Setting-up Kernel Development Host

1. Setting-up Kernel Development Host

a. Introduction

Although originally developed first for 32-bit x86-based PCs (386 or higher), today Linux also runs on (at least) the Compaq Alpha AXP, Sun SPARC and UltraSPARC, Motorola 68000, PowerPC, PowerPC64, ARM, Hitachi SuperH, Cell, IBM S/390, MIPS, HP PA-RISC, Intel IA-64, DEC VAX, AMD x86-64, AXIS CRIS, Xtensa, Tilera TILE, AVR32 and Renesas M32R architectures.

Kernel Development is bit different from usual application development like you are building a specific application in unique language for example java, C sharp, C++, C etc. You just learn the language then implement the idea. Before building any service for kernel or device driver for a peripheral, kernel development requires that you have some skills, additional to language. You must know the overall structure of kernel, kernel system call interface, kernel APIs and underlying hardware. You must know how to build your own customize kernel, how to debug kernel oops (kernel exceptions), kernel source code hierarchy that where you put your own custom code. Kernel is the core part of Operating System, so these are the basic requirements for kernel development because without it you may crash your system.

b. Linux Kernel

The history of Linux began in 1991 with the commencement of a personal project by a Finnish student, Linus Torvalds when he was studying at University of Helsinki, he was using a version of UNIX operating system called “Minix”. Linus and other users sent requests for modifications and improvements to Minix's creator, Andrew Tanenbaum, but he felt that they weren't necessary.

That's when Linus decided to create his own operating system that would take into account users comments and suggestions for improvements. What is kernel

basically? Without going into detail, kernel tells the big chip that controls your computer to do what you want the program that you're using to do. Without a kernel, operating system does not exist. Linux kernel has received contributors from thousands of programmers. Linux rapidly accumulated developers and users who adapted code from other free software projects for use with the new operating system. These people started to offer their help. The version numbers of Linux getting higher and higher. Developers began writing drivers for different video cards, sound cards and other gadgets inside and outside your computer could use Linux. As per the statistics of 2009 the Linux Kernel comprised of 11,560,000 LOCs in about 27,900 files. About 10,900 LOCs were added in Linux kernel every day, 5,500 LOCs were removed from Linux kernel every day, and 2200 LOCs were modified in Linux kernel every day.

Linux, at first, was not for everybody. Later on, companies like Red Hat made it their goal to bring Linux to the point where it could be installed just like any other operating system; by anyone who can follow a set of simple instructions, and they have succeeded. Today, Linux can be installed on a home PC as well as a network server for a fraction of the cost of other companies' software packages. Its main reason is to provide facilities to the user and developer so that more and more people could interact with this free open source project. Linux has proven to be a tremendously stable and versatile operating system, particularly as a network server.

c. Linux Kernel Versions

Linux Kernel comes in two flavors: Stable and development. Stable kernel is suitable for development and released as product. On the other hand, development kernel is where new features are tested and solutions provided. Linux kernel has numbering scheme to distinguish between stable and development kernel. With time the meaning of this scheme changed as Linus uses it. However, at the time of this writing 5.3.7.1 is the latest Linux kernel. Where first number represents the Kernel version, second is major revision, third is minor revision and the optional fourth number is patch number. The latest stable Linux kernel at the time of this writing is 5.3. You can download source from `https://www.kernel.org`

d. Linux Distributions

Most of the Linux distributions are GNU/Linux. It means the Linux kernel bundled with numerous utilities, programs, packages, libraries, tools from GNU to form a complete Operating System. Linux just refers to kernel (a core part of Operating System). Orderly packaging of different utilities, programs or tools makes different distributions like Ubuntu, Kali, Gentoo, Slackware, Red Hat, Fedora, OpenSUSE, Debian, CentOS etc. The Operating System that we are using is Ubuntu 14.0, having Linux kernel. It is installed in our virtual environment (virtual box).

e. Online Resources

www.kernel.org

The main repository for Linux kernel is www.kernel.org. Here you can find all released kernel versions.

www.lkml.org

The Linux Kernel Mailing List (LKML) is the forum where developers debate on design issues and decide on future features. You can find a real-time feed of the mailing list at www.lkml.org. LKML acts as the thread that ties all these developers together.

www.lwn.net

lwn stands for Linux weekly news. It is the place where you find the latest news from the Linux kernel developer community.

www.lxr.free-electrons.com

lxr stands for Linux Cross Reference. It is a web-based indexer of Linux kernel source code. You can find any function and identifier definition in the source code by just one click.

