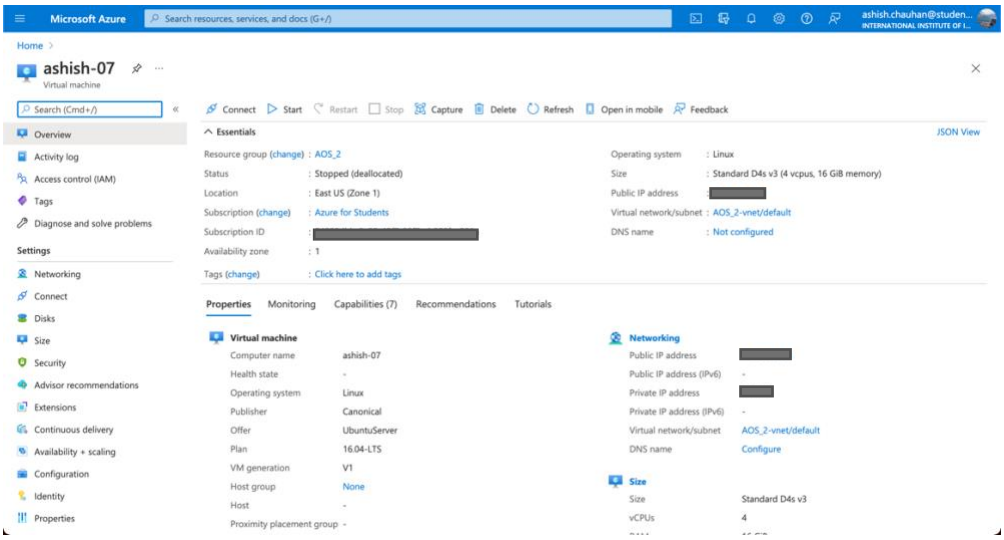
SYSTEM CALLS

1. SETTING UP VM

Since there was no compatible software available for M1 chip which was free and not many resources were available or provided so Virtual Machine was setup on Microsoft Azure.

A. Azure – Virtual Machine



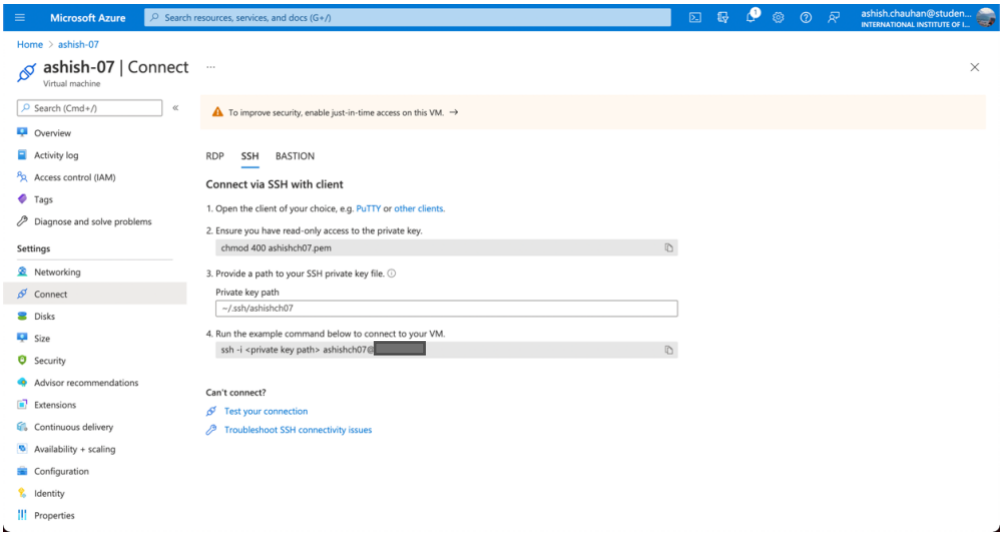
i. Above is the snapshot of the virtual machine that was setup for the purpose of creating and adding self-created system calls to the specified kernel.

