**Introduction:**

This is a summary of the activity carried out over the summer of 2024 with the guidance of professor R. Badrinath. The goal was to get a first hand experience of an actual OS. Therefore the focus was on exploring linux and this document summarises the activity carried out over the period.

**Learning The Linux Kernel**

-Vrajnandak Nangunoori

**B**efore trying to understand what the "kernel" or "Linux" is, let's begin by understanding how our computers operate. Every computer, from laptops to servers, relies on a complex orchestration of hardware components—processors, memory, storage devices, and input/output devices—that work together to execute tasks. To manage these resources efficiently, we need an operating system (OS), which serves as the intermediary between users and the computer hardware. Consider this: the screen displaying these words and your ability to navigate with a mouse are all orchestrated by an OS like Linux, which powers a multitude of devices globally. Linux, one of the most prominent examples of an OS, powers a vast array of devices worldwide, from smartphones to supercomputers. At its core lies the kernel, a fundamental component that facilitates communication between software and hardware, ensuring seamless operation and optimal performance.

So, how do you go about making a kernel? The answer is simple—it's just code. Developing a kernel involves writing and refining lines of code that serve as the foundational software layer directly interacting with the hardware. These codes are written to manage system resources, handle input and output operations, schedule tasks efficiently, and provide essential services to higher-level software. Kernels are designed to be highly optimized, ensuring that they operate swiftly and reliably under varying workloads and conditions. Programmers proficient in systems programming languages like C and assembly language (which are well-suited for systems programming) delve into the intricacies of hardware architecture, memory management, device drivers, and system calls to create a robust kernel. Each decision in kernel development impacts how the entire operating system functions, making it a critical and challenging aspect of software engineering.

To understand and work with the Linux kernel effectively, downloading its source code is crucial. This allows developers to inspect its inner workings, make modifications, and contribute improvements back to the community. Setting up a suitable development environment is equally important; Ubuntu, a Linux-based operating system, provides an environment tailored for kernel development. Unlike using Windows, which can introduce compatibility issues and complexities that are challenging to resolve, Ubuntu offers seamless integration with kernel development tools and libraries. This ensures a smoother coding experience and facilitates debugging and testing processes.

Overall, using ubuntu will be more efficient and productive. If you have a windows OS, then you can download VirtualBox (a powerful, open-source virtualization software developed by Oracle. It allows you to create and run virtual machines (VMs) on your computer, effectively enabling you to run multiple operating systems simultaneously on a single physical machine. VirtualBox supports a wide range of guest operating systems, including various versions of Windows, Linux, macOS, and others. VirtualBox runs on Windows, macOS, Linux, and Solaris and other OS.

The below explanation uses ‘**Ubuntu 22.04**’ (64-bit), and the linux kernel version we will be working with is the ‘**Linux-5.15.161**’ version. However I did face a problem saying that “<current\_user> is not in the sudoers file. This incident will be reported.”, So I added the following line:

‘<current\_user> ALL=(ALL:ALL) ALL’ in the ‘/etc/sudoers’ file. This will add the ‘<current\_user> to the list of sudo users. (replace ‘current\_user’ with the user name that shows up in the error message).

**->Steps to Download the Linux Kernel Source Code:**

**Reference:** [**https://davidaugustat.com/linux/how-to-compile-linux-kernel-on-ubuntu**](https://davidaugustat.com/linux/how-to-compile-linux-kernel-on-ubuntu)

1. We will first download a few packages required to be able to load, make modifications to the kernel code.
   1. Run ‘sudo apt update’ on the terminal. This updates the list of available packages and their versions stored in the system’s package index. This also helps avoid things like trying to install a package that has been moved or updated in the repository. It refreshes the local cache.
   2. Now, run ‘sudo apt install build-essential libncurses-dev bison flex libssl-dev libelf-dev fakeroot’. A quick breakdown of everything:
      1. ‘apt’ is a package manager to install multiple packages on a Debian-based Linux distribution(like Ubuntu).
      2. Build\_essential: (<https://itsfoss.com/build-essential-ubuntu/>) It belongs to Debian and not a software package in itself. It contains a list of packages that are req to create a Debian package. These packages are libc,gcc,g++,make. The build-essential package has dependencies on those packages and hence u end up downloading them with a single command of ‘sudo apt install build-essential’. Build-essential only has some but not all packages. If you get an error saying ‘couldn’t find package build-essential’ then type ‘sudo apt-get install build-essential’ and then presse TAB instead of pressing ENTER (Haven’t tried this out because I didn’t get this error but I read that this worked for people who got the error).
      3. Libncurses-dev: has the development tools and libraries for ‘ncurses’. It is a programming library providing an API that allows the programmers to write text-based user interfaces (TUI) in a terminal-independent manner. It is a toolkit for developing “GUI-like” application software that runs under a terminal emulator.
      4. Bison: It is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR(L stands for ‘left-to-right’ and R stand for ‘rightmost derivation in reverse’) or Generalized LR (aka GLR) parser, employing LALR(1) parser tables. (LALR(1) stands for Look-Ahead LR with 1 token of look ahead). Bison is part of GNU project.
      5. Flex: Stands for “Fast Lexical Analyzer Generator”, and is a faster variant of lex. It generates a lexical analysis program (named lex.yy.c) based on the regular expressions and C statements contained in one or more input files.
      6. Libssl-dev: This package is part of the OpenSSL project's implementation of the SSL and TLS cryptographic protocols for secure communication over the Internet. (SSL is Secure Sockets layer and TLS is Transport layer security).
      7. Libelf-dev: This package provides the development files (headers and libraries) for working with the Executable and Linakable Format(ELF).
      8. Fakeroot: This tool simulates superuser(root) privileges for the purpose of building
2. Now, run ‘sudo apt install dwarves’. Dwarves package involves a set of tools for working with ELF files and debugging information. One primary tool is ‘pahole’ used to examine layout of data structures in binaries. (ex: ‘pahole my\_binary\_file’ would examine the binary named ‘my\_binary\_file’).
3. We can now download the linux kernel source code after having downloaded the pre-requisite packages. Go to ‘kernel.org’ website and use the “tarball” link on the ‘longterm’ version. It’s always better to use a relatively older version as they’re much more stable and less prone to errors.

First, make a directory inorder to keep the linux kernel code separate from your other files and for the sake of easiness.

‘mkdir Linux\_Proj’

‘cd Linux\_Proj’

‘wget <https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.15.161.tar.xz>’ //Replace this link with the link you chose from the ‘kernel.org’ website.

