# Adding a system call to Linux kernel, on Ubuntu 20.04

Credits: https://dev.to/jasper/adding-a-system-call-to-the-linux-kernel-5-8-1-in-ubuntu-20-04-lts-2ga8

Beware: This process takes many hours, and the “make” command consumes most part of it. If the process fails or if you make some mistakes, your virtual machine may become unbootable (but there is always a way to recover!).

# Section 1 - Preparation

*In this section, you will download all necessary tools to add a basic system call to the Linux kernel and run it. This is the only part of the entire process where network connectivity is necessary.*

**1.0 Install Ubuntu 20.04  and boot the system.**

Although these instructions should largely work for any GNU/Linux and any kernel, there are minor differences depending on OS and kernel version. Hence using Ubuntu 20.04 will be the safest for you.

**1.1 - Fully update your operating system.**

```
sudo apt update && sudo apt upgrade -y
```

**1.2 - Download and install the essential packages to compile kernels.**

```
sudo apt install build-essential libncurses-dev libssl-dev libelf-dev bison flex -y
```

If would rather use `vim` or any other text editor instead of `vim`, below is an example of how you install it.

```
sudo apt install vim -y
```

**1.3 - Clean up your installed packages.**

```
sudo apt clean && sudo apt autoremove -y
```

**1.4 - Download the source code of the latest stable version of the Linux kernel (*which is 5.11.9 as of 12 August 2020*) to your home folder.**

```
cd; wget -P ~/ https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.11.9.tar.xz
```

If you have downloaded a newer version of the Linux kernel, refer to this documentation to learn about any relevant change made to system calls.

**1.5 - Unpack the tarball you just downloaded to your home folder.**

```
cd; tar -xvf ~/linux-5.11.9.tar.xz
```

**1.6 - Reboot your computer.**

# Section 2 - Creation

*In this section, you will write a basic system call in C and integrate it into the new kernel.*

**2.1 - Check the version of your current kernel.**

```
uname -r
```

As of 12 August 2020, it should display the following.

`5.4.0-42-generic`

It may be different depending your computer, but do not worry about it.

**2.2 - Change your working directory to the root directory of the recently unpacked source code.**

```
cd ~/linux-5.11.9/
```

**2.3 - Create the directory for code of your system call.**

Decide a name for your system call, and keep it consistent from this point onwards. I have chosen `hello`.

```
mkdir hello
```

**2.4 - Create a C file for your system call.**

Create the C file with the following command.

```
vim hello/hello.c
```

Write the following code in it.

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

SYSCALL_DEFINE0(hello)

{
    printk("Hello, from inside Linux kernel\n");// ignore colouring here
    return 0;
}
```

You can write anything you like here.

Save it and exit the text editor.

**2.5 - Create a Makefile for your system call.**

Create the Makefile with the following command.

```
vim hello/Makefile
```

Write the following code in it.

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

Save it and exit the text editor.

**2.6 - Add the home directory of your system call to the main Makefile of the kernel.**

Open the Makefile with the following command.

```
vim Makefile
```

Search for `core-y`. In the second result, you will see a series of directories.

```
kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/
```

Add the home directory of your system call at the end like the following.

```
kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ hello/
```

Save it and exit the editor.

**2.7 - Add a corresponding function prototype for your system call to the header file of system calls.**

Open the header file with the following command.

```
vim include/linux/syscalls.h
```

Navigate to the bottom of it and write the following code just above `#endif`.

```
asmlinkage long sys_hello(void);
```

Save it and exit the editor.

**2.8 - Add your system call to the kernel's system call table.**

Open the table with the following command.

```
vim arch/x86/entry/syscalls/syscall_64.tbl
```

Navigate to the bottom of it. You will find a series of x32 system calls. Scroll to the section above it. This is the section of your interest. Add the following code at the end of this section respecting the chronology of the row as well as the format of the column. Use Tab instead of space.

```
442     common  hello               sys_hello
```

In the fresh source code of Linux 5.11.9 kernel, the number for your system call may be 442, but verify it once. If it’s different then just add +1 to last system call number.

Save it and exit the editor.

# Section 3 - Installation

*In this section, you will install the new kernel and prepare your operating system to boot into it.*

**3.1 - Configure the kernel.**

Make sure the window of your terminal is maximized.

Open the configuration window with the following command.