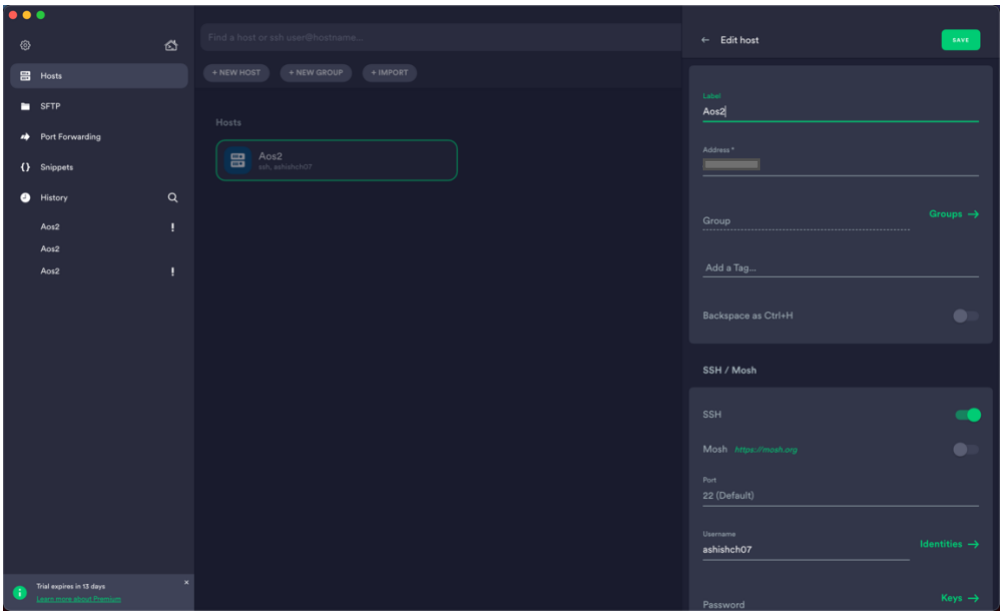
ii. Machine was setup with Linux operating system (Ubuntu 16.04-LTS) with 4 CPUs and 16 gigs of RAM.

B. Termius – SSH Client

i. To remotely access the virtual machine Termius was made use of.

ii. Host was created using the keys of azure vm and terminal solution/access was provided.





2. CONFIGUIRING THE VM

To make sure VM is fit to the task we need to perform, some needful packages must be installed. Following are the commands used in terminal to do the same:

- sudo apt install -y build-essential flex bison libssl-dev libelf-dev
- sudo apt-get update
- sudo apt-get install bc

A. Acquiring Source Code

After VM has been configured, next step is to download the specified kernel 4.19.210. Following commands in the terminal would download and extract the needed kernel:

- wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.19.210.tar.xz
- xz -v -d linux-4.19.210.tar.xz

- tar xvf linux-4.19.210.tar

Now we need to change the directory and enter the folder where kernel files reside.

B. Configuring the Kernel

We can disable or enable many of kernel's features, as well as set build parameters. We can simply configure it by copying our existing kernel's configuration into our .config file. This can be performed as follows:

- cp -v /boot/config-$(uname -r) .config

In the kernel configuration file we will find this line:

- CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-certs.pem"

Which is to be changed to –

- CONFIG_SYSTEM_TRUSTED_KEYS=""

3. ADDING THE SYSTEM CALLS

A. System Call Table

After the kernel has been compiled, we can start compiling it right away. Creating a system call requires editing a table (syscall_64.tbl) that is included by huge amount of code. Since compiling takes a lot of time, we would start writing system calls first.

- cd arch/x86/entry/syscalls/syscall_64.tbl

This table is read by scripts and used to generate some of the boilerplate code, which makes our lives a lot easier! Go to the bottom of the first and we'll add the following lines:

```
548     common  ashishhello            __x64_sys_ashishhello
549     common  ashishprint            __x64_sys_ashishprint
550     common  ashishprocess          __x64_sys_ashishprocess
551     common  ashishgetpid           __x64_sys_ashishgetpid
-- INSERT --                                                    392,48-63
```

The first column is the system call number. The second column says that this system call is common to both 32-bit and 64-bit CPUs. The third column is the name of the system call, and the fourth is the name of the function implementing it.

B. Adding Declarations

syscalls.h is the header file which contains the declarations for system calls. So, we need to add declarations of our system calls that we're going to create and save it.

- sudo vim include/linux/syscalls.h

Add the following lines at the end of the file and save the file:

```
asmlinkage int ashishhello(void);
asmlinkage int ashishprint(char*);
asmlinkage int ashishprocess(void);
asmlinkage int ashishgetpid(void);
#endif
                                                                1299,1
```