‘tar -xf linux-5.15.161.tar.xz’ //This is to unzip the tar file you’ve just downloaded.

This will now unzip the zipped folder and you can now change into that directory and work with the kernel code.

1. Change to the kernel code directory: ‘cd linux-5.15.161’
2. ‘make localmodconfig’ will setup the configuration file (‘.config’) in the directory using the loaded kernel modules and will include only these in the build. This will let you use a variety of options with the kernel code such as ‘scripts/config –state MACRO\_NAME’ which will let you check if a certain macro has been defined or not.
3. Now, inorder for the build to work, a few modifications to the configuration are needed:
   1. scripts/config –disable SYSTEM\_TRUSTED\_KEYS
   2. scripts/config –disable SYSTEM\_REVOCATION\_KEYS
   3. scripts/config –set-str CONFIG\_SYSTEM\_TRUSTED\_KEYS “”
   4. scripts/config –set-str CONFIG\_SYSTEM\_REVOCATION\_KEYS “”
4. Now, you can compile the kernel using command ‘fakeroot make’. This will take a lot of time, even upto 3 hours. After the build finishes, run ‘echo $?’ which should output ‘0’ for a successful build. If anything else is returned, it indicates that there has been an error.
5. Now, you will have to install the kernel modules and the kernel itself by doing ‘sudo make modules\_install’ and ‘sudo make install’ respectively. After doing this you will have to boot into this kernel version.

We can now make changes to the kernel code and see their effects only after booting into this kernel code. Right now, we are booted into a different version of the kernel code than the one we’ve just downloaded. So in order for us to see the changes we’ve made to the kernel it is absolutely necessary for us to boot into the correct kernel version. In case you don’t already get a menu display on opening the Ubuntu, follow the below steps to update your grub.

**->Steps To Update Grub:**

In some systems(unfortunately mine), the setup for accessing the boot-menu is not set by default and so we will have to change it. Follow the below steps in order to be able to boot into a different version of the kernel using a menu display:

* We need to make changes to the ‘/etc/default/grub’ file which contains the values considered during booting. Open the file with sudo privileges and a text editor for making changes: ‘sudo nano /etc/default/grub’.
* Now, change the value ‘GRUB\_TIMEOUT\_STYLE’ from ‘hidden’ to ‘menu’. This will give a menu display for booting into different kernel versions.
* Also, change the value ‘GRUB\_TIMEOUT’ from ‘0’ to ‘10’. This will tell the computer that to choose the default kernel version if you don’t choose the version yourself within 10 seconds. You can set it to a higher number if you’d like.
* Now, ‘sudo update-grub’ will update the values to be considered during booting.
* ‘sudo reboot’ to basically restart, which should now give you a menu display of a few options.
* Choose ‘Advanced options for Ubuntu’ and then choose the kernel version you’ve downloaded. Cross-check by running ‘uname –rs’ and checking if the version displayed matches the version you’ve downloaded.

We can now make changes to the kernel and see their effects.

**->Changing the user name from ‘Linux’ (visible on doing a ‘uname -rs’) to ‘MyOS’:**

**Reference:** [**https://davidaugustat.com/linux/how-to-compile-linux-kernel-on-ubuntu**](https://davidaugustat.com/linux/how-to-compile-linux-kernel-on-ubuntu)

The default name ‘Linux’ is stored in the ‘/include/linux/uts.h’ file, where ‘include’,’linux’ are obviously directories and ‘uts.h’ is the actual file. Once you open it with a text editor, change the ‘#define UTS\_SYSNAME’ field by replacing the text “Linux” with “MyOS” (or whatever name you’d like to give).

Now, run ‘fakeroot make’ followed by ‘sudo make modules\_install’, ‘sudo make install’ and then a ‘sudo reboot’. Once you boot into the version you’ve downloaded, you can run ‘uname -rs’ and see the change of the name to the text you’ve put.

**->Modifying An Existing syscall:**

Syscalls, short for system calls, are the fundamental interface between a user-level process and the kernel in an operating system. They provide a way for applications and user programs to request services from the operating system kernel. Syscalls are essential for tasks that require privileged operations or access to system resources that are managed by the kernel. They allow user-level processes to interact with the operating system kernel to perform tasks such as reading and writing files, creating and managing processes, network communication, memory management, and more. Essentially, they enable applications to access system resources that would otherwise be protected and isolated from user-level programs.

**->Modifying the ‘do\_syscall\_64’ function**

There is a common entry point whenever any syscall is called. This is the ‘do\_syscall\_64’ function in the ‘arch/x86/entry/common.c’ file :

“”””

\_\_visible noinstr void do\_syscall\_64(struct pt\_regs \*regs, int nr)  
{

/\* The code I added starts from here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static int times\_to\_print=0;

if(times\_to\_print<30) //Simple if condition to print the below statement a max of 30 times.  
{

printk(KERN\_INFO “This is the entry point, nr: %d\n”, nr); //’nr’ in this context, is the syscall number. For example, if ‘nr’ were ‘0’, then it would most likely refer to ‘sys\_read()’ if the system’s architecture is ‘x86’.

times\_to\_print++;

}

/\* The code I added ends here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

add\_random\_kstack\_offset();

nr=syscall\_enter\_from\_user\_mode(regs, nr);

instrumentation\_begin();

if(!do\_syscall\_x64(regs, nr) && !do\_syscall\_x32(regs, nr) && nr!=-1)  
 {

regs->ax = \_\_x64\_sys\_ni\_syscall(regs);

}

instrumentation\_end();

syscall\_exit\_to\_user\_mode(regs);

}

“”””

So, whenever this entry point function is invoked, the statement that we’ve printed using ‘printk()’ (A function which prints the message to the kernel ring buffer), will be visible on viewing the buffer using a ‘sudo dmesg’. The ‘KERN\_INFO’ is a level which tells the kernel about the urgency/importance of the message. Some other levels include ‘KERN\_ALERT’ etc.

Now, to see the effects you will have to recompile the kernel:

‘fakeroot make’ in the root source directory of the downloaded kernel code.

‘sudo make modules\_install’

‘sudo make install’

‘sudo reboot’

Now, boot into the kernel code that you’ve modified and make a call to any syscall in maybe a .C file. Now, doing a ‘sudo dmesg’ will let you see the kernel ring buffer and you can see your printk statements.

‘sudo dmesg | grep “This is the entry point, nr:”’ //This will let you see the printk() that have been executed.

**->Modifying the ‘ksys\_write()’ syscall:**

We can also modify some other well known functions like the ‘write()’ function, but instead of making changes to this wrapper function (uses vfs\_write(), ksys\_write()), we can modify the inner functions as well(You can obviously make changes to any functions as long as the code is available, you have the write privileges, the necessary header files are included).