2. The architecture of Linux kernel source

The root of the kernel source code consists of following folders. Students are suggested to download the Linux kernel and spend some time having a birds eyeview of the contents of all these folders.

`arch (16.3%)`	It contains all of the architecture specific kernel code (30 dirs). It has further subdirectories, one per supported architecture, e.g., i386, x86_64, alpha, mips, sparc, powerpc, arm, arm64, blackfin…
`block (0.1%)`	This folder holds code for block-device drivers (46 files). Block devices are devices that accept and send data in blocks. Data blocks are chunks of data instead of a continual stream.
`crypto(0.4%)`	This folder contains the source code for many encryption algorithms.
`documentation (2.8%)`	This folder contains plain-text documents that provide information on the kernel and many of the files. If a developer needs information, they may be able to find the needed information in here. Also available on http://kernel.org/doc
`drivers(57%)`	This directory contains the code for the drivers, a software that controls a piece of hardware. It contains subdirectories like block, char, cdrom, pci, scsi, net, sount, …
`fs (5.5%)`	This is the FileSystem folder. All of the code needed to understand and use filesystems is here.
`include (3.5%)`	This directory contains most of the include files needed to build the kernel code. It too has further subdirectories including one for every architecture supported.
`init`	This directory contains the initialization code for the kernel and it is a very good place to start looking at how the kernel work. For example, on

	an intel based system, the kernel starts when grub has loaded the kernel into memory and passed control to it, after doing some architecture specific things in /arch/, it jumps to the main() routine in init/main.c
ipc	IPC stands for Inter-Process Communication. This folder has the code that handles the communication layer between the kernel and processes
kernel (1.2%)	The code in this folder controls the kernel itself.
lib(0.5%)	This directory contains the kernel's library code. The architecture specific library code can be found in arch/*/lib/.
mm (0.5%)	This directory contains all of the memory management code. The architecture specific mm management code lives down in arch/*/mm/, for example the page fault handling code is in mm/memory.c and memory mapping and page cache code is in mm/filemap.c. The buffer cache is implemented in mm/buffer.c and the swap cache in mm/swap_state.c and mm/swapfile.c
Scripts (0.4%)	This folder has the scripts needed for compiling the kernel. It is best to not change anything in this folder. Otherwise, you may not be able to configure or make a kernel.
net (4.3%)	The network folder contains the code for network protocols. This includes code for IPv6 and protocols for Ethernet, wifi, bluetooth, etc.
Virt (0.1%)	This folder contains code for virtualization, which allows users to run multiple operating systems at once.
Usr (0.1%)	The code in this folder creates those files after the kernel is compiled.
security (0.3%)	This folder has the code for the security of the kernel. It is important to protect the kernel from computer viruses and hackers. Otherwise, the Linux system can be damaged.

Linux Kernel Compilation

1. Preparing your work environment

If you know that how to build a custom kernel it means you can instruct your kernel that how it will act or react. You can add your own new system call or a service. You can define your own Heap manager, your own scheduler scheme, your own memory management algorithms. You can optimize the kernel by removing useless drivers to speed up boot time. Create a monolithin instead of a modularized kernel. You can add a new hardware support. Last but not the least, it gives you an inside look and you get familiar with different source files, parameters and overall structure of the kernel.

I am currently running Kali Linux 4.19.0 (guest OS with 4GB RAM) in Virtual Box 5.2.26 on my Apple machine (2.9 GHz Intel core i7 with 16 GB DDR3 RAM) running Mac OS X Mojave 10.14.1 (host OS).

2. Packages you need

For Kernel compilation you need to have these packages installed on your system, namely `gcc`, `make`, `bc`, `build-essential`, `libelf-dev`, `libssl-dev`, `bison`, `flex`, `initramfs-tools`, `git-core`, and `libncurses5-dev`. To install these packages, use **Advanced Packaging Tool** available on all Debian distributions. *(Do view the file `/etc/apt/sources.list` which contains the names and links of packages to install)* The general syntax for installing a package using apt is:

```
$ sudo apt-get install <space separated packageName(s)>
```

Before installing the packages, first `update` apt and after installation `upgrade`.

```
$ sudo apt-get update

$ sudo apt-get install gcc

$ sudo apt-get install build-essential

- - -

$ sudo apt-get upgrade
```

Note: To check which package contains what all commands or to check which shell command become available after installation of which package you can use dpkg command (Debian package manager)

