create a screenshot showing both .config files in both folders.]**</p>

3. In Ubuntu 20.04, there are some trusted keys that we need to turn off. Inside both .config files, search for all occurrences of the **"debian/canonical-certs.pem"** and "**debian/canonical-revoked-certs.pem**" values and replace them with "".

4. Build both kernels with the commands:

```
$ sudo make clean
$ sudo make -j $(nproc)
$ sudo make modules_install
$ sudo make install
```

Note: You can run the first three commands in parallel (in two separate windows at the same time), but for the fourth command

```
$ sudo make install
```

You must let it finish for one kernel, and until it is done, you can run it for the second kernel.
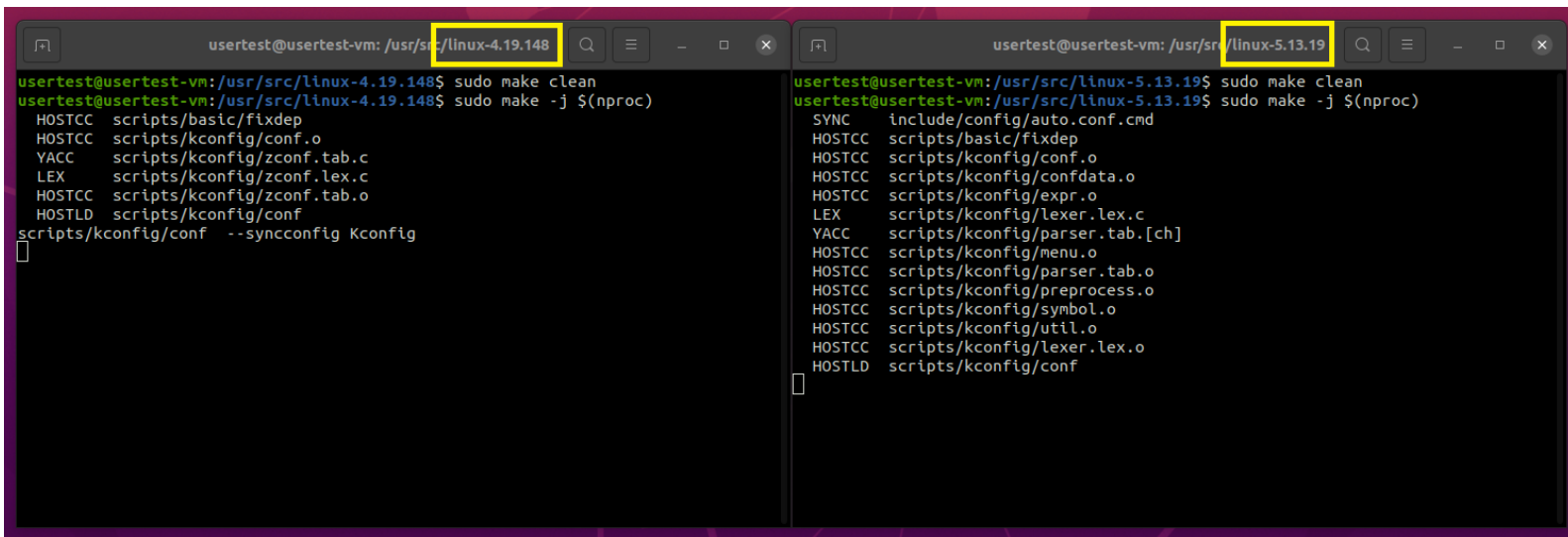
[Screenshot # 3: Create a screenshot showing both terminals running as shown above.]