Here, we will be adding a printk() statement in the ksys\_write() function in the ‘/fs/read\_write.c’ file:

“”””

ssize\_t ksys\_write(unsigned int fd, const char \_\_user \*buf, size\_t count)  
{

struct fd f=fdget\_pos(fd);

ssize\_t ret=-EBDAF;

/\* The code I added starts from here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static int times\_to\_print=0;

if(times\_to\_print==0)  
 {

printk(KERN\_INFO “Inode number: %lu\n”, (file\_inode(f.file))->i\_ino); //Prints the inode number of the inode associated with the said file. In Linux, every file/directory is associated with a unique inode unless it’s a hardlink(they refer to the same inode to which the hardlink points to).

printk(KERN\_INFO “ksys\_write filename: %s\n”,f.file->f\_path.dentry->d\_name.name); //This will print the file name.

times\_to\_print++;

}

/\* The code I added ends here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//The rest of the functions code.

}

“”””

This will log the name of the file into which a write has been called for.

Inorder for this modified code to be invoked, you will have to recompile the kernel and then call this function as described above.

‘sudo dmesg | grep “Inode number:”’ //To see the above printk() statements.

‘sudo dmesg | grep “ksys\_write filename:” //To see the above printk() statements.

**->Modifying the ‘vfs\_write()’ syscall:**

This function is called by ‘ksys\_write’. We will add a statement to print the filename, similar to how we’ve printed the name above.

“”””

ssize\_t vfs\_write(struct file \*file, const char \_\_user \*buf, size\_t count, loff\_t \*pos)

{

ssize\_t ret;

/\* The code I added start here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static int counter=0;

if(counter<30)  
 {

printk(KERN\_INFO “vfs\_write invocation for file: %s\n”, file->f\_path.dentry->d\_name.name);

counter++;

}

/\* The code I added ends here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//The rest of the code of the vfs\_write function.

}

‘sudo dmesg | grep “vfs\_write invocation for file:”’ //To see the above printk() statements.

The flow is basically that ‘sys\_write()’ calls ksys\_write() which calls vfs\_write(). It is the same for the read() calls as well.

You can even make a call to these functions using a .C file and see the statements being printed. However, you will have to increase the value of the counter in this case since during boottime, some of these functions will be counted and may cause the counter to be incremented beyond the limit you’ve put and hence the ‘if’ condition won’t evaluate to true.

I’ve made a sample usage.c file at the bottom of this document.

**IMPORTANT NOTE: -> Only declare your own variables if you’re going to use them, else the warning ‘unused variable’ may be thrown.**

**->When declaring the variables, always declare them with the correct data types (include headers if necessary) and at the top of the function before any executable instruction. This is to confirm with the C90 standards. If you declare a variable after some other call inside the function, it is highly likely that you may get a warning/error saying that the code doesn’t confirm to the C90 standards.**

**->It is important to ensure that any modification to the kernel code doesn’t cause something like an infinite recursion or infinite printk or anything that might put a load on the kernel. For example, in the above modifications, if we were to endlessly execute the printk() (maybe if you change the ‘if’ condition or if you forget to increment the counter) then there will be infinite messages in the kernel ring buffer and hence you may not be able to boot into this kernel version.**

**->Sometimes, putting just one printk() may also affect the kernel for the worse. For example, if you were to put just one printk() in ‘do\_syscall\_64()’ thinking that it will be fine since it’s just going to be called once on a syscall invocation, then you are wrong… If you were to do a ‘sudo dmesg’, this would call the ‘sys\_read’ and hence the printk() in the ‘do\_syscall\_64()’ gets executed. But the printk() in this would write to the kernel ring buffer and since ‘sudo dmesg’ reads from the kernel ring buffer, it would try to read this new message again. So, just one call to ‘sudo dmesg’ will cause an infinite recursion of a ‘read’ (by ‘sudo dmesg’) followed by a write(due to the printk() in ‘do\_syscall\_64’ which is called by ‘read’).**

**->Adding a New Syscall:**

Two ways of adding a syscall will be discussed here. The first way is to add the new syscall’s definition within an existing file which will not require you to make changes to the makefile. The second way is to write the new syscall’s definition within a different file.

Severy system call in linux, is identified by a unique number which are usually defined in an architecture-specific syscall table(In my case, my computer had the architecture of ‘x86’ so the file for me was ‘arch/x86/entry/syscalls/syscall\_64.tbl’. This file contains a list of 64-bit system call numbers and entry vectors with each entry having the format “number abi name entry\_point” where,

number – system call number

abi – Application Binary interface, in this context it refers to the ABI number or specification associated with a particular syscall. The value can be ‘common’ or ‘64’ or ‘x32’.

name – is the symbolic name for the associated syscall. ex: ‘read’, ‘write’ etc.

entry\_point – Is the function which has the core code for the specific syscall ex: ‘sys\_read’, ‘sys\_write’ etc.

Inorder to make a new syscall, we have to define a new unique number for it and place it in this file. It is important to go through the entire file once and check for unique numbers so as to not use an existing sys number.

After adding the entry in the syscall.tbl file, we will now declare the prototype of the function in the header file, ‘include/linux/syscalls.h’ which should contain most of the prototypes for the various syscalls.

A macro ‘asmlinkage’, (source: <https://www.cs.uic.edu/~tkhatiwa/kernel/kernel_gotchas.html#:~:text=asmlinkage%3A%20This%20macro%20(defined%20in,in%20the%20current%20address%20space)>. ) tells the compiler to pass the arguments on the stack, and is used in the prototype declaration of all the existing syscalls. (But when I checked the ‘include/linux/linkage.h’ file, I found the following:

“”””

#ifdef \_\_cpluplus

#define CPP\_ASMLINKAGE extern “C”

#else

#define CPP\_ASMLINKAGE

#endif

#ifndef asmlinkage

#define asmlinkage CPP\_ASMLINKAGE

#endif

“”””

Not too sure what it does, but nonetheless it’s better to declare it. It is quite possible that the definition in a different file is being used. Haven’t checked it out yet.)

The prototypes are of form:

“asmlinkage long sys\_sysname()”

The definition of the syscalls is written using some pre-defined MACRO’s of form ‘SYSCALL\_DEFINEx’ which are function-like macros taking in different parameters and perform a bit more than just simply defining the function. The ‘x’ is a number from 1 to 6, to specify the number of arguments to the system call. It takes in the ‘name’ of the system call followed by the arguments. In order to implement the definition of the new syscall, this macro has to be used.

