SOME IMPORTANT RESOURCES: <https://www.kernel.org/doc/html/latest/process/adding-syscalls.html?highlight=syscall_define>

Project Aim: To understand a bit more about the linux kernel and be able to make changes to it and compile it. At the end, be able to create a new system call, create a module etc.

While installing Virtual Box:

1. Sudo apt install bzip2

WEEK 1: Compiling the kernel

<https://davidaugustat.com/linux/how-to-compile-linux-kernel-on-ubuntu>

Sudo apt update: It’s a good thing to run this before installing any other packages because this will ensure that the latest packages will be installed. This also helps avoid things like trying to install a package that has been moved or updated in the repository. It refreshes the local package cache.

1. sudo apt install build-essential libncurses-dev bison libssl-dev libelf-dev fakeroot:

so, ‘apt’ is a package manager to install multiple packages on a Debian-based Linux distribution(like Ubuntu).

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).

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.

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.

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).

Libelf-dev: This package provides the development files (headers and libraries) for working with the Executable and Linakable Format(ELF).

Fakeroot: This tool simulates superuser(root) privileges for the purpose of building

‘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’).

wget <https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.6.31.tar.xz> :

downloads the linux kernel source code for version 6.6.31

tar -xf linux-6.6.31.tar.xz :  
extracts the contents of the linux archive into a directory named ‘linux-6.6.31’

make localmodconfig :  
generates a kernel configuration based on currently loaded modules in the system. This creates a .config file in the kernel source directory.

Encountered the following error on ‘make localmodconfig’ :

HOSTCC scripts/basic/fixdep

HOSTCC scripts/kconfig/conf.o

HOSTCC scripts/kconfig/confdata.o

HOSTCC scripts/kconfig/expr.o

LEX scripts/kconfig/lexer.lex.c

/bin/sh: 1: flex: not found

make[2]: \*\*\* [scripts/Makefile.host:9: scripts/kconfig/lexer.lex.c] Error 127

make[1]: \*\*\* [/mnt/d/LMS\_IIITBangalore/WSL\_C

HOW I FIXED IT: ‘sudo apt install flex’ to download flex since my system didn’t have it downloaded.( <https://askubuntu.com/questions/1245882/kernel-make-menuconfig-error>).

I tried running ‘make loclmodconfig’ again, it ran for about 10 minutes(not sure why this long). The system kept prompting me a lot of times, I kept hitting ‘Enter’ as instructed in the reference given by sir (<https://davidaugustat.com/linux/how-to-compile-linux-kernel-on-ubuntu>) and finally the system printed ‘configuration written to .config’. After another 2 minutes, it finally ended executing.

I simply ran ‘fakeroot make’ instead of ‘fakeroot make -j8’ since I didn’t want any trouble with my system. Started at ‘4:49’ am and it looked like It was going to take a while so I decided to force stop it using ‘control C’ which gave error:

^Cmake[4]: \*\*\* Deleting file 'kernel/dma/mapping.o'

make[4]: \*\*\* [scripts/Makefile.build:243: kernel/dma/mapping.o] Error 130

make[3]: \*\*\* [scripts/Makefile.build:480: kernel/dma] Interrupt

make[2]: \*\*\* [scripts/Makefile.build:480: kernel] Interrupt

make[1]: \*\*\* [/mnt/d/LMS\_IIITBangalore/WSL\_C\_py/extra\_coding\_stuff/linux-6.6.31/Makefile:1913: .] Interrupt

make: \*\*\* [Makefile:234: \_\_sub-make] Interrupt

and then I ran a ‘fakeroot make clean’ which gave an error :  
make[2]: \*\*\* Documentation/Kbuild: Is a directory. Stop.

make[1]: \*\*\* [/mnt/d/LMS\_IIITBangalore/WSL\_C\_py/extra\_coding\_stuff/linux-6.6.31/Makefile:1918: \_clean\_Documentation] Error 2

make: \*\*\* [Makefile:234: \_\_sub-make] Error 2  
  
I didn’t bother checking it out and just ran ‘fakeroot make -j8’. It started at 5:50am and ended at 6:37 am giving the following error:   
make[1]: \*\*\* [/mnt/d/LMS\_IIITBangalore/WSL\_C\_py/extra\_coding\_stuff/linux-6.6.31/Makefile:1913: .] Error 2

make: \*\*\* [Makefile:234: \_\_sub-make] Error 2

I tried running ‘fakeroot make -j8’ again after simply doing ‘echo #?’

net/ipv4/netfilter/ipt\_ECN.c:26:46: warning: ‘struct ipt\_ECN\_info’ declared inside parameter list will not be visible outside of this definition or declaration

26 | set\_ect\_ip(struct sk\_buff \*skb, const struct ipt\_ECN\_info \*einfo)

WEEK 2: Simulating a logging function of sorts which prints all the calls/activities performed by the system.

1. ‘sudo cat kmsg’:

WEEK 3: Make changes to the Scheduler. When a process is doing IO operation, it needs to get blocked, understand how it happens. After the operation is done, understand how the process is unblocked.

Ubuntu 22.04

50 GB disk space

Linux 5.19.0

I tried to download the above Ubuntu and after logging in and trying to use the ‘sudo’ command, it kept saying “vrajnandak is not in the sudoers file. This incident will be reported.”. So I added the following line ‘vrajnandak ALL=(ALL:ALL) ALL’ inside the ‘/etc/sudoers’ file. I think this added the name ‘vrajnandak’ to the list of sudo users.

I tried to download the same linux version

wget -c https://kernel.ubuntu.com/~kernel-ppa/mainline/v5.19/amd64/linux-headers-5.19.0-051900\_5.19.0-051900.202207312230\_all.deb

but it gave me an error ‘404 not found’. So I decided to download the 5.15.160 version. ‘wget <https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.15.160.tar.xz>’

I now unzipped using ‘tar -xf’. Changed to that directory, ran ‘make localmodconfig’. Made the scripts config modification and finally ran ‘fakeroot make’.

(I went to the settings in ubuntu by clicking the top right corner of the screen, and changed the ‘screen blank’ option to ‘never’ so that my ubuntu doesn’t sleep after some time of inactivity.)

The ‘fakeroot make’ command showed a msg while running “Now generating an x.509 key pair to be used for signing modules. If this takes a long time you might wish to run rngd in the background to keep the supply of entropy topped up. It needs to be run as root, and uses a hardware random generator if one is available.”

Luckily, it generated a key pair (not sure how long it took because I just left the computer on).

The ‘fakeroot make’ command ran successfully, ‘echo $?’ gave an output of 0.