$ which ls

```
/bin/ls
```

$ dpkg –s /bin/ls or **(apt-file search /bin/ls)**

```
coreutils: /bin/ls
```

$ dpkg –S coreutils or **(apt show coreutils or apt-cache show coreutils)**

```
Package: coreutils

Installed-Size: 6,164 kB

Original-Maintainer: Michael Stone <mstone@debian.org>

Version: 8.21-1ubuntu5.1

Replaces: mktemp, timeout

Pre-Depends: libacl1 (>= 2.2.51-8), libattr1 (>= 1:2.4.46-8),
libc6 (>= 2.17), libselinux1 (>= 1.32)

Homepage: http://gnu.org/software/coreutils

Bugs: https://bugs.launchpad.net/ubuntu/+filebug

Description: GNU core utilities. This package contains the
basic file, shell and text manipulation utilities which are
expected to exist on every operating system. Specifically,
this package includes:  cat chgrp chmod chown chroot cp cut
date dd df dir dirname du echo env expr false flock groups
head hostid id join link ln logname ls mkdir mkfifo mknod
mktemp mv nice nl od paste pathchk printenv printf pwd
readlink rm rmdir seq sleep sort split stat stty sum sync tail
tee test timeout touch tr true truncate tty uname uniq unlink
users wc who whoami yes
```

3. View the existing Kernel Information

Before proceeding, lets visit the existing kernel components on your system one by one

3.1 Check the details of existing kernel version by following command:

```
$ uname -a
Linux kali 4.19.0-kali1-amd64 #1 SMP Debian 4.19.13-1kali1
(2019-01-03) x86_64 GNU/Linux
$ uname -r
4 .19.0-kali1-amd64
```

3.2 There is quite a useful stuff lying in the `/boot` directory. The four files of our interest are:

- **vmlinuz:** The compressed, bootable kernel image. On Linux systems, vmlinuz is a statically linked executable file that contains the Linux kernel in one of the object file formats supported by Linux.The letter z at the end denotes that it is compressed.
- **initrd.img:** The initial RAM disk; an ASCII cpio archive, an early root filesystem that allows your kernel to bootsrap and get essential device drivers to get the final, official root filesystem
- **config:** An ASCII text file that contains the configuration parameters for the kernel
- **System.map:** An ASCII text file that contains Symbol table used by kernel

  ```
  $ ls /boot/
  vmlinuz-4.19.0-kali1-amd64
  initrd.img-4.19.0-kali1-amd64
  config-4.19.0-kali1-amd64
  System.map-4.19.0-kali1-amd64
  ```

  Each kernel installed on your system will have a corresponding directory of loadable modules to go with it. This is what it looks like on my system at the moment:

```
$ls /lib/modules
3.13.0-43-generic
```

3.3 Last essential thing in booting a new kernel is that you need to add an entry for your new kernel into the GRUB boot loader configuration file on your system. On Ubuntu, it is `/boot/grub/grub.cfg`. View the contents of this file. Remember once you are done installing the new kernel, you have to run the appropriate utility to add entries for it to your GRUB config file; otherwise, GRUB will never know about it and you'll never be able to boot it.

4. Steps in Installation of Linux Kernel

- Download the kernel source
- Configure the kernel
- Compile/Build the kernel
- Install the kernel
- Update boot loader and reboot
- Verify your installation

Note: *This document must be read in conjunction with the README file that you can find in the toplevel kernel source directory.*

4.1 **STEP-I:** Download Linux Kernel Source Code

To download the source code you can visit the `www.kernel.org` web site and get the stable kernel release. You can also use following command on the shell to download the current linux kernel repository into your pwd, by cloning its reference git tree, the one managed by Linus Torvald either using the git or the http protocol.

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

$ git clone http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

I have downloaded `linux-5.3.7.tar.xz` from `kernel.org` in my `~/Download/` directory. The size of tar ball is 104 MiB

```
$ tar xvf linux-5.3.7.tar.xz
```

It will create a directory named `linux-5.3.7` and extract all files to it. Now change your current working directory to your extracted source directory

```
$ cd ~/Download/linux-5.3.7
```

```
$ ls
```

```
arch       CREDITS   drivers   include   Kbuild   lib   mm   tools
usr        REPORTING-BUGS   securit block      crypto    firmware
init       Kconfig   MAINTAINERS   net samples sound virt COPYING
Documentation    fs    ipc    kernel Makefile README scripts
```

To get a view of the size of the linux kernel you can run following commands (if you have cloned the kernel from a git repository), which tells you the number of files in the Linux kernel source:

```
$ git ls-files | wc -l
```

```
65695
```

Do go through `Documentation` directory. For example, a must read is the `Documentation/process/Codinng.rst` file, which contains Linux Kernel Coding Style. One should also install the `cscope` tool to examine the Kernel code. `cscope` is an interactive, screen-oriented tool that allows the user to browse through C source files for specified elements of code.