Also, There is generally a need to add a default implementation which will be executed incase the kernel is not able to find the definition to the specified syscall, this is called the ‘fallback stub’ and is present in the ‘kernel/sys\_ni.c’. These fallback stub functions usually only return ‘-ENOSYS’ and don’t contain any other code. However, if the following function is already defined, then there shouldn’t be a need for you to explicitly create a fallback stub for your newly created syscall. This function, according to the comments in the file, is used for the non-implemented syscalls.

“”””

asmlinkage long sys\_ni\_syscall(void)

{

return -ENOSYS;

}

“”””

To still add a fallback stub for your system call, you can use an existing macro ‘COND\_SYSCALL(name)’ where name is the thing you specify in the ‘.tbl’ file mentioned above.

There is also the ‘include/uapi/asm-generic/unistd.h’ file which contains a generic list of the syscalls and is used by the other architectures as well. The file also contains a variable ‘\_\_NR\_syscalls’ which is used to defined the total number of syscalls.

You can create a new entry by doing:

#define \_\_NR\_NameOfSyscall

\_\_SYSCALL(\_\_NR\_NameOfSyscall, sys\_NameOfSyscall)

Replace the ‘NameOfSyscall’ with the value of the name field in the .tbl file

You can also change the value of the “\_\_NR\_syscalls” to a bigger number to increase the number of syscalls.

**->Adding a syscall in an existing file:**

Here, we will be adding a new syscall to print a message “hello world, I am developing linux kernel” and another syscall which will take 2 arguments(integers) to add them and return the sum.

1. In file ‘arch/x86/entry/syscalls/syscall\_64.tbl’, added the following line:

‘449 common print\_hi sys\_print\_hi’

In already existing ‘kernel/sys.c’ file, added the following piece of code:

“”””

SYSCALL\_DEFINE0(print\_hi)

{

printk(KERN\_INFO “Printing from new syscall\n”);

return 0;

}

“”””

In file ‘include/linux/syscalls.h’ file, added the following line:

‘asmlinkage long sys\_print\_hi(void);’

Now run the following commands in the source root directory of the downloaded kernel code:

‘make headers\_install’

‘fakeroot make’

‘sudo make modules\_install’

‘sudo make install’

‘sudo reboot’

The usage.c file:

“”””

#include<unistd.h>

#include<sys/syscall.h>

#include<stdio.h>

int main()

{

long ret=syscall(449);

printf(“ret: %ld\n”, ret);

return 0;

}

“”””

Calling the syscall using ‘sys\_print\_hi()’ or ‘print\_hi’ will throw an error saying ‘undefined reference’. Not too sure about the reason for this but it is probably because of the protection mechanisms of ‘kernel space’ and ‘user space’.

1. Using the above syscall implementation as reference, use the below lines of code:

‘450 common add\_2\_nums sys\_add\_2\_nums’

“”””

SYSCALL\_DEFINE2(add\_2\_nums, int , a, int , b)

{

printk(“sys\_add\_2\_nums invoked\n”);

return a+b;

}

“”””

‘asmlinkage long sys\_add\_2\_nums(int a,int b);’

Use the above usage.c file and put the syscall(450) as the function to be called.

Run the above commands as well to compile the kernel.

**->Adding a syscall in a new file:**

The same steps are followed except for where you define the function.

‘mkdir dir\_name’

‘cd dir\_name’

‘vim my\_new\_syscall.c’

Add the following:

“”””

#include<linux/kernel.h>

#include<linux/syscalls.h>

SYSCALL\_DEFINE0(nangu\_hi)

{

printk(KERN\_INFO “Printing from sname”);

return 0;

}

“”””

‘vim Makefile’

Add the following line ‘obj-y := my\_new\_syscall.o’. This will be used by the main Makefile in the build to include this object file in the build, providing the definition to the specified function.

Now, in the main Makefile present in the root directory of the downloaded kernel source code, add “ dir\_name/” with the space at the end of ‘core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/”.

Now, compile and run the code using above commands and the above usage.c file, using the proper syscall number.

The overall usage.c file, using the above 3 syscalls would look like:

“”””

#include<unistd.h>

#include<stdio.h>

#include<sys/syscall.h>

int main()  
{  
 //Executing sys\_print\_hi.

long ret=syscall(449);

printf(“sys\_print\_hi: %ld\n”,ret);

//Executing sys\_add\_2\_nums.

long add\_ret=syscall(450);

printf(“sys\_add\_2\_nums: %ld\n”, add\_ret);

//Executing sys\_nangu\_hi.

long ret2=syscall(451);

printf(“sys\_nangu\_hi: %ld\n”,ret2);

return 0;

}

“”””

Run using ‘gcc usage.c’ and then ‘./a.out’. Doing a ‘sudo dmesg’ will make the printk() in the syscall definitions visible.

**->Making Changes to the Scheduler Subsystem:**

**//Tried several things but requires a bit more work.**

**->Creating Modules:**

A kernel module(or loadable kernel mode) is an object file containing code to extend kernel functionality at runtime i.e., it is loaded as needed and can be unloaded when not needed. They don’t require a complete kernel recompilation. They also allow for easy installation and removal of device drivers without rebooting the system. Kernel driver directly interacts with the hardware devices, bridging the OS kernel and the specific hardware components.

The advantage of creating a module is that you can have direct access to the hardware as it is in kernel but there would be certain limitations like a smaller stack maybe.

Thus, Kernel drivers are an integral part of the Linux kernel, providing direct interaction with hardware devices, while kernel modules offer flexibility and modularity through dynamic loading.

1. ‘modprobe’ command: A high-level command to automatically load,unload or list modules based on module dependencies listed in ‘/lib/modules/<kernel-version>/modules.dep’. It loads a specified module and any other modules it depends on. It also handles dependencies and can load modules dynamically as needed.

Ex: modprobe <module\_name> 🡺Loads the module and it’s dependencies.

Ex: modprobe -r <module\_name> 🡺Unloads the module and any modules that depend on it.

1. ‘insmod’ command: A lower-level command used to insert a module directly into the running kernel without checking dependencies. It inserts a module into the kernel. However, it does not handle dependencies automatically, so it's generally used when you need more control or want to insert a module explicitly.

Ex: insmod <path\_to\_module.ko> 🡺Inserts the specified module into the kernel.

1. ‘rmmod’ command: To remove(unload) kernel modules from the running kernel. It unloads a module that was previously loaded into the kernel. It can also handle dependencies, ensuring that modules depending on the one being removed are also properly handled.