Inorder to boot into a new kernel, you need to change the file ‘/etc/default/grub’ because the default values for the grub reboot would be ‘hidden’, ‘0’ for the variables ‘GRUB\_TIMEOUT\_STYLE’ and ‘GRUB\_TIMEOUT’. Use the command ‘sudo nano /etc/default/grub’ and then change value ‘GRUB\_TIMEOUT\_STYLE’ to ‘menu’ because we want a menu of options and then change value of ‘GRUB\_TIMEOUT’ to ’10’ (or possibly something more?). Now ‘sudo update-grub’. Now, you can ‘sudo reboot’ and you will get a list of options of the kernels you can boot into.

/\*\*\*\*\*\*Using the first option that appears in menu for 5.15.160\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Now, after making change to ‘include/linux/uts.h’ where u change the name from ‘Linux’ to ‘NanduOS’, you will run ‘fakeroot make’ (I started at 6:43 and ended before 6:57 but not sure at what exact time), and then ‘sudo make modules\_install’, ‘sudo make install’, and then ‘sudo reboot’.

I noticed in the menu that was a new option but with the word ‘\_old’ appended to the linux version that I downloaded (5.15.160) but I booted into the version without the ‘\_old’ in it.

Successful execution of ‘uname -rs’, showed ‘NanduOS’.

* Adding a printk() statement in the do\_syscall\_64() function might not be a good decision because if you were to do a ‘dmesg’ on the terminal then every time that the ‘dmesg’ reads from the kernel ring buffer, it would invoke another syscall to read from the buffer which would inturn execute the printk() statement in the common entry point(do\_syscall\_64()) and so what happens is that the ‘dmesg’ continues to read forever because every time it reads from the buffer, it ends up writing to that buffer due to the different system calls it might execute. I haven’t confirmed it but it is a possibility.

//////////////////COMMON.C//////////////////

Now, I went to file ‘arch/x86/entry/common.c’ and in the function ‘do\_syscall\_64’ wrote a printk() statement printing the value of ‘nr’ after it gets the return value from the ‘syscall\_enter\_from\_user\_mode()’ function. I think ‘nr’ is no way related to the syscall number that is associated with a system call but rather to just check if the syscall is a valid one or not.

//////////////arch/x86/entry/syscall\_64.c////////////

Functionality of the single function in the file is: based on value of ‘nr’, return \_\_x64\_##sym(regs). ‘##’ is the ‘token-concatenation’ operator and so it returns ‘\_\_x64\_sys\_read(regs);’ or something like that. Now this looks awfully close to a syscall, so we should search for where these kinds of function names are present. The only header files it included are ‘linux/linkage.h’,’linux/sys.h’,’linux/cache.h’,’linux/syscalls.h’,’asm/syscall.h’. So I went through all of them quickly:

* Linux/linkage.h: ‘./include/linux/linkage.h’ found using the find command. Didn’t find anything that looked related. Also checked the file ‘arch/x86/include/asm/linkage.h’ but nothing looked related either. I noticed that ‘asmlinkage’ is a macro and is defined to be ‘CPP\_ASMLINKAGE’ which is either just defined or has value ‘extern “C”’. Not sure how this would be used later on.
* Linux/sys.h: ‘/include/linux/sys.h’ was a file which was basically full of comments and had some syscall names that haven’t been implemented yet but have an entry in the table for future expansion.
* Linux/cache.h: Didn’t check this file as I felt that cache would be related to memory itself rather than the functions.
* Linux/syscalls.h: in the file ‘include/linux/syscalls.h’, I first saw that it had a lot of struct declarations(probably to use them later on???). There was a comment saying that it may be useful for an arch to override the definitions of the SYSCALL\_DEFINE0() and \_\_SYSCALL\_DEFINEx() macros, in particular to use a diff calling convention for syscalls. It then had a very long macro named ‘SYSCALL\_TRACE\_ENTER\_EVENT(sname)’ which I couldn’t really understand what it exactly did because the structs used inside the definition of the macro weren’t declared in the file. Other macros were ‘SYSCALL\_TRACE\_EXIT()’, ‘SYSCALL\_METADATA()’, etc. In line 209, there was a macro for ‘SYSCALL\_DEFINE0’ and was defined to do an ‘asmlinkage long sys\_##sname(void);’ (so I think since CPP\_ASMLINKAGE value would not be anything, ‘asmlinkage long’ would just become long after pre-processing is done) and another asmlinkage . Starting from line 316 there were a lot of syscall function declarations in the form ‘asmlinkage long syscall\_name();’ where syscall\_name was like ‘sys\_io\_setup’,’sys\_io\_destroy’ etc. As I kept scrolling through the file, at line 497, I saw the comment ‘fs/read\_write.c’ and below it were read, write, lseek etc syscall declarations (but they were like ‘asmlinkage long sys\_read(unsigned int fd,char \_\_user \*buf, size\_t count);’). At line 686, there was ‘asmlinkage long sys\_syslog();’ with a comment ‘kernel/printk.c’. Starting from line 693, there were some functions that looked related to scheduling and had a comment ‘kernel/sched/core.c’. At line 740, there were functions based on processes I think and they had a comment ‘kernel/sys.c’, from line 783 as well. At line ‘1280’, there was a comment telling that the kernel code shdn’t call syscalls(i.e., sys\_xyzyyz()) directly and instead shd use one of the functions which work equivalently, such as the ksys\_xyzyyz() functions which were prototyped in this same file towards the end of the file. At line ‘1292’, there was ksys\_pread64(); and in the next line there was ksys\_pwrite64(); which looked awfully familiar to the read(),write() calls we use and they even had a return value of ‘ssize\_t’ datatype.

SPECIAL MENTION: while searching for the above kind of function, I saw a file ‘/arch/x86/entry/syscalls/syscall\_64.tbl’ which had all the 449 syscalls in a tabular looking format. There was even a comment saying “don’t use numbers 387 through 423, add new calls after the last ‘common’ entry”. It was also mentioned in the lower section of the file that due to a historical design error, in x32, certain syscalls are numbered differently as compared to the native x86\_64 whose syscalls have numbers 512-547.

In the same file ‘arch/x86/entry/syscall\_64.c’ there was a comment that mentioned “sys\_call\_table[] is no longer used for system calls but ‘kernel/trace/trace\_syscalls.c’ still wants to know the system call address.”. So, I decided to check that file out and I found the ‘syscall\_enter\_register’ and ‘syscall\_exit\_register’ functions defined to be static and return int value. Wasn’t too sure how they’re being called and where to they’re returning.