Note: Now for the rest of this tutorial my current working directory will be `linux-5.3.7` and from now on-wards I will use it as command prompt in all commands below. But before proceeding execute the following command in the top-level kernel source directory to get an insight of different targets in the top level Makefile

```
$ make help  | less
```

```
Cleaning targets:
clean - Remove most generated files, keeping config and enough build support
to build external modules
mrproper  - Remove all generated files + config + various backup files
distclean  - mrproper + remove editor backup and patch files
Configuration targets:
```

```
config          - Update current config utilising a line-oriented program
menuconfig      - Update current config utilising a menu based program
xconfig         - Update current config utilising a QT based front-end
gconfig         - Update current config utilising a GTK based front-end
oldconfig       - Update current config utilising a provided .config as base
localmodconfig  - Update current config disabling modules not loaded
silentoldconfig - Same as oldconfig, but quietly, additionally update deps
defconfig       - New config with default from ARCH supplied defconfig
savedefconfig   - Save current config as ./defconfig (minimal config)
allnoconfig     - New config where all options are answered with no
allyesconfig    - New config where all options are accepted with yes
randconfig      - New config with random answer to all options
tinyconfig      - Configure the tiniest possible kernel
Other generic targets:
all             - Build all targets marked with [*]
- - -
```

4.2 **STEP-II:** Configure the Kernel

Configuring the kernel involves selelcting which features you want to built into the kernel image, which features you want to built as loadable modules, and which features you want to omit entirely. The configure process will finally create a `.config` script for building the kernel. The `.config` file contains the configuration information (which has information about the features to be installed) of the kernel to be compiled. There are different methods to configure Linux kernel, few of them are:

- `make config` (Command Line configuration method)

  It makes sure all of the dependencies for the rest of the build and install process **are** available, and finds out whatever it needs to know to use those dependencies

- `make oldconfig` (Recommended for beginners)

  Make oldconfig takes the .config and runs it through the rules of the Kconfig files and produces a .config which is consistant with the Kconfig rules.

- `make menuconfig` (Based on libncurses5-dev package)

  It starts a terminal-oriented configurationtool

- `make xconfig` (GNOME toolkit interface)

  xconfig is short for the '***xconfig***' target for the Linux Makefile. It is a graphical Linux compilation utility, which uses Qt

- `make localmodconfig`

  compile only the loaded modules in the running kernel

Option 1

The best configuration method for beginners (who do not know what a specific parameter will actually do?) is to use your current kernel configuration file in the `/boot/` directory. Just copy the configuration file from `/boot/` folder to top of Linux source folder with "`.config`" name, and then run the `make oldconfig` command.

```
$ cp /boot/config-3.13.0-43-generic     ./.config
```

`$ yes '' | make oldconfig` (two single quotes, no space in between)

Without piping, the above command generates configuration file for your kernel but again ask you to enter right choice for some parameters. This is because of the configuration file you use is not up-to-date and make ask you to enter choices for new parameters. If you know, you enter the choice but if you are beginner you simply hit the ENTER to take the default option.

Option 2

To ensure we only compile the modules that are currently running on the kernel we use

```
$ yes '' | make localmodconfig
```

Using this option, the compilation time of the kernel is reduced to a great extent (from around 5 hours to 1 hour or may be less) as a lot of the modules in the configuration file are usually not running. This only compiles the currently running modules make the compilation process a lot quicker.

After this step is completed (using any of the above two options) you can see two new files in the current directory `.config` and `.config.old`. Must view the file for different configuration parameters.

$ less .config

```
# Automatically generated file; DO NOT EDIT.
# Linux/x86 5.3.7 Kernel Configuration
# Compiler: gcc (Debian 9.2.1-8) 9.2.1 20190909
CONFIG_CC_IS_GCC=y
---------------
---------------
# end of Kernel hacking
```

Your new Kernel needs to be somehow distinguishable from all of the other possible kernels you can boot, and that identification comes from the first few lines of the top level Makefile in the source tree. Open Makefile in vim, the EXTRAVERSION appears blank, write -pu1 , save and exit.