Ex: rmmod <module\_name> 🡺Unloads the specified module.

Thus, ‘modprobe’ handles module dependencies automatically, whereas ‘insmod’ and ‘rmmod’ require explicit handling of dependencies.

A module can be created even in a directory outside of the root directory containing the downloaded kernel’s source code. To create a basic module, there are only 4 basic steps:

‘mkdir module\_dir\_name’

‘cd module\_dir\_name’

‘vim module\_functionality.c’

The code of a basic module looks like this:

“”””

#include<linux/kernel.h>

#include<linux/init.h>

#include<linux/module.h>

MODULE\_DESCRIPTION(“Simple example Kernel module”);

MODULE\_AUTHOR(“Vrajnandak Nangunoori”);

MODULE\_LICENSE(“GPL”);

static int \_\_init my\_mod\_VN\_init(void)

{

printk(“Hello World, from my\_mod\_VN\n”);

return 0;

}

static void \_\_exit my\_mod\_VN\_exit(void)

{

printk(“Goodbye World, from my\_mod\_VN\n”);

}

module\_init(my\_mod\_VN\_init);

module\_exit(my\_mod\_VN\_exit);

**“”””**

Now, we need to create a Makefile, which is used to create a kernel object for the module which can be loaded into the running kernel. The content of the makefile itself is very basic and there are 2 way to go about to make a kernel object:

->Method 1: To create only a ‘Makefile’. The content would look like this:

“”””

obj-m += module\_functionality.o #’obj-m’ variable is used to indicate a kernel module object file. There’s a good reason to use ‘+=’ instead of just ‘=’ because the ‘+=’ let’s us append the specified target object file to the existing list of objects(‘obj-m’). This is useful when dealing with multiple source files that need to be compiled into separate kernel modules.

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

“”””

->Method 2: To create a ‘Makefile’ and a ‘Kbuild’ file:

The Makefile would look something like this:

“”””

KDIR = /lib/modules/$(shell uname -r)/build

kbuild:

make -C $(KDIR) M=$(PWD)

clean:

make -C $(KDIR) M=$(PWD)

“”””

The Kbuild file would look like this:

“”””

EXTRA\_CFLAGS = -Wall -g

obj-m = module\_functionality.o

“”””

After having followed one of the above steps, run the below commands:

‘make’

‘sudo insmod module\_functionality.ko’

You can now do ‘sudo dmesg’ to indeed check the printk() in the ‘init’ method.

‘sudo rmmod module\_functionality’

You can now do ‘sudo dmesg’ to see the printk() in the ‘exit’ method as well.

**->Creating a module that prints a message whenever the system’s time is a multiple of 5 minutes, using a timer:**

The idea is now to be able to create a timer, execute it every minute, read the system’s time at that instant and then print a message if it fits our condition of being a multiple of 5 minutes.

**To get the current time:** In file ‘/kernel/time/timekeeping.c’ that there was a comment saying that ‘kernel timekeeping code and accessor functions. Based on code from timer.c, moved in commit 8524070b7982’ and saw a function ‘ktime\_get\_real\_ts64()’ which according to the comments says that it returns the time of day in a timespec64. This ‘ktime\_get\_real\_ts64’ takes in a ‘ struct timespec64 \*ts’ argument and puts the time inside it using ‘timespec64\_add\_ns’ which puts the values into ‘ts->tv\_sec’ and ‘ts->tv\_nsec’. The definition of timespec64:

Struct timespec64 {

time64\_t tv\_sec; //seconds.

long tv\_nsec; //nanoseconds.

}

In the same file, time64\_t is defined to be \_\_s64 by statement ‘typedef \_\_s64 time64\_t;’ where \_\_s64 represents ‘signed 64-bit integer’ and similarily, ‘\_\_u64’ would represent ‘unsigned 64-bit integer’.

So now, we can use the ‘ktime\_get\_real\_ts’ function to get the system’s time.

**To format the time read:** Now, I had to search for a function, if any, that could convert this to proper format of “HH:MM:SS”. Found the file ‘/include/linux/rtc.h’ where ‘rtc’ stands for ‘real time clock’ but I think the functions are for drivers. Then I found the function ‘time64\_to\_tm’ in the file ‘kernel/time/timeconv.c’ which according to the comments, converts the calendar time to local broken-down time. time64\_to\_tm(time64\_t totalsecs, int offset, struct tm \*result) puts the year, month, day, day\_of\_year values into the ‘struct tm \*result’. Now, this function was defined in file ‘/include/linux/time.h’. The definition of ‘struct tm’ is as follows from ‘/include/linux/time.h’ file:

struct tm {

int tm\_sec; //number of seconds after minute, [0,60]. ‘60’ to allow for leap seconds.

int tm\_min; //number of minutes after hour, [0,59].

int tm\_hour; //number of hours past midnight, [0,23].

int tm\_mday; //day of the month, [1,31].

int tm\_mon; //number of months since January, [0,11].

long tm\_year; //number of years since 1900.

int tm\_wday; //number of days since Sunday, [0,6].

int tm\_yday; //number of days since January 1, [0,365].

}

**To setup a timer:** Now, I have to setup a timer which can execute a function using the above 2 functions to implement the required functionality of printing a message when the time is a multiple of 5 minutes. In file ‘/include/linux/timer.h’, there is a function ‘timer\_setup’, which is a function-like macro and uses another helper function-like macro ‘\_\_init\_timer’, which according to the comments prepares a timer for first use. The ‘\_\_init\_timer’ macro was defined based on whether or not ‘CONFIG\_LOCKDEP’ macro was defined. In my source code, this macro wasn’t defined (checked using ‘scripts/config –state CONFIG\_LOCKDEP’). The ‘\_\_init\_timer’ was defined as follows:

#define \_\_init\_timer(\_timer, \_fn, \_flags) \

Init\_timer\_key((\_timer),(\_fn),(\_flags),NULL,NULL)

In the same file, there was a function ‘void init\_timer\_key(struct timer\_list \*timer, void (\*func)(struct timer\_list \*), unsigned int flags, const char \*name, struct lock\_classs\_key \*key);’. The file ‘/kernel/time/timer.c’ had the function definition of the ‘init\_timer\_key’ function and according to the comments, it initializes a timer and also that init\_timer\_key() must be done to a timer prior calling \*any\* of the other timer functions. The function was defined as follows:

void init\_timer\_key(struct timer\_list \*timer, void (\*func)(struct timer\_list \*), unsigned int flags, const char \*name, struct lock\_class\_key \*key)

{

debug\_init(timer);

do\_init\_timer(timer,func,flags,name,key);

}

The function ‘do\_init\_timer()’ was defined in the same file ‘/kernel/time/timer.c’ and I think this function is used only by this file as it’s function prototype was declared in this file before defining it. The function is defined as follows:

static void do\_init\_timer(struct timer\_list \*timer, void (\*func)(struct timer\_list \*), unsigned int flags, const char \*name, struct lock\_class\_key \*key)

{

timer->flags.pprev=NULL;

timer->function=func;

if(WARN\_ON\_ONCE(flags & ~TIMER\_INIT\_FLAGS))

{

Flags &= TIMER\_INIT\_FLAGS;

}

timer->flags= flags | raw\_smp\_processor\_id();

locked\_init\_map(&timer->lockdep\_map,name,key,0);

}

Thus, I could now use ‘timer\_setup()’ to setup a timer.