I then tried to find where the ‘asm/syscalls\_64.h’ file is located, which was mentioned in the ‘arch/x86/entry/syscall\_64.c’ file at by running ‘grep -r “syscalls\_64.h” .’ but it looked like it wasn’t going to work and so I decided to use ‘find . -name “syscalls\_64.h”’ which gave me output:  
./arch/x86/um/shared/sysdep/syscalls\_64.h

./arch/x86/include/generated/asm/syscalls\_64.h

Since, I was searching for ‘asm/syscalls\_64.h’, I checked out the 2nd file only.

On opening it with vim, I saw 449 lines of code with each line being of the form:

‘\_\_SYSCALL(sys\_num,syscall\_name)’ where ‘sys\_num’ ranged from 0 to 448 and ‘syscall\_name’ were just the names of different system calls I think because of the names like ‘sys\_read’,’sys\_write’,’sys\_open’ and so on. The names ‘sys\_num’ and ‘syscall\_name’ are my interpretations of what the lines of code were looking like and this may not be correct. The following are the 1st two lines in that file:  
\_\_SYSCALL(0,sys\_read)

\_\_SYSCALL(1,sys\_write)

Now, In file ‘arch/x86/include/asm/syscall.h’, there was a comment telling to see asm-generic/syscall.h for descriptions of what must be done in this said file. So I did a ‘find . -name ‘syscall.h’ and found the following line from the output of find:  
./include/asm-generic/syscall.h

So in file ‘include/asm-generic/syscall.h’, I saw that syscall\_get\_nr finds out what system call a task is executing and returns the syscall number or -1 as needed. They mentioned that in case of x64, a valid call might also get truncated to -1 as the syscall\_get\_nr returns an int (which is 32 bits). Some other functions were also defined such as ‘syscall\_rollback’, ‘syscall\_get\_error’, ‘syscall\_get\_return\_value’,’syscall\_set\_return\_value’,’syscall\_get\_arguments’,’syscall\_set\_arguments’,’syscall\_get\_arch’.

Now, with nothing in hand I checked out the ‘fs’ directory to look up the definitions of the sys\_read(),sys\_write() functions if possible. There was a comment saying that SEEK\_SET and SEEK\_END are unsynchronized but SEEK\_CUR is synchronized against other SEEK\_CURs, but not read/writes. Read/writes behave like SEEK\_SET against seeks. There was a function ‘vfs\_read’ at line 465, not sure what it did. There was ‘vfs\_write’ at line 574. The ksys\_read() definition at line 611 uses vfs\_read() and there was a line below the end of the function definition ‘SYSCALL\_DEFINE3(read,unsigned int, fd, char \_\_user \*, buf, size\_t, count) which is basically going to be the sys\_read definition. The ksys\_write() definition at line 636 uses vfs\_write() to write to file ig. The ksys\_pread64() at line 662 also called ‘vfs\_read()’, at this point I’m not sure what the difference between these functions is.

So, sys\_read() calls ksys\_read() which calls vfs\_read()

Similarily, sys\_write() calls ksys\_write() which calls vfs\_write()

**Steps I followed to add a new syscall:**

Made a new entry in ‘arch/x86/entry/syscalls/syscall\_64..tbl’: “449 common print\_nandu sys\_nandu”

Since the function is just going to print ‘nandu’, I’ll be using SYSCALL\_DEFINE0() and this would generate a system call of form ‘sys\_nandu’, that’s why we had to declare in the above file as ‘sys\_nandu’. I added the definition in ‘kernel/sys.c’:  
SYSCALL\_DEFINE0(nandu)  
{  
printk(KERN\_INFO “Hello, this is printing from NanduOS\n”);

Return 0;  
}

Now, I added the line ‘asmlinkage long sys\_nandu(void);’ to the file ‘include/linux/syscalls.h’.

Now, I regenerated the header files using command ‘make headers\_install’.

On running fakeroot main, I got a warning in between ‘drivers/cpufreq/cpufreq\_ondemand.c: 446:1: warning: the frame size of 1032 bytes is larger than 1024 bytes.

‘sudo make modules\_install’ followed by ‘sudo make install’.

Now, sudo reboot.

I made a temporary .C file:  
  
#include<unistd.h>

#include<sys/syscall.h>

#include<stdio.h>

Int main()  
{

Long res=syscall(449); //Because 448 was the last use sys number.

Printf(“sys\_nandu ret avl: %ld\n”,res); //Should return 0 because that’s what we returned.  
return 0;

}

//After the syscall gets called, the printk shd be executed and so the kernel ring buffer would have the message that we wrote in the definition of our syscall in the file ‘kernel/sys.c’

Now, you can do a ‘dmesg | tail’ which will show the last 10 messages of the kernel ring buffer. If you get an error message saying ‘read kernel buffer failed: Operation not permitted’ then try ‘sudo dmesg | tail’.

I can affirm that it has run successfully for me.

On doing all the above steps, you would also notice a new line in the ‘/arch/x86/include/generated/asm/syscalls\_64.h’ file of form ‘\_\_SYSCALL(sysnum,sys\_sname)’ where sysnum is the syscall number you assigned and ‘sname’ is the name of the syscall, mine was ‘nandu’ for sname.

Now, thinking about why it didn’t run before, I can think of a good reason:

When I downloaded Ubuntu24.04, the ubuntu was running real slow and even opening the terminal took 10 minutes to register. Now, when I made change to the common.c file, I wrote a printk statement and I think it’s because I wrote it here that my computer looked like it was stuck. Because everytime it does a read syscall, it would do a printk and hence there would be more content to read and hence dmesg would do another read() which would only call another printk(). And so this infinite loop might have been too tough for my computer to handle with Ubuntu 24.04.(Not too sure why tho).

I got this realization of the infinite loop of reading,writing only after Downloading Ubuntu22.04 with linux-5.15.160. When I tried to boot into it, on my 2nd attempt I saw my screen being printed with continuous lines of the printk statement that I had added in the common.c file. So, I came to the conclusion that every read() call to the kernel ring buffer is only adding another line to read from that buffer and so the infinite loop.

So, I removed that line from the common.c file and left the changes in the other files untouched. I made a C file called ‘nandu.c’ in the same directory in which my downloaded linux source code was there. After compiling normally and running it, the command ‘sudo dmesg | tail’ displayed the printk() statement that I had added in my syscall.

SIDE NOTE: -> I think sir’s code is saying ‘undefined reference to mycall()’ because the return type has not been specified which is supposed to be long. ‘asmlinkage’ is a macro which is empty under the context of C but not under the context of C++, and so being empty would mean you haven’t specified the return type. Without having defined the return type of the new syscalls in the syscall.h file, the compiler might not be able to get the reference for the prototype and hence the error.