$ vim Makefile

```
# SPDX-License-Identifier: GPL-2.0
VERSION = 5
PATCHLEVEL = 3
SUBLEVEL = 7
EXTRAVERSION = -pu1
```

This is to have a unique kernel identifier for this kernel, I have written -pu1 over here. Later if I will be building the same source with a bit of different functionality, I could write something else there. With above data the `uname -r` command after the installation will show us `5.3.7-pu1`. However, even now, i.e. before build process you can check the kernel release by giving the following command:

$ make kernelrelease

```
5.3.7-pu1
```

4.3 **STEP-III:** Compile/Build the Kernel

The compilation involves two things, first generate compress bootable kernel image and second compile modules that are necessary for kernel working. You can do both the steps separately

```
$ make –jn bzImage
```

```
$ make –jn modules
```

Or you can give a single command to build

```
$ time make –jn
```

The n after j tells how many cores the program is allowed to use. (I have used 4). As mentioned above Machine 1 has 2 cores and machine 2 has 4 cores. So n can be replaced by no. of cores you wish to give to the process. The time command tells the time took by the program for completion.

If you have used option 1 (`make oldconfig`) to create the .config file, the build time taken will be around 5 to 6 hours. I have used option 2 (`make localmodconfig`) to create the .config file, the build time drastically reduced to just 28 minutes ☺.

Afte the build process is complete, run the `ls` command in the `linux-5.3.7` directory and note the new files and directories created after the build process. I have highlighted in red.

```
$ ls
```

```
arch        certs      crypto            fs          ipc         kernel
MAINTAINERS     samples    sound         usr  COPYING  Documentation
include  lib        Makefile        net     scripts      virt README
security    tool  block         CREDITS    drivers              init
LICENSES  mm  Kconfig   Kbuild
```

modules.builtin modules.builtin.modinfo Module.symvers
modules.order System.map vmlinux.o vmlinux

4.4 **STEP-IV:** Install the Kernel

If you have been successful in building the kernel, now it is the time to install it. This is the only step where you need root/sudoers privileges.

```
$ sudo make   modules_install
```

It is always better to strip the modules before you install them. So following command is better than above as it will reduce the size of the modules before installing them:

```
$ sudo make   INSTALL_MOD_STRIP=1    modules_install
```

This step will take just a second and create a new directory `/lib/modules/5.3.7-pu1/` and copy all the .ko files (modules) over there. You can check the contents of that directory

```
$ ls /lib/modules/5.3.7-pu1
```

```
Build     kernel     modules.alias.bin     modules.builtin.modinfo
modules.devname              modules.symbols       modules.builtin
modules.dep              modules.order    modules.symbols.bin
modules.alias         modules.builtin.bin         modules.dep.bin
modules.softdep  source
```

After you have installed the modules, now is the time to install the kernel, by giving the following command:

```
$ sudo make install
```

The install section of Makefile will move the files to their destination locations mentioned in the DIR variables (BIN_DIR, MAN_DIR, BIN_DIR_D, …). Instead of using mv or cp command the install target of Makefile use linux install command that not only moves files but also change permissions to those files. The install target of Makefile will mainly perform following four steps:

- `sudo cp ./arch/x86/boot/bzImage   /boot/vmlinuz-5.3.7-pu1`
- `sudo cp ./.config  /boot/ config-5.3.7-pu1`
- `sudo cp ./System.map /boot/System.map-5.3.7-pu1`
- `sudo mkinitramfs –o initrd.img-5.3.7-pu1`

Note: Building in an alternate directory:

Until now, all of your configuration and compilation has been done in the same directory as the kernel source itself, which is fine for most people but, in some cases, it's more convenient to preserve the kernel source untouched and have all the configuration output and compilation results generated in a separate directory. To do this, from the top of the kernel source tree, you can do something resembling to the following:

```
$ mkdir ../build_dir
$ cp /boot/config-5.7.0-43-generic     ./.config
$ yes '' | make O=../build_dir/  localmodconfig
$ make           O=../build_dir/    -j4
$ make           O=../build_dir/    modules_install
$ make           O=../build_dir/    install
```

The advantages of above are manifolds. Firstly, this will leave the source unpolluted by all of those output files, which makes it easier if you want to search the tree using something like grep. Secondly, it allows you to work with a directory of kernel source for which you have no write access. Finally, this feature lets you work with multiple configurations and builds simultaneously, since you can simply switch from one output directory to another on the fly, using the same kernel source directory as the basis for all those builds. However, you must specify that output directory on every make invocation. In short, this feture is meant to be used with a pristine kernel source tree.