5. In the **/etc/default/grub** file, comment out the **GRUB_TIMEOUT_STYLE** command and set

GRUB_TIMEOUT = 20

   ```
   #GRUB_TIMEOUT_STYLE=hidden
   GRUB_TIMEOUT=20
   ```

6. Update the grub so the grub has the new settings.

$ sudo update-grub

7. Restart the virtual machine. In the grub's kernels list, you should have both Kernel 5.13.19 and Kernel 4.19.148.
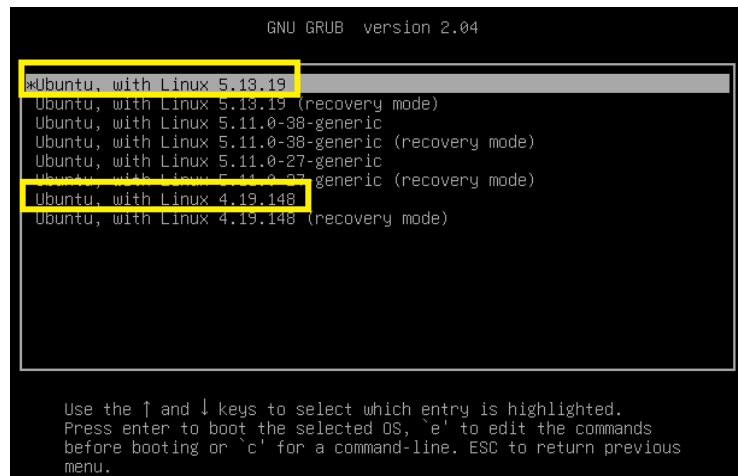
[Screenshot # 4: Create a screenshot showing the grub with both kernels.]

8. Restart using both kernels. You should be able to read the kernels version from terminal.

[Screenshot # 5: Create a screenshot showing two terminals displaying both kernel versions.]

NOTE: DO NOT PROCEED TO THE NEXT SECTION UNTIL YOU CAN BOOT USING BOTH KERNELS.

SECTION 2:
SYSTEM CALLS

Section 2: System calls

Section 2.1: Basic system call – example

In this section, we will create two basic system calls that display custom messages from both kernels. For this subsection, you can restart using any kernel.

NOTE: The steps below are performed inside the Kernel 4.19.148 folder.

1. Inside the kernel folder (4.19.148), create a folder named "**systemCallTests**", and inside this new folder, create a folder named "**echoTest**".
2. Create an **echoTest.c** file inside the **echoTest** folder, with the following content:

```c
#include <linux/syscalls.h>
#include <linux/kernel.h>

SYSCALL_DEFINE0(syscalltest_helloworld)
{
    printk("[Ker-4.19.148] Hello world from a system call! - OS_Project02!\n");
    return 0;
}

SYSCALL_DEFINE1(syscalltest_echo, int, studentId)
{
    printk("[Ker-4.19.148] My student id is : [%d]\n", studentId);
    return 0;
}
```

3. Inside the **echoTest** folder, create a **Makefile**, and add this line to it:

```
obj-y := echoTest.o
```

[Screenshot # 6: Create a screenshot showing the folder, and the contents of both echoTest.c and Makefile]

4. Go to the Makefile in the linux-4.19.148 folder and open it. Search for this line

```
core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/
```

and append **systemCallTests/echoTest/** to the end of it, so it looks like

```
core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ systemCallTests/echoTest/
```

```
ifeq ($(KBUILD_EXTMOD),)
core-y        += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ systemCallTests/echoTest/
```

5. Still inside the linux-4.19.148 folder, go to **arch/x86/entry/syscalls/** and open the **syscall_64.tbl** file. (Here we assume that the last line in this file had number 547). Add the following line at the end of the file (continue the numeration and use tabs instead of spaces to separate the columns).

```
548    common        syscalltest_helloworld    __x64_sys_syscalltest_helloworld
549    common        syscalltest_echo          __x64_sys_syscalltest_echo
```

Pay attention to the number you used for **syscalltest_helloworld**, as we will use it in step 7. Let's call it **M1**.

[Screenshots #7,8: Create a screenshot showing the before and after of the syscall_64.tbl file]

6. Go to the **linux-4.19.148/include/linux** folder. Add the signature of your system call at the end of the **syscalls.h** file:

```
asmlinkage long syscalltest_helloworld(void);
asmlinkage long syscalltest_echo(int);
```

immediately before the **#endif** line, so it will look like

```
asmlinkage long syscalltest_helloworld(void);
asmlinkage long syscalltest_echo(int);

#endif
```

[Screenshot #9: Create a screenshot showing the added lines in the syscalls.h file]