* Another thing I noticed is that sir is using mycall0() and not sys\_mycall0(). While building the functions and everything, most of the SYSCALL\_DEFINE0(), SYSCALL\_DEFINEx() macros always called/built functions of form sys\_sname() so maybe that’s why sir got the error of ‘undefined reference’ because the actual syscall that is present is ‘sys\_mycall()’ and not ‘mycall()’. I might be wrong but idk.

////Changing the Scheduler///

Notice that ‘/etc/modules’ is the file where we can put a module of our choice if we want that module to be loaded at boot time. This is with respect to the linux that comes with the Ubuntu22.04 and not the linux version that we downloaded.

In WSL, the file with credentials is ‘/etc/ssmtp/ssmtp.conf’ file.

The email I’ll be using is

‘[emailtesting1223@gmail.com](mailto:emailtesting1223@gmail.com)’

‘emailpassword’

->Trying to add a syscall to send a mail:

1. I noticed i didn't have smtp nor sendmail and so on searching the internet i saw 2 sites saying that ssmtp has deprecated and to use msmtp instead. Msmtp is a ligthweight SMTP client to send email via an SMTP server. It is often used as a simpler alternative to more complex mail transfer agents like 'sendmail' or 'postfix'. Msmtp-mta provides a sendmail-compatible interface for 'msmtp'. This will set up a '/usr/sbin/sendmail' file to the 'msmtp' executable.

Sudo apt update

sudo apt install msmtp msmtp-mta

(I got a message asking to enable AppArmor Support which i clicked on Yes)

Not sure why but the config file wasn't there in my computer. Searched for it using 'find . -name “msmtprc”' but couldn't find the file so i created the file manually.

Sudo vim /etc/msmtprc

sudo chmod 600 /etc/msmtprc

sudo touch /var/log/msmtp.log

sudo chmod 600 /var/log/msmtp.log

echo -e “Subject: Test email\n\nThis is test email.” | msmtep -a default thisisforchatgpt@gmail.com

But unfortunately, i still continue to get the error that the configuration file is not available. So, i decided to download sendmail.

<https://tecadmin.net/install-sendmail-on-ubuntu/>

Followed the steps in above link to try downloading and setting up sendmail but it didn’t work.

SPECIAL NOTE: your.smtp.server is found by command ‘nslookup -type=mx email.com’ and in the output, you will have a line with ‘mail exchanger’ under Non-authoritative answers. The last thing is the required your.smtp.server. For me it was ‘mx00.mail.com’

Followed separate steps:

Sudo vim /etc/mail/sendmail.mc

Added the define, FEATURE lines in the above file as shown in the link mentioned.

Added the line ‘AuthInfo:mx00.mail.com “U:emailtesting1223@gmail.com” “P:emailpassword” “M:PLAIN”’ in the file ‘/etc/mail/authinfo’

On trying to run ‘sudo makemap hash /etc/mail/authinfo < /etc/mail/authinfo’, I keep getting the error saying permission denied. I did the following commands thinking there might have been an issue with permissions:

Sudo chmod 600 /etc/mail/authinfo

Sudo chown root:root /etc/mail/authinfo

But the issue still persisted. So, I thought maybe the directory itself needed more permissions and did the following:

Sudo chmod 755 /etc/mail

Sudo chown root:root /etc/mail

But permission denied error continued to persist. So, I created a separate directory in the ‘/tmp’ directory thinking I would create the db file over there and move it back into the ‘/etc’ directory:

Sudo mkdir -p /tmp/mail

Sudo chmod 755 /tmp/mail

Sudo chown root:root /tmp/mail

Sudo vim /tmp/mail/authinfo

(Added the line: “AuthInfo:mx00.mail.com “U:emailtesting1223@gmail.com” “P:emailpassword” “M:PLAIN””)

Sudo chmod 600 /tmp/mail/authinfo

Sudo chown root:root /tmp/mail/authinfo

Sudo makemap hash /tmp/mail/authinfo < /tmp/mail/authinfo

I still got the error of permission denied.

I checked to see if makemap($which makemap) was present and it was present. I ditched the idea of Authentication and followed the steps numbered from 5 to end but didn’t receive any mail.

I’ve decided to try and download ssmtp even though people were saying that msmtp is deprecated. The only reason I didn’t move ahead with ssmtp in the first place is because I thought this was something that’s preinstalled as I didn’t have to install it explicitly in my WSL.

Sudo apt install ssmtp

Sudo vim /etc/ssmtp/ssmtp.conf

Configure the file now according the domain you want to use(I’m using gmail) and other authentication information that gmail needs, if any.

‘hostname’ on terminal to view your hostname.

You have to create an app password for gmail for your google account. It was tough just to find the app passwords page and to create one. Found the link to the app passwords page from the following link: <https://itsupport.umd.edu/itsupport?id=kb_article_view&sysparm_article=KB0015112>

But this won’t work unless you turn on 2-step verification and properly add a recovery phone and email. After that you have to generate an app password and you can use it to set the AuthPass field in the ‘/etc/ssmtp/ssmtp.conf’ file.

My configure file looks like this (my\_email\_name: ‘emailtesting1223@gmail.com’ and AuthPass: ‘vmsuuadybjyefoea’):

root=my\_email\_name

mailhub=smtp.gmail.com:465

hostname=Ubuntu22

UseTLS=YES

AuthUser=my\_email\_name

AuthPass=my\_app\_pass\_for\_that

Now, you can test using ‘ssmtp recipient\_mail < msg.txt’ where msg.txt is a file containing the email in following format:

To: recipient\_email

From: sender\_email

Subject: email\_subject

Mail\_content

It’s not mandatory to fill the To, From fields and they can be ignored if you want to. Similarily for Subject field unless you want to mention a subject field.

I simply tried to add the fields ‘FromLineOverride’ and ‘UseSTARTTLS’ but they didn’t work for some reason and gave error ‘ssmtp: Cannot open smtp.gmail.com:465’ and so I changed the port number from 465 to 587 (some sources on internet said this might be due to encryption issues on 465 not being secure enough) and surprisingly it works now.

Now that we’ve set up the prerequistes of sending a mail and know how to add a syscall, we can now create a syscall to send a mail. Probably use SYSCALL\_DEFINEe() or something as we’d need email\_name,auth\_pass,recipient\_email.

Since we want to be able to use dynamic sender,receiver emails we need to be able to change the contents of the file before actually sending the mail

I wrote a set\_sender.sh file to change the AuthUser,AuthPass fields of the ssmtp.conf file.