4.5 **STEP-V:** Update Bootloader and Reboot

Now there are two kernels, i.e., `vmlinuz-4.19.0` and `vmlinuz5.3.7` in the `/boot/` directory. See the contents of the file `/boot/grub/grub.cfg` and you can see new entries in that file for the newly installed kernel. Now open the file `/etc/default/grub` in vim and change the `GRUB_TIMEOUT` value from 5 to 30. Moreover, let the `GRUB_DEFAULT` value to 0, which means by default boot from the first entry of the kernel in the `/boot/grub/grub.cfg` file, which will be the newly installed kernel. After making these changes you need to update the boot loader use the following command:

```
$ sudo update-grub2
```

This will update GRUB with the new kernel. And if it is the latest kernel version then it will become the default to be loaded when rebooting. Otherwise, we need to select the kernel to run during the booting process explicitly. To open the boot loader, restart the system, during system startup hold the shift key and you will see a screen with all installed kernels.

```
$ sudo systemctl reboot
```

4.6 Verify your Installation

After your system is up and running again view following, to ensure that all the files after installation of new kernel on your system are at right place and working correctly:

4.6.1 Check the currently running kernel version

```
$ uname -r
5.3.7-pu1
```

4.6.2 Check the new versions of following four files in the `/boot` directory.

```
$ ls /boot/
initrd.img-5.3.7-pu1
System.map-5.3.7-pu1
config-5.3.7-pu1
vmlinuz-5.3.7-pu1
```

4.6.3 Also see the modules directory. This is what it looks like on my system after the installation:

```
$ ls  /lib/modules
4.19.0-kali1-amd64     5.3.7-pu1
```

4.6.4 View the contents of `/boot/grub/grub.cfg`. Note the new kernel entry:

```
$ less /boot/grub/grub.cfg

……
```

4.6.5 Last optional step is to clean your system. This is important if you want to play again

```
$ make clean
```

Above target will remove most generated files, keeping config and enough build support to build external modules

```
$make mrproper
```

The mrproper target will remove all generated files as well as the config file and various backup files.

```
$make distclean
```

The distclean target will do mrproper + remove editor backup and patch files. This takes you back to literally the pristine, distribution version of the kernel source. This is a good option if you want to play again, on the same source tree.

If you think your installation is a success you can always remove the kernel source folder ***provided if you have not build your kernel in the same directory*.**

```
$ rm –rf linux-5.3.7
```

To repeat, you should delete all related files under the `/boot/` directory and the corresponding subdirectory in the `/lib/modules/` directory. Then update the grub and finally reboot.

Adding a System Call in Linux Kernel

Let us create a plus system call and add its code to the kernel. The major steps involved are:

I. Write `plus.c` file in the `kernel/` directory
II. Add ptototype of system call and and system call ID in appropriate files
III. Update the `kernel/Makefile`
IV. Configure, Compile, install the kernel
V. Test your system call using `syscall()`
VI. Write wrapper of your system call and again test it.
VII. Write man page

Step-1: Create source File

- For Linux Kernel 5.3.7

  Create a new directory name Custom_Syscall

```
$ mkdir Custom_Syscall
$ vim Custom_Syscall/plus.c
#include <linux/kernel.h>
#include <linux/syscalls.h>
SYSCALL_DEFINE2(sys_plus, long, a, long, b)
{ return a + b; }
```

  `SYSCALL_DEFINE2` macro is used because this system call has 2 parameters. sys_plus in the name of the system call. This system call will add the two numbers and return it.

- For Linux Kernel 2.6, 3.10 and 4.2

```
$ vim kernel/plus.c
#include <linux/kernel.h>
#include <linux/syscalls.h>
asmlinkage long sys_plus(long a, long b)
{ return a + b; }
```

  The `asmlinkage` is a macro that is used to override the default conventions on parameter passing. It tells your compiler that the function expects all of its arguments to be on the CPU stack instead of registers. All system calls are marked with the `asmlinkage` tag.

Step-2: Add prototype and System Call ID in Syscall Table

- For Linux Kernel 5.3

  Add prototype of system call in the end of following file:

```
$ vim + include/linux/syscalls.h
asmlinkage long sys_plus (long, long);
```

  Now open the `syscall_64.tbl` and put syscall number, abi, name and entry point at the last row of the file. Syscall number should be unique on the kernel.

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

```
<number>     <abi>      <name>      <entry point>
436          common      plus     __x64_sys_plus
```

  Evey systemcall has a unique number. In order to call a syscall, we tell the kernel to call the syscall by its number rather than by its name. The fourth field (entry point) is the name of the function to call in order to handle the syscall. The naming convention for this function is the name of the syscall prefixed with __x64_sys_.