**To execute the timer every minute:** Now, I had to be able to set the timer’s timeout to 1 minute and then I would call this again everytime it expired. Thus, it would run indefinitely.

In file /kernel/time/timer.c’, there’s a function ‘mod\_timer()’ which according to the comments, modifies a timer’s timeout and is equivalent to deleting the timer using ‘del\_timer()’ and then adding a timer. The function ‘mod\_timer()’ is defined as follows:

Int mod\_timer(struct timer\_list \*timer, unsigned long expires)

{

return \_\_mod\_timer(timer, expires, 0);

}

The function ‘\_\_mod\_timer()’ was defined in the same file as follows:

static inline int \_\_mod\_timer(struct timer\_list \*timer, unsigned long expires, unsigned int options)

{

//Lot’s of code. After reading some of the comments, I’m thinking that since I am going to modify the timer’s timeout by the same amount, this function will simply extend the expiry time inorder to avoid the whole “dequeu/enqueue dance”.

}

-> msecs\_to\_jiffies() is a function defined in ‘/include/linux/jiffies.h’ file as follows:

static \_\_always\_inline unsigned long msecs\_to\_jiffies(const unsigned int m)

{

If(\_\_builtin\_constant\_p(m))  
 {

If((int)m <0)

Return MAX\_JIFFY\_OFFSET;

return \_msecs\_to\_jiffies(m);

}

else

{

return \_\_msecs\_to\_jiffies(m);

}

}

Thus, I am able to now infinitely call the required callback function that implements the above functionality of printing a msg whenever the system’s time is a multiple of 5 minutes.

The following is the code for writing the module to implement the above functionality.

mkdir print\_multiples\_of\_5time

cd print\_multiples\_of\_5time

vim time\_checker.c 🡺This file has all the code to implement the above functionality.

Inside this file, the following is the code:

“”””

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/timer.h> //To use the macro ‘timer\_setup’ which ultimately set’s up a timer.

#include<linux/time.h> //To use the ‘time64\_to\_tm()’ which converts the seconds into proper broken down time.

#include<linux/timekeeping.h> //To use the ‘ktime\_get\_real\_ts64()’ which helps get the current time.

MODULE\_DESCRIPTION(“Kernel module to print a msg whenever system’s time’s minutes is a multiple of 5. The timer checks every minute instead of being handled on an interrupt kind of basis”);

MODULE\_AUTHOR(“Vrajnandak Nangunoori”);

MODULE\_LICENSE(“GPL”);

static struct timer\_list my\_timer; //This struct is a timer object used for scheduling and managing timed events within the kernel. This will be used o execute our function every minute to check if the system’s time’s minute is a multiple of 5. Declaring it as a static variable sort of helps in optimization because then we won’t have to continuously redeclare the timer when the current timer expires and can instead continue using this by simply changing it’s timeout value by 1 minute. Since we are changing the timeout by only 1 minute, the code that changes the timeout value is also optimized so that the when the timeout is changed by the same amount, it’s expiry time is only changed without dequeueing it or enqueuing it.

//A simple function to modify the timer’s timeout by 1 minute.

void reschedule\_my\_timer\_VN(void)

{

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(60\*1000)); //msecs\_to\_jiffies() is a function in ‘/include/linux/jiffies.h’.

}

//The function to be called repeatedly, which implements the functionality of checking if the system’s time Is a multiple of 5 minutes. Ideally, all the functions being invoked should be handled properly but I’ve ignored them as I think that ignoring them won’t do any harm.

void timer\_callback(struct timer\_list \*timer)

{

//Declaring the structs to store current time.

struct timespec64 curr\_timer; //Used by ‘ktime\_get\_real\_ts64()’.

struct tm timeinfo;

//Getting the current system time in seconds.

ktime\_get\_real\_ts64(&curr\_time);

//Converting the time in seconds to local broken-down time.

time64\_to\_tm(curr\_time.tv\_sec,0,&timeinfo);

//Checking if the time is a multiple of 5 minutes

if(timeinfo.tm\_min % 5 ==0)  
 {

printk(KERN\_INFO “Current time is %d:%d – Is a multiple of 5 minutes\n”,timeinfo.tm\_hour,timeinfo.tm\_min);

}

//Rescheduling the timer to run again after 1 minute by changing it’s timeout by 1 minute.

reschedule\_my\_timer\_VN();

}

static int \_\_init print\_multiple\_of\_5\_init(void)

{

printk(KERN\_INFO “Initialization of 5 minute multiple checker module\n”);

//Initializing the timer.

timer\_setup(&my\_timer, timer\_callback,0); //I was able to deduce from the ‘init\_timer\_key()’ function as to what the data types of the parameters are.

//Setting the timeout of the timer to 1 minute.

reschedule\_my\_timer\_VN();

return 0; 🡺This return 0; is important as it tells the return status of this init() function.

}

static void \_\_exit print\_multiple\_of\_5\_exit(void)

{

printk(KERN\_INFO “Exiting print\_multiple\_of\_5\_module\n”);

//Deleting the timer.

del\_timer(&my\_timer);

}

module\_init(print\_multiple\_of\_5\_init);

module\_exit(print\_multiple\_of\_5\_exit);

“”””

-> The makefile is as follows:

“”””

obj-m = time\_checker.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

“”””

Now, run the following commands:

‘make’

‘sudo insmod time\_checker.ko’

Doing a ‘sudo dmesg’ every minute or when the system’s time is a multiple of 5 minutes will let you see the printk() statement that you’ve put. You will notice a difference of approximately 300 seconds between these printk() statements. The small time difference is due to the different kernel activities going on in the background.