The file has the following code:

#!/bin/bash

If [ “$#” -ne 2 ]; then

Echo “Usage is: $0 <email> <pass>”

Exit 1

fi

EMAIL=$1

PASSWORD=$2

SSMTP\_CONF=”/etc/ssmtp/ssmtp.conf”

cp $SSMTP\_CONF $SSMTP\_CONF.bak #Creating a backup file just incase.

sudo sed -i “s/^AuthUser=.\*/AuthUser=$EMAIL/” SSMTP\_CONF

sudo sed -I “s/^AuthPass=.\*/AuthPass=$PASSWORD/” SSMTP\_CONF

On trying to run the file you might get an error saying permission denied which might probably be because it only has read,write perms. Do the following:

Sudo chmod +x set\_sender.sh

Now, you can take in the arguments use the following C code to change the AuthUser,AuthPass freely which required sudo permission:

#include<stdlib.h>

#include<stdio.h>

Int main(int argc,char argv[])

{

Char \*email=argv[1];

Char \*pass=argv[2];

Char command[256];

Snprintf(command,sizeof(command),”sudo %s %s %s”,file\_path,email,pass);

Int result=system(command);

If(result==-1)  
{  
perror(“system(command): “);

Exit(1);  
}

Return 0;

}

Now, we can call our syscall to make the call to

TO DO:

Figure out why sys\_nandu() doesn’t work but syscall(449) works.

Figure out the working of scheduler code.

I want to make a change in the kernel so that everytime a process with a pid>=8000 is selected, then it’s nice value is incremented by 5 and the pid,old\_policy,new\_policy,old\_nice\_val,new\_expected\_nice\_val,new\_nice\_val are all printed using printk().

Find . -name ‘sched’

Got the following output:

./include/linux/sched

./kernel/sched

/////////CHECKED THE BELOW FILE AFTER CHANGING INTO ‘/include/linux’ and ‘ls’////////

‘/Documentation’ has a few .rst(reStructuredText files) with information about the kernel. My understandings:

1. ‘/Documentation/process/coding-style.rst’: In line 487, they tell that instead of using the ‘extern’ keyword to functions, they use ‘EXPORT\_SYMBOL(func\_name);’. In line 781, they mention to use a ‘do-while(0)’ loop when defining function-like macros which have multiple statements(In this word document, I have mentioned the same about there being empty do-while(0) loops for macros, Ig it was just the convention being used for kernel development.)
2. ‘/Documentation/core-api/symbol-namespaces.rst’: Has a lot of information about the macros for exporting, namely, EXPORT\_SYMBOL,EXPORT\_SYMBOL\_GPL, EXPORT\_SYMBOL\_NS(),EXPORT\_SYMBOL\_NS\_GPL(). The latter 2 use an additional argument which is the namespace.
3. ‘/Document/scheduler’: Has a few .rst files related to some basic scheduler information.

NOTE: I just noticed but how exactly do the include files work. For example, ‘fs/read\_write.c’ has some include statements ex: ‘#include<linux/sched/xacct.h>’ but this header file is supposed to be a user written file and the ‘linux’ directory is in a different directory as well. I’m thinking that there might be a path to the ‘include’ directory and then whatever is specified inside the ‘<>’ tag is taken as the path to the header file to be included, if this is the case I would like to know how exactly such a path is specified and if we can make modifications to it.

Checked out ‘/include/linux/sched.h’. It had a lot of macros like ‘TASK\_RUNNING’, ‘task\_is\_running(task)’ etc. The lines starting from line 134 had 4 function-like macros ‘debug\_normal\_state\_change(state\_val)’, ‘debug\_special\_state\_change(state\_val)’, ’debug\_rtlock\_wait\_set\_state()’,’debug\_rtlock\_wait\_restore\_state()’ which were defined to be ‘empty do-while’ loops but the while condition was ‘while(0);’. Not sure if there was a particular reason for this where they could’ve instead just simply not put anything or maybe just put a ‘if’ statement. There was a macro ‘MAX\_SCHEDULE\_TIMEOUT’ defined to be ‘LONG\_MAX’ at line 282. Line 331 has a ‘struct vtime’ with a few time trackers. Line 358, there is a ‘struct sched\_info’ with 4 attributes for knowing the number of times (the process??) has run on the CPU, time spent waiting on runqueue, the last time it ran on CPU, last time it was queued to run. There are a lot more structs with information on times. ‘/include/linux/sched.h’ had a task\_nice() function which returned the nice value of the task that is passed to the function.

Checked out ‘/kernel/sched/sched.h’, it has some one line inline functions of form ‘x\_policy’ which return 1 or 0 based on the integer value(policy value) passed, and also one line inline functions of form ‘task\_has\_x\_policy’ which also return 1 or 0 by using the previously said inline functions by passing the p->policy value to these where p is a struct of type ‘task\_struct’. A macro ‘cap\_scale(v,s)’ which uses the ‘SCHED\_CAPACITY\_SHIFT’ macro defined in the ‘/include/linux/sched.h’ to return ‘((v)\*(s) >> SCHED\_CAPACITY\_SHIFT)’. SUGOV stands for ‘SchedUtil GOVernor’. ‘struct rt\_prio\_array’ is the priority\_queue data structure of the RT scheduling class, RT stands for ‘real-time’ I think but I’m not sure.

Noticed in ‘include/linux/types.h’ the different ‘typedef’ statements for all the data types needed in kernel code.

Checked ‘/kernel/sched/core.c’ where ‘sysctl\_sched\_rt\_period’ is defined to be the period over which the rt task CPU usage is measured. The ‘/kernel/sched/core.c’ file had an interesting function ‘pick\_next\_task()’ but it had 2 different definitions with each being under the if,else part of the ‘ifdef CONFIG\_SCHED\_CORE’. So, to determine whether this macro was defined or not, I did the following:

cscope -d

CONFIG\_SCHED\_CORE //In the ‘Find this C symbol’ option.

And thus I navigated to the file ‘/include/generated/autoconf.h’ where I saw ‘CONFIG\_SCHED\_CORE’ to be defined as 1. Just to be sure, I decided to check the config file(is a hidden file in the root directory) as well by doing ‘**grep CONFIG\_SCHED\_CORE .config**’ which output the line ‘CONFIG\_SCHED\_CORE=y’. I also did ‘**scripts/config –state CONFIG\_SCHED\_CORE**’ which output ‘y’ hence indicating that the CONFIG\_SCHED\_CORE is defined. This thus confirmed that the macro was defined and now I had essentially found the definition to the pick\_next\_task() function.

Had trouble finding out what ‘struct rq’ is. So searched the internet on how to navigate through C code easily. Found ‘cscope’ and downloaded it:

Sudo apt install cscope

cscope -Rbq //In the root directory only of the kernel source code.

cscope -d

(The above command gives a list of options. Choose the appropriate one. I went ahead with the ‘egrep’ option since the other appropriate options kept saying ‘This is not a C symbol: struct rq’).