  All system calls are identified by a unique number. In order to call a syscal, we tell the krenel to call the syscall by its number rather than by its name. The Application binary interface is normally i386 for 32 bit and 64 for 64 bit OS. The third field is simply the name of the syscall. The fourth field (entry point) is the name of the function to call in order to handle the syscall. The naming convention for this function is the name of the syscall prefixed with sys_.

- For Linux Kernel 2.6

  Add prototype of system call (`plus.c`) in `include/linux/syscalls.h`

```
asmlinkage long sys_plus(long,long);
```

  Add following line in `arch/x86/kernel/syscall_table_32.S`

```
.long sys_plus
```

  Add following line in `arch/x86/include/asm/unistd_32.h`

```
#define __NR_plus           338
#define __NR_syscalls       339
```

Add following line in `arch/x86/include/asm/unistd_64.h`

```
#define __NR_plus           300
#__SYSCALL(__NR_plus, sys_plus)
```

- For Linux Kernel 3.10

  Add prototype of system call in `arch/x86/include/asm/syscalls.h`

  ```
  asmlinkage long sys_plus(long,long);
  ```

  Add following line in `arch/x86/syscalls/syscall_32.tbl`

  ```
  <number>  <abi>   <name>   <entry point> <compact entry pt>
  358       i386     plus       sys_plus
  ```

  Add following line in `arch/x86/syscalls/syscall_64.tbl`

  ```
  <number>  <abi>   <name>   <entry point> <compact entry pt>
  314       64.      plus       sys_plus
  ```

- For Linux Kernel 4.2

  Add prototype of system call in `arch/x86/include/asm/syscalls.h`

  ```
  asmlinkage long sys_plus(long,long);
  ```

  Add following line in `arch/x86/syscalls/syscall_32.tbl`

  ```
  <number>  <abi>   <name>   <entry point> <compact entry pt>
  358       i386     plus       sys_plus
  ```

  Add following line in `arch/x86/entry/syscalls/syscall_64.tbl`

  ```
  <number>  <abi>   <name>   <entry point> <compact entry pt>
  314       64.      plus       sys_plus
  ```

Step-3: Update the Makefile

- For Linux Kernel 5.3.7

Create a new Makefile in Custom_Syscall/ directory and write a single line in it:

```
$ vim Custom_Syscall/Makefile

obj-y := plus.o
```

Open the `linux-5.3.7/Makefile` and search for `core-y` string:

```
$ vim linux-5.3.7/Makefile
  ----
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
        ----
```

Replace above line with the following

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

This tells kbuild that there is one object in directory Custom_Syscall named `plus.o`. This `plus.o` will be build from `plus.c`

- For Linux Kernel 2.6, 3.10 and 4.2

There are over 1000 Makefiles in the kernel source folder, just open the `kernel/Makefile` and add the following line in it.

```
$ vim kernel/Makefile
- - -
obj-y += plus.o
```

This tells kbuild that there is one object in that directory named `plus.o`. This `plus.o` will be build from `plus.c` or `plus.S`. If plus.o shall be built as a module, the variable `obj-m` is used. Therefore, the following pattern is often used:

Example: `obj-$(CONFIG_FOO) += plus.o`

$(CONFIG_FOO) evaluates to either y (for built-in) or m (for module). If CONFIG_FOO is neither y nor m, then the file will not be compiled nor linked.

Step-4: Configure, Compile and Install

You may skip the configure step, if you have already done it in the previous tutorial, and just need to compile giving the following command, which will now take about 10 minutes (Enjoy):

```
$ time make –j4
```

On success you can give the following command to install:

```
$ sudo  make install
```

Step-5: Test Your System Call

Before executing this step, it is assumed that you have successfully installed the new kernel and are running it. You have verified all the steps to confirm that the new kernel is running correctly. Now to test the system call, you have added to your kernel, lets write some test code, lets say we do this in driver.c.

```
$vim driver.c
#include <stdio.h>
#include <stdlib.h>
#include<sys/syscall.h>
#include <unistd.h>
long plus (int, int);
int main (int agrc, char* argv[]){
     int arg1=atoi(argv[1]);
     int arg2=atoi(argv[2]);
     long x= syscall(436, arg1, arg2);
     printf ("Sum: %ld\n", x);
     return x;
}
```

Compile the program and enjoy executing it. Finally, you did it. Congratulations

```
$ gcc driver.c
$ ./a.out 5 3
Sum: 8
```

- Step-6: Write Wrapper of System Call and Test

Normally, we donot call a system call using `syscall()` system call. Rather we have a library wrapper function to almost all of the available system calls. Wrapper function simplifies the system call interface and do necessary things like invoke system call by its number, placing arguments in CPU registers, switch to kernel mode, and set errno in case of failure. Otherwise these are the things that are done by programmer himself. So it hides the complexity and provide simple interface.