Module parameters: They are configuration variables that allow you to customize the behaviour of kernel modules without recompiling the kernel or modifying the module’s source code. They provide a flexible way to adjust module settings at runtime, making the kernel more adaptable to different environments and use cases.

In file ‘/include/linux/moduleparam.h’, there are macros:

-> ’\_\_module\_param\_call’ which according to comments is the fundamental function for registering boot/module parameters.

-> ‘\_\_module\_param\_string’ which according to comments, copies the string when it’s set.

-> ‘module\_param’ which according to comments is a typesafe helper for a module/cmdline parameter.

-> ‘module\_param\_named’ which according to comments is a typesafe helper for a renamed module/cmdline parameter.

**->Tracing functions:**

**Reference:** [**https://www.kernel.org/doc/html/v5.15/trace/kprobes.html**](https://www.kernel.org/doc/html/v5.15/trace/kprobes.html)

**->Using ‘kprobes’ to trace functions:**

Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address, specifying a handler routine to be invoked when the breakpoint is hit. There are currently two types of probes: kprobes, and kretprobes (also called return probes). A kprobe can be inserted on virtually any instruction in the kernel. A return probe fires when a specified function returns

Note: In the context of debugging and tracing, single-stepping refers to executing a program one instruction at a time. This allows for detailed inspection of the program's state and behavior at each step.

The ‘/include/linux/ftrace.h’ file includes the ‘/include/linux/kallsyms.h’. ‘kallsyms’ extracts all kernel symbols and symbols exported from modules, constructs a list of the sections, symbols and their addresses and writes a relocatable object containing just the \_\_kallsyms section. It is more of a mechanism or facility.

We will now create a module ‘trace\_using\_kprobe’ to trace the first 100 functions (traceable and of only one function whose name you specify) that are invoked. We will write some information such as the ‘timestamp’,’func\_name’,’address’,’ip’,’flags’ into a file.

In a file ‘krpobe\_one\_function.c’, write the following:

“”””

#include<linux/kernel.h>

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kprobes.h>

#include<linux/kallsyms.h> //for macro ‘KSYM\_NAME\_LEN’, used for length of symbol. It stand for ‘Kernel Symbol Name Length’.

#include<linux/slab.h> //I don’t think this is needed but still.

#include<linux/uaccess.h> //I don’t think this is needed but still.

#include<linux/time.h> //same reason as above’s module which prints multiple of 5 minutes.

#include<linux/timer.h> //same reason as above’s module which prints multiple of 5 minutes.

#include<linux/timekeeping.h> //same reason as above’s module which prints multiple of 5 minutes.

MODULE\_DESCRIPTION(“A simple module that writes the timestamp, name,address,flags, and instruction pointer, for the probed instruction to a file. This happens only ‘MY\_MAX\_FUNCTIONS\_TRACED’ number of times. The probed instruction is set to ‘’vfs\_write” by default, change by modifying this .C file’s value for that variable or from command line while loading the module into the kernel. The information is written into the file only once the module has been unloaded.”);

MODULE\_AUTHOR(“Vrajnandak Nangunoori”);

MODULE\_LICENSE(“GPL”);

#define MY \_TRACE\_FILENAME “my\_kprobe\_trace.txt”

#define MAX\_FUNCTIONS\_TRACEABLE 100

struct my\_trace\_record

{

char timestamp[8]; // “ HH:MM “ format

char func\_name[KSYM\_NAME\_LEN];

kprobe\_opcode\_t \*address; //This is the data type of ‘addr’ field in ‘struct kprobe’ definition in file ‘/include/linux/kprobes.h’. Looking at it’s definition by typedef, using ‘int \*’ as the data type for this ‘address’ should also work but It didn’t work for me for some reason.

unsigned long instruction\_pointer; //This data type is from the definition of ‘struct pt\_regs’. There are lots of header files having definitions of ‘struct pt\_regs’ but they mostly had ‘unsigned long’ as the data type for ‘ip’, ‘flags’. Luckily, this worked for me.

unsigned long flags;

//We could also implement ret\_val field here but we would need a proper way to track the individual record from ‘mytrace\_records’ array.

//int ret\_val;

}

static char symbol[KSYM\_NAME\_LEN]=”vfs\_write”; //Name of the function to be probed.

module\_param\_string(symbol,symbol,sizeof(symbol),0644); //Making the symbol a module parameter.

static struct kprobe kp; //The probe for tracing.

static file \*my\_trace\_file; //Information to be written to this file pointer.

static struct my\_trace\_recrod records[MAX\_FUNCTION\_TRACEABLE];

static int my\_trace\_counter;

//Kprobes have 2 points, one for the entry and another for the exit. We can specify our own callback function to be called at each of these points.

//This will be called as soon as the function is invoked.

static int \_\_kprobes my\_pre\_handler(struct kprobe \*p, struct pt\_regs \*regs)  
{

struct timespec64 curr\_time;

struct tm timeinfo;

struct my\_trace\_record \*record\_now;

if(my\_trace\_counter<MAX\_FUNCTIONS\_TRACEABLE)

{  
 record\_now=&records[my\_trace\_counter];

my\_trace\_counter+=1;

//Getting the timestamp.

ktime\_get\_real\_ts64(&curr\_time);

time64\_to\_tm(curr\_time.tv\_sec,0,&timeinfo);

snprintf(record\_now->timestamp,8,” %02d:%02d “,timeinfo.tm\_hour,timeinfo.tm\_min);

//Getting function name, address

snprintf(record\_now->func\_name,KSYM\_NAME\_LEN,p->symbol\_name);

record\_now->address=p->addr;

//Getting the instruction pointer and flags.

record\_now->instruction\_pointer=regs->ip;

record\_now->flags=regs->flags;

printk(KERN\_INFO “Exiting the pre handler of %s \n”,p->symbol\_name);

}

else

{

unregister\_krpobe(&kp); //Since we won’t trace any more functions after the limit. Unregistering will remove the extra overhead of having to go through this function.