```
make menuconfig
```

Use **Tab** to move between options. Make no changes to keep it in default settings.

Save and exit.

This will create the file .config in your kernel code directory. See the contents of this file. It’s a listing of various kernel configuration parameters. Setting each of them to a proper value will

enable or disable particular code in the kernel. Almost each of these parameters is passed as a option to “gcc -D”. Read more about “gcc -D”.

**NOTE:** Many times the kernel does not compile due to issues in the specification of the configuration. The “Make Menuconfig” creates a “default” configuration, that may not be suitable for your computer.

You may try copying the “config” file for your computer from /boot

*Run the following command when your CWD is the folder of the kernel code.*

```
$ cp /boot/config-`uname -r` ./.config
```

*# For example this does copy /boot/config-5.11.0-46-generic on Abhijit’s computer.*

**3.2 - Find out how many logical cores you have.**

```
nproc
```

The following few commands require a long time to be executed. Parallel processing will greatly speed them up. If it’s 12, then use 12 as argument to make in subsequent commands. If it’s 1, then you may still use a number more than 1, but do not choose a very large number as it does not help at all. Upto 4 should be ok.

**3.3 - Compile the kernel's source code.**

```
make -j12
```

At the end of this command, you will see the file `vmlinux` created in the kernel code folder. This is the Linux kernel’s machine code (binary) file, uncompressed. `vmlinuz` is `vmlinux` compressd. Note that this file is not an executable file like a normal programme! So no point in running “./vmlinux”. But I’m sure you will try that :-;

**3.4 – Install various kernel modules**

```
sudo make modules_install -j12
```

The above will copy different kernel modules (.ko files, kernel objects) to various locations on your computer.

**3.5 - Install the kernel.**

```
sudo make install -j12
```

This will copy the `vmlinuz` file to /boot and create initial ram disk image (initrd.img) file and also update the configuration file of grub2 boot loader.

**3.6 - Update the bootloader of the operating system with the new kernel.**

```
sudo update-grub
```

This step is really redundant, as update-grub was already done as part of the last step!

**3.7 - Reboot your computer.**

```
sudo reboot
```

# Section 4 - Result

*In this section, you will write a C program to check whether your system call works or not. After that, you will see your system call in action.*

**4.1 - Check the version of your current kernel.**

```
uname -r
```

It should display the following.

```
5.11.9
```

**4.2 - Change your working directory to your home directory.**

```
cd ~
```

**4.3 - Create a C file to test  your system call.**

Create the C file with the following command.

```
vim hello.c
```

Write the following code in it.

```
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define __NR_hello 442

long hello_syscall(void)
{
    return syscall(__NR_hello);
}

int main(int argc, char *argv[])
{
    long activity;
    activity = hello_syscall();

    if(activity < 0)     {
        perror("System call failed\n");
    }

    else     {
        printf("System call worked!\n"); // ignore colouring here
    }

    return 0;
}
```

You can customize the messages for failure and success anyhow you like.

Save it and exit the editor.

**4.4 - Compile the C file you just created.**

```
gcc -o hello hello.c
```

**4.5 - Run the C file you just compiled.**

```
./hello
```

**If it displays the following, everything is working as intended.**

```
System call worked!
```

**4.6 - Check the last line of the `dmesg` output.**

```
dmesg
```

At the bottom, you should now see the following.

```
Hello, from inside Linux kernel\n
```

**4.7 – Done!**

You have successfully added a hello-world type of system call to Linux kernel. To Add more meaningful system calls, you need to study the Linux kernel code, data structures and write a meaningful code inside your system calls.

## Possible errors and ways to deal with them

**(1)Problem**

make[1]: *** No rule to make target 'debian/canonical-certs.pem', needed by 'certs/x509_certificate_list'.  Stop.

**Solution See: https://stackoverflow.com/questions/67670169/compiling-kernel-gives-error-no-rule-to-make-target-debian-certs-debian-uefi-ce**

**(2) Problem**

The I/O cache encountered an error while updating data in medium "ahci-0-0" (rc=VERR_DISK_FULL). Make sure there is enough free space on the disk and that the disk is working properly. Operation can be resumed afterwards.

| | |
|---|---|
| Error ID: | BLKCACHE_IOERR |
| Severity: | Non-Fatal Error |

**Solution**

In the Ubuntu VM shared by abhijit, do the following

```
$ sudo rm -f  /var/cache/apt/archives/*deb
```

*if you run*

```
$ df -h /
```

then you will see that the disk of Abhijit's VM is only 18 GB.

To incresae the size of this disk do the following:

1) Shut down the Ubuntu VM

2) In VirtualBox Manager, go to file->Virtual Media Manager , and locate the Ubuntu VDI File.

Increase the size to, say, 50 GB.

3) Now, boot the Virtual Machine.

4) Now if you run

```
$ df -h /
```

You will see it to be 18 GB, even now!

But

```
$ cat /proc/partitions  | grep sda
   8        0   51943040 sda
Shows that disk has grown to 50GB.
```

This means disk is bigger, but the “on disk data sturcture” that is file-system is not big!

5) run

```
$ sudo gparted /dev/sda
```

and create a new partition in the free space. It should become /*dev*/sda4

```
6) sudo vgextend vgubuntu /dev/sda4
7)  sudo lvextend -L +30G vgubuntu/root
8) sudo resize2fs /dev/mapper/vgubuntu-root
```

9) Now

$ df -h /

will show 48 GB disk.

**(3) Problem**

**BTF: .tmp_vmlinux.btf: pahole (pahole) is not available**

**Solution**

```
$ sudo apt install dwarves
```

**(4) Problem**

Solution

(5)