For our system call we write a simplest wrapper function, make its static library and compile the test program with this library.

```
$ vim pluswrapper.c
#include<unistd.h>
#include<sys/syscall.h>
#define PLUS 436
long plus(long a, long b)
{
     return syscall(PLUS,a, b);
}
```

Now make its static library by following commands

```
$ gcc -c pluswrapper.c
$ ar crs libarifplus.a pluswrapper.o
```

Rewrite the test program that is, test.c

```
$vim driver.c
     #include <stdio.h>
     #include<sys/syscall.h>
     #include <unistd.h>
     long plus (int, int);
     int main (int agrc, char* argv[]){
     `     int arg1=atoi (argv[1]);
           int arg2=atoi (argv[2]);
           long x= plus(arg1, arg2);
           printf ("Sum: %ld", x);
           return 0;
```

```
}
```

Now compile the driver.c file and link it with this static library by following commands:

```
$ gcc –c driver.c –o driver.o
$ gcc driver.o –larifplus –L.
```

Now run Add and pass 2 arguments through command line.

```
$ ./a.out 3 5
Sum: 8
```

- Step-7: Write man page of System Call

Manual pages are the canonical type of documentation for UNIX systems. Manual pages are reference documentation, intended to quickly answer questions like "what is the purpose of this command" or "is there an option to enhance its functionality". It is to give information about the basic functionality, description and knowledge about the kernel commands and system calls. To see a man page use the following command.

All the man pages are stored in `/usr/share/man/man{x}/` directory, where x is the chapter number of what man page is about. The man pages are named as `command-name.{x}.gz` where x is the chapter number to which the command belongs. There are a total of eight chapters of man pages, which are as follows:

1. Executable/Shell commands
2. System calls (functions provided by the kernel)
3. Library functions (functions within program libraries)
4. Special files (usually devices, those found in /dev) and drivers
5. File formats and conventions
6. Games and screensavers
7. Miscellaneous
8. System administration commands (usually only for root)

Layout of Man page

All man pages follow a common layout that is optimized for presentation on a simple ASCII text display, possibly without any form of highlighting or font control. Sections present may include but are not limited to following:

NAME
SYNOPSIS
CONFIGURATION [Normally only in Section 4]
DESCRIPTION
OPTIONS [Normally only in Sections 1, 8]
EXIT STATUS [Normally only in Sections 1, 8]
RETURN VALUE [Normally only in Sections 2, 3]
ERRORS [Typically only in Sections 2, 3]
ENVIRONMENT
FILES
ATTRIBUTES [Normally only in Sections 2, 3]
VERSIONS [Normally only in Sections 2, 3]
CONFORMING TO
NOTES
BUGS
EXAMPLE
SEE ALSO

Note: For more details please refer to

`$ man man-pages` or `$ man man` or `$ man 7 man`

Writing man page of plus system call

Writing manual pages is just like writing static html pages. In html we use <tag> to modify the text here in man pages we use macros. In the same sense, always put the macro in the start of the line e.g. to make text bold we use the macro .B and put it in the start of the line, like

```
.B "text to be bold"
```

The draft man page of the plus system call which we added to kernel is shown below

$ vim plus.2

```
.\" This is the manual page for reverse system call .\"
.TH PLUS 2 “February,2015”  “Linux Programmer's Manual”
.SH NAME
     plus \- adds two integers
.SH SYNOPSIS
.B #include<unistd.h>
.br
.sp
.BI "long plus(long " " integer1 ", "long " integer2);
.SH DESCRIPTION
This system call is written by Arif Butt to teach his students of
Advance Operating System at PUCIT. It adds two integers and it
returns the resultant integer.
.SH "RETURN VALUE"
return 0 on success or -1 on failure.
.SH "SEE ALSO"
.BR plus (1),
.BR plus (7)
.SH "MESSAGE"
Dear students of PUCIT, there is no short cut to hardwork. Happy
Learning with Arif Butt...
```

To install/add the man page we created, gzip it and copy it to chapter2 of man pages as it is a man page for system call

$ gzip plus.2

$sudo cp plus.2.gz /usr/share/man/man2/

After copying the file to the man page location check man page.

$ man plus