7. **[Do it yourself]**

Repeat steps 1 to 6 under the **Kernel 5.13.19** folder, using the following code for **echoTest.c**

```
#include <linux/syscalls.h>
#include <linux/kernel.h>

SYSCALL_DEFINE0(syscalltest_helloworld)
{
    printk("[Ker-5.13.19] Hello world from a system call! OS_Project02!\n");
    return 0;
}

SYSCALL_DEFINE1(syscalltest_echo, int, studentId)
{
    printk("[Ker-5.13.19] My student id is : [%d]\n", studentId);
    return 0;
}
```

In step 5, inside the **arch/x86/entry/syscalls/syscall_64.tbl** file, you must use the following rules:

a. Check the next available number for new system calls (most likely it is also 548 as in step 5, but let's call it **M2**).

b. Calculate **M = max(M1,M2)+6.** Most likely for both kernels, M1=M2=548, so

M = max(M1,M2)+6 = 548+6 = 554

but adapt the formula to the values in your kernels. Use M as the next available ID.

c. Type "**64**" instead of "**common**", and remove "**__x64_**" from the last column.

Your code should look like the one below:

```
547    x32    pwritev2            compat_sys_pwritev64v2
# This is the end of the legacy x32 range.  Numbers 548 and above are
# not special and are not to be used for x32-specific syscalls.

554    64     syscalltest_helloworld  sys_syscalltest_helloworld
555    64     syscalltest_echo        sys_syscalltest_echo
```

IMPORTANT NOTE: These rules apply for Kernel 5.13.19 only.

Before proceeding, please be sure that the system calls inside Kernel 4.19.148 and Kernel 5.13.19 HAVE DIFFERENT IDs. In this example:

Kernel 4.19.148		Kernel 5.13.19	
System call	**ID number**	**System call**	**ID number**
syscalltest_helloworld	548	syscalltest_helloworld	554
syscalltest_echo	549	syscalltest_echo	555

Adapt the values according to your kernels.

[Screenshot #10,11,12,13: Create screenshots replicating screenshots 6,7,8,9 but with Kernel 5.13.19]

8. Rebuild both kernels:

```
$ sudo make clean
$ sudo make -j $(nproc)
$ sudo make modules_install
$ sudo make install
```

DO NOT PROCEED ANY FURTHER UNTIL BOTH KERNELS HAVE BEEN BUILT SUCCESSFULLY.

[Screenshot #14: Create a screenshot showing both kernels successfully built, by showing the result of $sudo make install]

After both are done, restart your machine and boot into **Kernel 4.19.148**.

9. In your desktop, create a folder called **SystemCallsRunTests**, and inside a subfolder called **echo**. Inside the echo folder, create the following program and call it **syscallsHelloEco.c**

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

int main()
{
    int studentId = 12345;

    printf("studentId = [%d]\n",studentId);

    // Kernel 4.19.148

    printf("\n=== Kernel 4.19.148 ===\n");

    printf("helloworld : %ld\n", syscall(548));
    printf("echo : %ld\n", syscall(549, studentId));

    // Kernel 5.13.19
    printf("\n=== Kernel 5.13.19 ===\n");

    printf("helloworld : %ld\n", syscall(554));
    printf("echo : %ld\n", syscall(555, studentId));

    return 0;
}
```

(Check steps 5 and 7 for the correct system call ID. Also **replace 12345 with your student ID**)

10. Open a terminal and clear the dmesg log with the command

```
$ sudo dmesg --clear
```

11. <u>Display the kernel version in terminal</u>, and **Build** and **run** the program (no special gcc attribute is required). Check Project 01 or check online to see how to build c programs in Linux if you are not sure how to do it. You should get an output like the following:

```
usertest@usertest-vm:~/Desktop/systemCallsRunTests/echo$ uname -r
4.19.148
usertest@usertest-vm:~/Desktop/systemCallsRunTests/echo$ ./syscallsHelloEco
studentId = [12345]

=== Kernel 4.19.148 ===
helloworld : 0
echo : 0

=== Kernel 5.13.19 ===
helloworld : -1
echo : -1
usertest@usertest-vm:~/Desktop/systemCallsRunTests/echo$
```

Notice that the functions under Kernel 4.19.148 return the value 0 and the ones under Kernel 5.13.19 return the value -1.

12. Check the kernel log by running the command

$ dmesg

You should get the following output