I added the following:

In ‘/kernel/sched/core.c’ , I added the function ‘check\_and\_modify\_task()’ for the required functionality. I call this function everytime a new task is picked to be added to the runqueue and so this function is invoked in pick\_next\_task(). However, I think this function isn’t being called because I used fork() to create a process with a pid >8000 but still I couldn’t see the printk() statement. I continue to get a warning ‘ISO C90 forbids mixed declarations and code’ despite having tried ‘int’,’\_\_s32’, ‘\_\_unsigned int’ etc

I’ll try including ‘/include/linux/types.h’ or else ‘/include/uapi/linux/types.h’ by doing

#include<linux/types.h> or #include<uapi/linux/types.h>

Just noticed that in the file ‘/include/uapi/linux/sched/types.h’, it already includes ‘/include/linux/types.h’ by doing an ‘#include<linux/types.h’ so ig I should try using ‘\_\_u32’ for policy and ‘\_\_s32’ for nice value because that’s how it is there in the ‘/include/uapi/linux/sched/types.h’.

**Side note**:

* (Pick\_next\_task() calls either of ‘\_\_pick\_next\_task()’, ‘pick\_task()’.)
* I read the ‘/Documentation/admin-guide/README’ and it had information about ‘make menuconfig’ and ‘make modules\_install install’ and it showed how to make them into a different directory as well. ‘make 0=/home/name/build/kernel menuconfig’, ‘make 0=/home/name/build/kernel modules\_install install’ were there in the README.
* (Now, the function called pick\_task(rq) but I couldn’t find it’s definition anywhere and I didn’t really know what it did. So I ‘cscope -d’ and searched ‘pick\_task(‘ and realised that it’s a pointer to a function(that accepts only one parameter of ‘struct rq’ and returns a pointer to a ‘struct task\_struct’ variable). )
* ‘/kernel/sched/sched.h’ had ‘idle\_sched\_class’ to be declared as ‘extern const struct sched\_class idle\_sched\_class;’ and so ig it’s just an empty thing. I noticed that there is a ‘DEFINE\_SCHED\_CLASS(name)’ macro which expanded to a ‘const struct sched\_class name\_##sched\_class’ and in the ‘/kernel/sched/idle.c’ file, this macro is used after having it’s value defined, the pick\_task points to pick\_task\_idle function which is again defined in that same file. Similarily, there is fair.c file with it’s own ‘DEFINE\_SCHED\_CLASS(fair)’ statement and it’s pick\_task\_fair() function.

Line 811 in the file shows that ‘policy’ is an ‘unsigned int’.

Line 1846 in the file shows that ‘task\_nice’ returns an ‘int’ value.

Things I’ve come to realise after going through kernel code:

1. Kernel views processes as ‘tasks’. Task list is probably a doubly linked list of ‘task\_struct’ to iterate over all the processes.

‘Cat /proc/sys/kernel/pid\_max’ to view the maximum number of processes that can exist simultaneously.

1. All processes are descendants of init process whose pid is 1. At line 64 of ‘/init/init\_task.c’ file, there is some code which initialises this process with pid of 1.

//////////////HAD TO DELETE MY KERNEL CODE BECAUSE FOR SOME REASON I CONTINUED TO GET ERRORS SAYING FORBIDDEN MIXED DECLARATIONS DESPITE RE-EXTRACTING THE FILE(TO REPLACE THIS FILE SINCE I HAD MADE CHANGES TO IT BY DOWNLOADING THE ORIGINAL FILE FROM THE ZIP FOLDER) THAT GAVE ME THE WARNING BUT IT STILL DIDN’T WORK AND GAVE ERRORS FOR EVEN THAT.

HOW I UNINSTALLED EVERYTHING THAT I DOWNLOADED ON LINUX:

1. Go to the directory where you downloaded your linux source code and delete it using ‘rm -rf linux\_folder\_name’, I also removed the tar zip folder ‘rm linux.tar.xz’
2. Since we did ‘make install’, we need to remove the installed files:
   1. ‘ls /boot | grep vmlinuz’
   2. ‘sudo rm /boot/names\_of\_all\_files\_that\_indicate\_the\_version\_you\_downloaded’
   3. ‘sudo rm -rf /lib/modules/5.15.160’ (Removing the module, since I downloaded 5.15.160 version, I put this.)
3. Updating the Bootloader, ‘sudo update-grub’
4. Sudo reboot.

**Reattempt at changing scheduler code:**

**Note: The warning/error that comes saying ‘forbidden declaration in C90’ is probably because C90 standard requires you to declare all the variables before any executable instruction regardless of whether that variable is being used at a much later point.**

1. **Added ‘#include<linux/kernel.h>’ in ‘kernel/sched/core.c’ and then did printk() statements while entering the pick\_next\_task() functions and others. I rebooted into the kernel and did ‘sudo dmesg’ and found out that the ‘pick\_next\_task()’, definition in the #ifdef CONFIG\_SCHED\_CORE part, was indeed being called but after about 2 seconds of ‘sudo dmesg’ having ended, my computer was stuck. This was probably because when I did ‘fakeroot make’, I got a few warnings saying forbidden declarations for the static keyword (I used them to ensure that the function will do the printk() only once) and the output did look like it was considering it as a normal int and hence was printing all the time instead of just once.**

**So, now I have to somehow figure out a way of declaring a static variable.**

**Not sure why, but the following article:** [**https://stackoverflow.com/questions/7187870/declare-variable-as-locally-as-possible**](https://stackoverflow.com/questions/7187870/declare-variable-as-locally-as-possible) **says that declaring all variables before and only then execute instructions so I did the same and surprisingly enough, it worked.**

**I also added printk() statements in all functions upto and including the functions in ‘A’, ‘B’ (I defined these as points in my notebook for where multiple functions made a call to a function I was inspecting).**