}

return 0;

}

//This will be called just before the function exits.

static void \_\_kprobes my\_post\_handler(struct kprobe \*p, struct pt\_regs \*regs, unsigned long flags)

{

printk(KERN\_INFO “In posthandler of %s, pid: %d\n”,p->symbol\_name,p->pid);

}

static int \_\_init my\_kprobe\_init(void)  
{  
 int ret;

my\_trace\_counter=0;

printk(KERN\_INFO “Initializing module for tracing using kprobes\n”);

//Initializing the file to put the information in.

my\_trace\_file=filp\_open(MY \_TRACE\_FILENAME, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if(IS\_ERR(my\_trace\_file))  
 {  
 printk(KERN\_ALERT “Failed to open file: %s\n”, MY\_TRACE\_FILENAME);

return PTR\_ERR(my\_trace\_file);

}

//Initializing the kprobe and setting the pre and post handler.

kp.pre\_handler=my\_pre\_handler;

kp.post\_handler=my\_post\_handler;

kp.symbol\_name=symbol;

//registering the kprobe.

ret=register\_kprobe(&kp);

if(ret<0)

{

pr\_err(“Registering kprobe failed, returned: %d\n”, ret);

return ret;

}

printk(KERN\_INFO “Planted kprobe at %p\n”,kp.addr);

return 0;

}

static void \_\_exit my\_kprobe\_exit(void)  
{

int err;

int counter;

int write\_error\_occured;

char trace\_rcd\_buffer[256];

struct my\_trace\_record \*curr\_rcd;

int to\_write;

//Unregistering the kprobe as we are unloading the module.

unregister\_kprobe(&kp);

printk(KERN\_INFO “kprobe at %p has been unregistered. Now, writing the data to the file.\n”, kp.addr);

//Setting the number of records to be written as min(MAX\_FUNCTIONS\_TRACEABLE, my\_trace\_counter).

to\_write=MAX\_FUNCTIONS\_TRACEABLE;

if(to\_write<my\_trace\_counter-1)

{

to\_write=my\_trace\_counter;

}

//Writing the data to the file.

for(counter=0;counter<MY\_MAX\_FUNCTIONS\_TRACED;counter++)

{

curr\_rcd=&mytrace\_records[counter];

snprintf(trace\_rcd\_buffer, 256, “timestamp: %s, name: %s, address: 0x%p, instruction ptr: %lx, flags: %lx”, curr\_rcd->timestamps, curr\_rcd->func\_name, curr\_rcd->address, curr\_rcd->instruction\_pointer, curr\_rcd->flags);

err=kernel\_write(my\_kernelclone\_trace\_file,trace\_rcd\_buffer,256,&my\_kernelclone\_trace\_file->f\_pos); //I used ‘vfs\_write()’ first but for some reason, on making it, I got an error saying ‘undefined reference’. So I searched the ‘/include/linux/fs.h’ file for other write functions and the first one I came across that looked like the one I needed was ‘kernel\_write()’ and so I used it. Luckily, it worked.

if(err<0)  
 {

write\_error\_occured=1;

}

}

filp\_close(my\_kernelclone\_trace\_file, NULL); //Closing the file.

if(write\_error\_occured)

{

printk(KERN\_ALERT “Error occurred while doing a write to file: %s\n”, MY\_KERNELCLONE\_TRACE\_FILENAME);

return;

}

printk(KERN\_INFO “Successfully wrote probed information to file: %s\n”, MY \_TRACE\_FILENAME);

}

module\_init(my\_kprobe\_init);

module\_exit(my\_kprobe\_exit);

“”””

Write a simple makefile as shown in the above ‘Module’ section.

In order to use grep on the text file, you can use ‘strings’ as the txt file is a binary file.

In order to change the function being traced, do

‘sudo insmod my\_kprobe\_trace\_module.ko symbol=<function\_name\_to\_be\_traced>’ //Replace <function\_name\_to\_be\_traced> with the function you want to trace. This has to be a traceable function like ‘schedule’, ‘vfs\_read’ etc.

**->Using ‘ftrace’ to trace functions:**

Since kprobes put a breakpoint In the function, it becomes impossible to probe a function without knowing it’s name or address. There is also no special symbol or character that allows to probe a function without specifying it’s name.

Hence, ftrace is often used to trace functions dynamically(without having to know the function name). They allow to filter functions and trace the filtered functions. They also have a entry point at the start and end of a function but you can only set one at a time. By default, this point is set to the start of the function.

Here is a small module to trace the first 100 functions invoked by the kernel.

“”””

#include<linux/kernel.h>

#include<linux/init.h>

#include<linux/module.h>

#include<linux/ftrace.h>

#include<linux/kallsyms.h>

#include<linux/errno.h>

#include<linux/slab.h>

#include<linux/sched.h>

MODULE\_DESCRIPTION(“A simple module to trace the first 100 functions invoked”);

MODULE\_AUTHOR(“Vrajnandak Nangunoori”);

MODULE\_LICENSE(“GPL”);

#define MAX\_FUNCTIONS\_TRACEABLE 100

static struct ftrace\_ops my\_ftrace\_ops; //To be able to set the ftrace points with our callback.

static void notrace my\_trace\_callback(unsigned long ip, unsigned long parent\_ip, struct ftrace\_ops \*ops, struct ftrace\_regs \*fregs)

{

int bit=0;

static int trace\_count=0;

bit=ftrace\_test\_recursion\_trylock(ip,parent\_ip); //So that there is no infinite recursion of the callback function.

if(bit<0)

return;

if(trace\_count<MAX\_FUNCTIONS\_TRACEABLE)

{

pr\_info(“Function traced: %ps\n”,(void \*)ip); //A shorthand to print the name of function using ip.

trace\_count++;

}

ftrace\_test\_recursion\_unlock(bit);  
}

static int \_\_init my\_mod\_init(void)  
{  
 int err;

int ret;

//Registering the ftrace.

my\_ftrace\_ops.func=my\_trace\_callback;

my\_ftrace\_ops.flags=FTRACE\_OPS\_FL\_SAVE\_REGS | FTRACE\_OPS\_FL\_RECURSION;

//Filtering the functions. Here the following arguments allow to trace any function that is traceable.

ret=ftrace\_set\_filter(&my\_ftrace\_ops, NULL,0, 1);

//ret=ftrace\_set\_filter(&my\_ftrace\_ops, “schedule”,strlen(“schedule”), 0); //To trace only ‘schedule’ function.

err=register\_ftrace\_function(&my\_ftrace\_ops);

if(err)

{

pr\_err(“Failed to register ftrace, errro: err\n”,err);

return err;

}

pr\_info(“Successfully registered ftrace function\n”);

return 0;

}

static void \_\_exit my\_mod\_exit(void)  
{

pr\_info(“Unregistering ftrace function\n”);

unregister\_ftrace\_function(&my\_ftrace\_ops);

}

module\_init(my\_mod\_init);

module\_exit(my\_mod\_exit);

“”””

To run, use the above makefile and the ‘sudo insmod module\_name.ko’.

**OTHER THINGS TO TRY OUT:  
-> Another thing to try out is ‘ebpf’ for tracing functions. Given the time constraints, I was unable to do it.**