```
usertest@usertest-vm:~$ dmesg
[  353.389644] [Ker-4.19.148] Hello world from a system call! - OS_Project02!
[  353.389649] [Ker-4.19.148] My student id is : [12345]
usertest@usertest-vm:~$
```

[Screenshot #15: Create a screenshot showing both outputs of terminal and dmesg when running the program using the Kernel 4.19.148]

13. Restart the machine and boot into **Kernel 5.13.19**. Again, <u>clear the dmesg log</u>, display the kernel version in terminal and run the **syscallsHelloEco** program. You should see an output like the one below.

```
usertest@usertest-vm:~/Desktop/systemCallsRunTests/echo$ uname -r
5.13.19
usertest@usertest-vm:~/Desktop/systemCallsRunTests/echo$ ./syscallsHelloEco
studentId = [12345]

=== Kernel 4.19.148 ===
helloworld : -1
echo : -1

=== Kernel 5.13.19 ===
helloworld : 0
echo : 0
usertest@usertest-vm:~/Desktop/systemCallsRunTests/echo$
```

Notice that now the functions under Kernel 4.19.148 return -1 and the ones under Kernel 5.13.19 return 0.

14. Check the kernel and you should get an output like the one below

```
usertest@usertest-vm:~$ dmesg
[  104.564295] [Ker-5.13.19] Hello world from a system call! OS_Project02!
[  104.564303] [Ker-5.13.19] My student id is : [12345]
usertest@usertest-vm:~$
```

At this point, we see that the same code (**syscallsHelloEco.c**) has different execution paths and produce different outputs, depending on the kernel we are currently using.

[Screenshot #16: Create a screenshot showing both outputs of terminal and dmesg when running the program using the Kernel 5.13.19]

Section 2.2: Basic system call – custom implementation [Do it yourself]

In this section you will make different system calls for different kernels, and one common C program will call them and, at the end, should have different execution outputs depending on the kernel you used.

1. In the files to download in E3, you will find two folders named **Kernel 4.19.148** and **Kernel 5.13.19**. The following is the **numericalTest.c** file from the **Kernel 4.19.148** folder.

```
#include <linux/syscalls.h>
#include <linux/kernel.h>

//STUDENT ID: 012345 (With leading 0)

int returnValue(int studentId, int a, int b){
      printk("[%d][Ker-4.19.148] syscalltest_returnIndividualValues : [%d][%d]\n",
studentId, a, b);
      return 0;
}

int minimum(int studentId, int a, int b, int c){
      //      Delete these lines, and insert your code here
      //      This function should return the minimum of three values: a,b,c, and output
      //            in kernel log a message of the form:
      //      [12345][Ker-4.19.148] syscalltest_minimum : [30][20][50] - [20]\n


}

int maximum(int studentId, int a, int b, int c){
      //      Delete these lines, and insert your code here
      //      This function should return the maximum of three values: a,b,c, and output
      //            in kernel log a message of the form:
      //      [12345][Ker-4.19.148] syscalltest_maximum : [30][20][50] - [50]\n


}

int displayDatatypes(int studentId) {
      //      Delete these lines, and insert your code here
      //      This function should return 0, and output in kernel log a message
      //            of the form:
      //      [12345][Ker-4.19.148] size of unsigned int : [%d] bytes\n
      //      [12345][Ker-4.19.148] size of signed int : [%d] bytes\n
      //      [12345][Ker-4.19.148] size of unsigned long : [%d] bytes\n
      //      [12345][Ker-4.19.148] size of signed long : [%d] bytes\n
      //      [12345][Ker-4.19.148] size of unsigned long long : [%d] bytes\n
      //      [12345][Ker-4.19.148] size of signed long long : [%d] bytes\n
      //      [12345][Ker-4.19.148] size of double : [%d] bytes\n
      //      [12345][Ker-4.19.148] size of char : [%d] bytes\n
      //      where you must get in runtime the value in bytes of the sizes of
      //            each datatype.