1. **I booted into this version, ran ‘sudo dmesg’ but couldn’t see a single printk statement for some reason. I thought maybe it’s not working because I tried to print the value of ‘tmp’ as well so I decided to remove them but while doing so, I realised that in ‘arch/arm64/kernel/entry-common.c’, inside the function ‘static void \_\_sched arm64\_preempt\_schedule\_irq()’ I had actually made a mistake by doing ‘printk(KERN\_INFO “Entering arm64\_preempt\_schedule\_irq, line\_num, tmp\n”,tmp);’ which should give an error as I hadn’t mentioned the data specifier ‘%d’ and instead just wrote tmp. However, this was not caught during ‘fakeroot make’ so not sure what to make of it.**
2. **I declared ‘static int tmp=0;’ outside the functions which would make this global to the file ‘kernel/sched/core.c’, but this still didn’t print the printk messages. So, I decided to remove the ‘if’ statement and just do printk() everytime the call is invoked. This led to the computer being stuck (probably because it did infinite printk statements and is not able to load only).**
3. **Now, I’m going to try to put the if statement as ‘if(tmp\_nandu<=20)’ then printk().**
4. **Logged in on ‘25th June’ and did ‘sudo dmesg’ and it was printing the printk() continuously of the ‘5765 line, I was using ‘static int’ variables that were declared outside the function scope. Going to remove it.**
5. **Had to redownload the kernel code again, probably because of the continuous printing of the printk(). Running fakeroot gives the ‘unreacheable instruction’ warning. Not sure why.**

IT IS IMPORTANT TO NOTE THAT WHENEVER YOU SEE AN INCLUDE HEADER FILE, YOU CAN NAVIGATE TO THAT FILE BY GOING INTO THE ‘/include’ directory.

Ex: #include<linux/workqueue.h> 🡺The file would be ‘/include/linux/workqueue.h’

BOTTOM HALF HANDLING – understanding how asynchronous signals and interrupts work.

Make ur own Module.

NOTE: I’ve made a change to my ‘history’ command inorder to see the timestamps of the commands executed on the terminal. Follow the below steps to be able to see the timestamps of the ‘history’ command.

1. Verify if ‘HISTTIMEFORMAT’ environment variable is set to display the timestamps or not by doing

echo $HISTTIMEFORMAT

If the above command returns nothing then we will need to set it. If it returns ‘%F %T’ then doing history should ideally give the timestamps as well.

1. Now, we will set the ‘HISTTIMEFORMAT’ in the shell configuration

nano ~/.bashrc

export HISTTIMEFORMAT=”%F %T “ 🡺Add this line at the end of the file you’ve just opened using nano.

Source ~/.bashrc 🡺This will apply the changes.

1. Verify if the environment variable has been set by repeating the command ‘echo $HISTTIMEFORMAT’ which should now give output ‘%F %T’.
2. Entering ‘history’ will now give the timestamps.

I also noticed how fast ‘fakeroot make’ was now working for me. Do ‘history’ and check the timestamps for the commands given the numbers ‘1398’ and ‘1399’, It took less than 20mins to do it.

Making my own module:

Reference:

1. <https://www.baeldung.com/linux/kernel-drivers-modules-difference>
2. <https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html>
3. <https://docs.fedoraproject.org/en-US/fedora/latest/system-administrators-guide/kernel-module-driver-configuration/Working_with_Kernel_Modules/>
4. <https://www.youtube.com/watch?v=SOo1rbnryeo>

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.

1. File name: ‘/include/linux/init.h’:

Should use ‘\_\_init’ immediately before the function name (this is present in the file)

Ex: static void \_\_init initme(int x,int y)

{

Extern int z; z=x\*y;

}

Should use ‘\_\_initdata’ or ‘\_\_initconst’ between variable name and equal sign followed by value;

Ex: static int init\_variable \_\_initdata=0;

Static const char linux\_logo[] \_\_initconst = {0x32,0x36, … };

Noticed that doing ‘sudo cat /var/log/syslog’ start printing the printk() statements that I had put for checking the flow of functions in the scheduler.

Creating a Module to print ‘hello world’ on initialization and ‘goodbye world’ on exiting:

mkdir my\_module\_VN

cd my\_module\_VN

vim my\_mod\_VN\_hello.c

Entered the following inside the .c file:

‘

#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);

‘

Created a makefile, for building and cleaning the linux kernel module we’ve created, in the same directory with the following content:

‘

obj-m += my\_mod\_VN\_hello.o 🡺’obj-m’ variable is used to indicate a kernel module object file. This line specifies the target object file(‘my\_mod\_VN\_hello.o’) that should be build as a module. 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: 🡺Target to build the kernel module.

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules 🡺Command to perform the build is the make command. The ‘/lib/modules/$(shell uname -r)/build’ specifies the directory where the kernel build system resides. This directory contains the necessary Makefiles and configuration files for building kernel modules. ‘M=$(PWD)’ tells the ‘make’ to look for the module source code ‘my\_mod\_VN\_hello.c’ in the current directory and that is why we should change into this directory, You could instead provide a hardcoded path to the directory and run the make from any other directory. ‘modules’ is the target specified in the kernel’s Makefile that build all specified modules(‘obj-m’ targets) listed in the main Makefile.

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean 🡺 ‘clean’ is the target in the kernel's Makefile that removes all generated files from previous builds, including object files (\*.o), module files (\*.ko), and temporary build artifacts.

‘

Now, I run the ‘make’ command. This creates the module with the ‘.ko’ extension.

Now, you can load this module onto the running kernel by doing

‘sudo insmod my\_mod\_VN\_hello.ko’

You can check that the printk() statements in the .c file would be executed, the ones related to the init() function. You can check this by doing a ‘sudo dmesg’

To unload/remove the module, you can do

‘sudo rmmod my\_mod\_VN\_hello’

And you can notice the printk() statement would get executed, the one related to the exit() function.

You can also view the modules using ‘lsmod’ and ‘lsmod | grep my\_mod\_VN\_hello’ to view your module.

**I have also tried the following approach of creating a Makefile and a Kbuild file to load/unload the module.**

The module directory, along with the .c file is the same as above.

The makefile is as follows:

‘

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

kbuild:

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

clean:

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

‘

The Kbuild file is as follows:

‘

EXTRA\_CFLAGS = -Wall -g

obj-m = my\_mod\_VN\_hello.o

‘

On running ‘make’, it will create the module file with the ‘.ko’ extension.

Now, ‘sudo insmod my\_mod\_VN\_hello.ko’ will load the module onto the kernel and the msg can be viewed by ‘sudo dmesg’. The ‘ko’ extension stands for ‘kernel object’, contains the machine code and metadata required by linux kernel to load and manage the module dynamically.

To unload the module, do ‘sudo rmmod my\_mod\_VN\_hello’ and the msg can be viewed by the ‘sudo dmesg’.

It is important to note that in the first approach I also got a msg:

“my\_mod\_VN\_hello: loading out-of-tree module taints kernel.”

“my\_mod\_VN\_hello: module verification failed: signature and/or required key missing – tainting kernel”

But the module was loaded and I could also see the printk() statement that I had put in.

In the second approach, I did not see any such message.

**The Goal: To create a module that will check every minute if the current system’s time’s minute is a multiple of 5 and prints a msg if it is.**

->I wanted to use a function ‘do\_gettimeofday’ inorder to get the current system’s time but I couldn’t find any function like that. I noticed 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’.