C. System Call Function

The last step is to write the function for the system call! We can implement system calls anywhere, but miscellaneous syscalls tend to go in the kernel directory, kernel/syscall.c . Following page shows the system calls that have been added to syscall_64.tbl and syscalls.h.

```
ashishch07@ashish-07:~/linux-4.19.210/kernel$ vim ashish
ashishgetpid.c   ashishhello.c    ashishprint.c    ashishprocess.c
```

i. ashishhello.c

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

SYSCALL_DEFINE0(ashishhello)
{
    printk("Hello World..!\n");
    return 0;
}
~
```

ii. ashishprint.c

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

SYSCALL_DEFINE1(ashishprint, char *, message)
{
  char buf[256];
  long copied = strncpy_from_user(buf, message, sizeof(buf));

  if (copied < 0 || copied == sizeof(buf))
    return -EFAULT;

  printk(KERN_INFO "Message -  \"%s\"\n", buf);

  return 0;
}
~
```

iii. ashishprocess.c

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

SYSCALL_DEFINE0(ashishprocess)
{
    printk("Parent Process ID : %d \n", current->parent->pid);
    printk("Current Process ID : %d \n", current->pid);

    return 0;
}
~
```

iv. ashishgetpid.c

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

SYSCALL_DEFINE0(ashishgetpid)
{
    return task_tgid_vnr(current);
}
~
```

SYSCALL_DEFINE<n> is a family of macros that make it easy to define a system call with 'n' arguments. The first argument to the macro is the name of the system call (without *sys_* prepended to it). The remaining arguments are type and name for the parameters.

printk is a C function from the Linux kernel interface that prints messages to the kernel log.

We use a handy *strncpy_from_user()* function which behaves like normal strncpy , but checks the user-space memory address first. If the string was too long or if there was a problem copying, we return EFAULT.

KERN_INFO is a macro that resolves to a string literal. The compiler concatenates that with the format string and *printk()* uses it to determine the log level.

task_tgid_vnr(current) is the helper function defined in sched.h which is used to get the pid of the process. Also, current->pid does the same job. Pid of the parent's process can be found using current->parent->pid.

4. COMPILE AND BOOT KERNEL

A. Add to Makefile

The systemcalls created need to be added to the Makefile of the kernel. For every syscall.c one entry of syscall.o goes into the Makefile. It has been as following:

```
# Makefile for the linux kernel.
#

obj-y     = fork.o exec_domain.o panic.o \
            cpu.o exit.o softirq.o resource.o \
            sysctl.o sysctl_binary.o capability.o ptrace.o user.o \
            signal.o sys.o umh.o workqueue.o pid.o task_work.o \
            extable.o params.o \
            kthread.o sys_ni.o nsproxy.o \
            notifier.o ksysfs.o cred.o reboot.o \
            async.o range.o smpboot.o ucount.o ashishhello.o ashishprint.o ashishprocess.o ashishgetpid.o
```

B. Prep using existing Kernel Config

Here we read from the old .config file that was used for the old kernel. "make olddefconfig" sets every option to their default value without asking interactively. It gets run automatically on make to ensure that the .config is consistent in case it was modified it manually.

```
ashishch07@ashish-07:~/linux-4.19.210$ sudo make olddefconfig
scripts/kconfig/conf  --olddefconfig Kconfig
#
# configuration written to .config
#
```

C. Building the Kernel

We start building the kernel using following command:

- sudo make -j5

here 5 is the (number of cores + 1) as we are using multicore. This is the most time taking process of all. So after we've compiled the kernel and all of its modules we need to follow certain commands to install the recently compiled modules and the kernel to make the changes.

- sudo make -j5 modules_install
- sudo make install

After this we need to restart our system :

- sudo reboot

D. Testing the System Calls

We have compiled and booted a kernel which has our own custom modifications made to it. Now to check and run our self-made system calls we need to create another .c file which would make use of these system calls.

Underlying code is snippet from execute.c which calls our system calls that we made and added to the kernel. It's making use of syscall.h file and calling the system calls by their system call number.

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

int main()
{
    printf("Q1 : %d\n", syscall(548));
    printf("Q2 : %d\n", syscall(549, "Parameter"));
    printf("Q3 : %d\n", syscall(550));
    printf("Q4 : %d\n", syscall(551));
    return 0;
}
~
```

Now we just need to compile this and run it to see the outputs and check if our system calls are working fine and providing the desired output.

- gcc execute.c
- ./a.out

```
ashishch07@ashish-07:~/linux-4.19.210$ ./a.out
Q1 : 0
Q2 : 0
Q3 : 0
Q4 : 24256
```

Each system call has executed and returned desired results. The 4th one returned the pid of the process as desired.
To check the outputs of our system calls we need to check the kernel logs where we've actually printed using *printk()* function.

- dmesg

```
[26773.400772] Hello World..!
[26773.400898] Message -  "Parameter"
[26773.400906] Parent Process ID : 24215
[26773.400907] Current Process ID : 24256
```

In the kernel logs, these final entries ensure that our system call have worked fine, and we get did achieve the desired goals. 🥳