}
```

```
SYSCALL_DEFINE3(syscalltest_returnIndividualValues, int, studentId, int, a, int, b){
        return returnValue(studentId, a, b);
}

//      Delete these lines, and insert your code here
//      You must create the three system calls:
//      1. syscalltest_minimum
//              - Inputs: int studentId, int a, int b, int c
//              - Outputs: return the value obtained by the minimum function above.
//
//      2. syscalltest_maximum
//              - Inputs: int studentId, int a, int b, int c
//              - Outputs: return the value obtained by the maximum function above.
//
//      3. syscalltest_dataTypes
//              - Inputs: int studentId
//              - Outputs: return the value obtained by the displayDatatypes function
above.
```

And the following is the **numericalTest.c** file from the Kernel 5.13.19 folder

```
#include <linux/syscalls.h>
#include <linux/kernel.h>

int returnValue(int studentId, int a, int b){
        printk("[%d][Ker-5.13.19] syscalltest_returnIndividualValues : [%d][%d]\n", studentId,
a, b);
        return 0;
}

int addition(int studentId, int a, int b){
        //      Delete these lines, and insert your code here
        //      This function should return the addition of two values: a,b, and output
        //              in kernel log a message of the form:
        //      [12345][Ker-5.13.19] syscalltest_addition : [30][20][50]\n
}

int multiplication(int studentId, int a, int b){
        //      Delete these lines, and insert your code here
        //      This function should return the multiplication of two values: a,b, and
        //              output in kernel log a message of the form:
        //      [12345][Ker-5.13.19] syscalltest_multiplication : [30][20][600]\n
}

int displayDatatypes(int studentId) {
        //      Delete these lines, and insert your code here
        //      This function should return 0, and output in kernel log a message
        //              of the form:
        //      [12345][Ker-5.13.19] size of unsigned int : [%d] bytes\n
        //      [12345][Ker-5.13.19] size of signed int : [%d] bytes\n
        //      [12345][Ker-5.13.19] size of unsigned long : [%d] bytes\n
        //      [12345][Ker-5.13.19] size of signed long : [%d] bytes\n
        //      [12345][Ker-5.13.19] size of unsigned long long : [%d] bytes\n
        //      [12345][Ker-5.13.19] size of signed long long : [%d] bytes\n
        //      [12345][Ker-5.13.19] size of double : [%d] bytes\n
        //      [12345][Ker-5.13.19] size of char : [%d] bytes\n
        //      where you must get in runtime the value in bytes of the sizes of
        //              each datatype.
}
```

```
SYSCALL_DEFINE3(syscalltest_returnIndividualValues, int, studentId, int, a, int, b){
        return returnValue(studentId, a, b);
}

//      Delete these lines, and insert your code here
//      You must create the three system calls:
//      1. syscalltest_addition
//              - Inputs: int studentId, int a, int b, int c
//              - Outputs: return the value obtained by the minimum function above.
//
//      2. syscalltest_multiplication
//              - Inputs: int studentId, int a, int b, int c
//              - Outputs: return the value obtained by the maximum function above.
//
//      3. syscalltest_dataTypes
//              - Inputs: int studentId
//              - Outputs: return the value obtained by the displayDatatypes function
above.
```

Create a folder named **numericalTest** inside the **systemCallTests** folders in both kernels (Section 2.1 – step 1) and save each **numericalTest.c** in their respective kernel,.

2. Finish writing both **numericalTest.c** programs:
   a. For **Kernel 4.19.148**, create these four system calls:
      i. syscalltest_returnIndividualValues
      ii. syscalltest_minimum
      iii. syscalltest_maximum
      iv. syscalltest_dataTypes
   b. For **Kernel 5.13.19**, create these four system calls:
      i. syscalltest_returnIndividualValues
      ii. syscalltest_addition
      iii. syscalltest_multiplication
      iv. syscalltest_dataTypes

   as specified in the comments inside that file.

   [Screenshots #17-18: Create two screenshots showing the completed numericalTest.c for both kernels]

3. Complete all the needed steps to add these new system calls to both kernels. Remember to re-build your kernels.
   [Screenshots #19-20-21: Create 3 screenshots showing the modifications of the Makefile(s), syscall_64.tbl and syscalls.h for Kernel 4.19.148]

   [Screenshots #22-23-24: Create 3 screenshots showing the modifications of the Makefile(s), syscall_64.tbl and syscalls.h for Kernel 5.13.19]
   After re-building both kernels, restart your machine into **Kernel 4.19.148**.

4. Inside the **SystemCallsRunTests** folder (Section 2.1 – Step 9), create a folder named **numericalRuns.** Download the following **syscallsNumerical.c** program.

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

int main()
{
    int studentId = 12345;

    int a = 15;
    int b = 43;
    int c = 30;

    printf("a = [%d]\n",a);
    printf("b = [%d]\n",b);
    printf("c = [%d]\n",c);
    printf("studentId = [%d]\n",studentId);

    // Kernel 4.19.148

    printf("\n=== Kernel 4.19.148 ===\n");

    //Modify system IDs depending on your syscall_64.tbl
    printf("helloworld : %ld\n", syscall(548));
    printf("echo : %ld\n", syscall(549, studentId));

    printf("returnIndividualValues : %ld\n", syscall(550, studentId, a, b));
    printf("minimum : %ld\n", syscall(551, studentId, a, b, c));
    printf("maximum : %ld\n", syscall(552, studentId, a, b, c));
    printf("dataTypes : %ld\n", syscall(553, studentId));

    // Kernel 5.13.19
    printf("\n=== Kernel 5.13.19 ===\n");

    //Modify system IDs depending on your syscall_64.tbl
    printf("helloworld : %ld\n", syscall(554));
    printf("echo : %ld\n", syscall(555, studentId));

    printf("returnIndividualValues : %ld\n", syscall(556, studentId, a, b));
    printf("addition : %ld\n", syscall(557, studentId, a, b));
    printf("multiplication : %ld\n", syscall(558, studentId, a, b));
    printf("dataTypes : %ld\n", syscall(559, studentId));

    return 0;
}
```

When you run it inside **Kernel 4.19.148**, you should get an output like the one below

```
usertest@usertest-vm:~/Desktop/systemCallsRunTests/numericalRuns$ uname -r
4.19.148
usertest@usertest-vm:~/Desktop/systemCallsRunTests/numericalRuns$ ./syscallsNumerical
a = [15]
b = [43]
c = [30]
studentId = [12345]

=== Kernel 4.19.148 ===
helloworld : 0
echo : 0
returnIndividualValues : 0
minimum : 15
maximum : 43
dataTypes : 0

=== Kernel 5.13.19 ===
helloworld : -1
echo : -1
returnIndividualValues : -1
addition : -1
multiplication : -1
dataTypes : -1
usertest@usertest-vm:~/Desktop/systemCallsRunTests/numericalRuns$
```

And when you check the kernel ring buffer, you should get the following output

```
usertest@usertest-vm:~$ dmesg
[  289.153366] [Ker-4.19.148] Hello world from a system call! - OS_Project02!
[  289.153380] [Ker-4.19.148] My student id is : [12345]
[  289.153396] [12345][Ker-4.19.148] syscalltest_returnIndividualValues : [15][43]
[  289.153407] [12345][Ker-4.19.148] syscalltest_minimum : [15][43][30] - [15]
[  289.153448] [12345][Ker-4.19.148] syscalltest_maximum : [15][43][30] - [43]
[  289.153461] [12345][Ker-4.19.148] size of unsigned int : [4] bytes
[  289.153461] [12345][Ker-4.19.148] size of signed int : [4] bytes
[  289.153462] [12345][Ker-4.19.148] size of unsigned long : [8] bytes
[  289.153462] [12345][Ker-4.19.148] size of signed long : [8] bytes
[  289.153463] [12345][Ker-4.19.148] size of unsigned long long : [8] bytes
[  289.153463] [12345][Ker-4.19.148] size of signed long long : [8] bytes
[  289.153463] [12345][Ker-4.19.148] size of double : [8] bytes
[  289.153464] [12345][Ker-4.19.148] size of char : [1] bytes
usertest@usertest-vm:~$
```

[Screenshots #25-26: Create 2 screenshots showing the outputs in terminal and kernel ring buffer of the program under Kernel 4.19.148.]

5. Restart the machine using Kernel 5.13.19. Run the program again and you should get outputs like the ones below

```
usertest@usertest-vm:~/Desktop/systemCallsRunTests/numericalRuns$ uname -r
5.13.19
usertest@usertest-vm:~/Desktop/systemCallsRunTests/numericalRuns$ ./syscallsNume
rical
a = [15]
b = [43]
c = [30]
studentId = [12345]

=== Kernel 4.19.148 ===
helloworld : -1
echo : -1
returnIndividualValues : -1
minimum : -1
maximum : -1
dataTypes : -1

=== Kernel 5.13.19 ===
helloworld : 0
echo : 0
returnIndividualValues : 0
addition : 58
multiplication : 645
dataTypes : 0
usertest@usertest-vm:~/Desktop/systemCallsRunTests/numericalRuns$
```

```
usertest@usertest-vm:~$ dmesg
[35721.105053] [Ker-5.13.19] Hello world from a system call! OS_Project02!
[35721.105060] [Ker-5.13.19] My student id is : [12345]
[35721.105064] [12345][Ker-5.13.19] syscalltest_returnIndividualValues : [15][43]
[35721.105067] [12345][Ker-5.13.19] syscalltest_addition : [15][43][58]
[35721.105070] [12345][Ker-5.13.19] syscalltest_multiplication : [15][43][645]
[35721.105106] [12345][Ker-5.13.19] size of unsigned int : [4] bytes
[35721.105108] [12345][Ker-5.13.19] size of signed int : [4] bytes
[35721.105108] [12345][Ker-5.13.19] size of unsigned long : [8] bytes
[35721.105109] [12345][Ker-5.13.19] size of signed long : [8] bytes
[35721.105110] [12345][Ker-5.13.19] size of unsigned long long : [8] bytes
[35721.105110] [12345][Ker-5.13.19] size of signed long long : [8] bytes
[35721.105111] [12345][Ker-5.13.19] size of double : [8] bytes
[35721.105112] [12345][Ker-5.13.19] size of char : [1] bytes
usertest@usertest-vm:~$
```

[Screenshots #27-28: Create 2 screenshots showing the outputs in terminal and kernel ring buffer of the program under Kernel 5.13.19]

IMPORTANT NOTES:

1. **Besides your report and video**, you must upload the following files to E3:
   a. numericalTest.c
   b. syscallsNumerical.c
   c. syscall_64.tbl
   d. syscalls.h

   for both kernels. Use the following naming convention:

No.	Original File Name	Kernel 4.19.148	Kernel 5.13.19
1	numericalTest.c	K4_numericalTest.c	K5_numericalTest.c
2	syscall_64.tbl	K4_syscall_64.tbl	K5_syscall_64.tbl
3	syscalls.h	K4_syscalls.h	K5_syscalls.h
4	syscallsNumerical.c	This is a unique file. Use the correct system call IDs as we will compare against your syscall_64.tbl files.	

   Keep your code clean, properly indented and well commented.

2. In **numercialTest.c** and **syscallsNumerical.c**, the first line must be your student ID:
   `//STUDENT ID: 012345 (With leading 0)`
3. If you have VMWare-related, Linux-related or C-related questions, please check online. **No support will be given in the forum for these types of questions.**

Code grading criteria:

1. <u>Do not use any special libraries</u>. The automatic graders will run basic C and compile kernels using your code. Your grade will depend if your code can be compiled into a kernel.
2. <u>Do not change the functions signatures (function names, output types, inputs order and input names)</u>. The automatic graders will use regular expressions to split your code into individual functions.
3. <u>Check the spacings of your output strings (terminal and kernel ring buffer)</u>.
4. <u>You cannot hard-code the outputs of the system calls</u>. Random integers will be used to test your code. Do not hard-code the outputs of the datatypes' sizes (**syscalltest_dataTypes**). Search online how to get their sizes in runtime (hard-coding their values will get 0 points).