Now, I could probably get the current time in seconds using ‘ktime\_get\_real\_ts’.

->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].

}

->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.

->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);

}

}

Now, I am able to 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

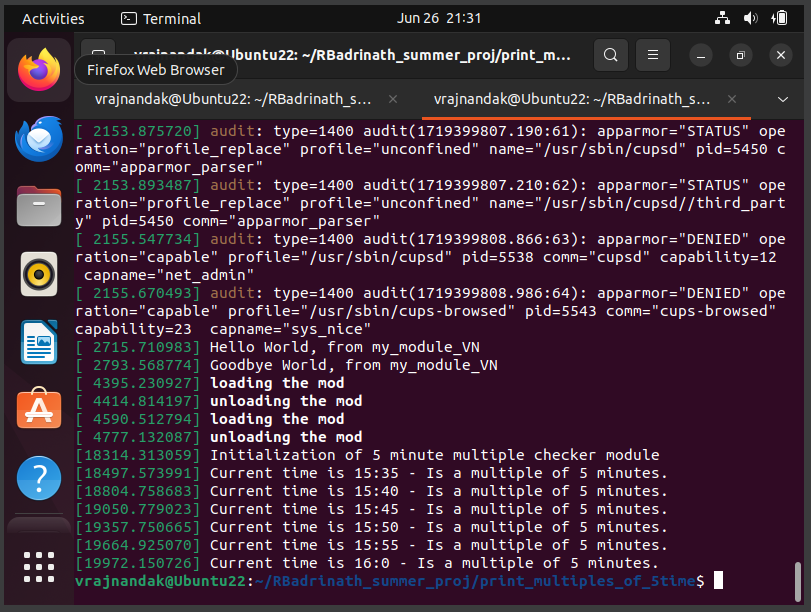
clean:

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

‘

->I now ran the ‘make’ command, followed by ‘sudo insmod time\_checker.ko’ and saw the initialization printk() statement. When my system time was displayed as ’21:05’, I did a ‘sudo dmesg’ and saw the printk() statement but for some reason, the time in the printk() was displayed as ’15:35’ and not ’21:05’. At ’21:11’, I again did a ‘sudo dmesg’ and saw the printk() statement but the time mentioned was ’15:40’ and not ’21:11’.

So, I think there might be an error in the algorithm to calculate the time from seconds to broken-down time or else, I’m not using the proper entry function somewhere.



It is clear from the above screenshot that the printk() is being executed every 300 seconds (approximately, the difference might be due to scheduling the processes???).

Now, you can unload the module by doing ‘sudo rmmode time\_checker’ and on doing the ‘sudo dmesg’ you would notice the printk() statement of exiting the module.

I also tried to do ‘sudo insmod time\_checker.ko’ 2 times, one after the other but got the message “insmod: ERROR: could not insert module time\_checker.ko: File exists” so there has to be some code that is checking whether the module being inserted has previously been inserted or not and inserts based on this. Similarily, while doing ‘sudo rmmod time\_checker’, it checks whether the module has been loaded or not.

Youtube Reference for creating drivers.

<https://www.youtube.com/watch?v=h7ybJMYyqDQ&list=PLCGpd0Do5-I3b5TtyqeF1UdyD4C-S-dMa&index=3>

Reference to know more about crontab: <https://linuxhandbook.com/crontab/>

TRY TO DO SOMETHING USING ‘kprobe’ or ‘ebpf’ AND FIGURE OUT THE SCHEDULER CODE OR SOMETHING LIKE THAT. PEOPLE ARE BEING ENCOURAGED TO USE ‘ebpf’ THESE DAYS SO LEARNING THIS MIGHT BE USEFUL.

GOALS:  
a. Use timer interrupt mechanism and such and re-implement the above module of the ‘mulitple of 5 minutes checker’.

b. Figure out the scheduler code.

c. Do something with ‘ebpf’ and later ‘kprobe’.

eBPF, or Extended Berkeley Packet Filter, is a powerful technology in the Linux kernel that enables safe and efficient programmability of networking, tracing, and security functions within the kernel. Originally inspired by the Berkeley Packet Filter (BPF), eBPF extends its capabilities significantly, allowing for dynamic and safe execution of custom code snippets directly within the kernel.

Tracing with eBPF allows developers and administrators to inspect and trace the execution flow of both kernel and user-space applications in real-time, without the need for intrusive modifications or recompilation of code.

<https://stackoverflow.com/questions/18858190/whats-in-include-uapi-of-kernel-source-project>

The ‘uapi’(Userspace Application Programming Interface) folder is supposed to contain the user space API of the kenel. Then, upon kernel installation, the uapi include files become the top level ‘/usr/include/linux’ files.

<https://www.qualcomm.com/developer/blog/2024/01/uapi-compatibility-checker-automated-tooling-detect-userspace-breakage-linux-kernel#:~:text=UAPI%20includes%20any%20interface%20between,userspace%2C%20it's%20considered%20a%20UAPI>.

UAPI includes any uitinerface between userspace and the kernel, such as system calls, data structures (used in IOCTLs), module parameters, sysfs files and procfs files. i.e., if it’s something that can change in the kenel and break compatibility with a program running in the userspace, it’s considered a UAPI.

Find . -name ebpf

Went to ‘/Documentation/userspace-api/ebpf’

Saw ‘index.rst’, it says that eBPF programs can be attatched to various kernel subsystems, including networking, tracing and linux security modules (LSM). It said to checkout ‘/Documentation/bpf’.

Also saw ‘syscall.rst’, it said to checkout ‘/include/uapi/linux/bpf.h’. Also, to checkout the manpage

<https://www.kernel.org/doc/html/v5.15/trace/kprobes.html>

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 [1](https://www.kernel.org/doc/html/v5.15/trace/kprobes.html#id2), 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.

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.

Checked the file ‘kernel/kprobes.c’. Below, is the flow of the function calls starting from the ‘register\_krpobes()’ function call:

Tracing the flow from ‘int register\_kprobes(struct kprobe \*\*kps, int num)’ function

Int register\_kprobes(struct kprobe \*\*kps, int num)

Calls ‘register\_kprobe(kps[i])’ for i=0 to num-1 🡺int register\_kprobe(struct kprobe \*p);

Ensure that ‘CONFIG\_KRPOBES’ is set to ‘y’ 🡺scripts/config –state CONFIG\_KPROBES

Similarily, ‘scripts/config –state CONFIG\_MODULES’ and ‘scripts/config –state CONFIG\_MODULE\_UNLOAD’ should be set to ‘y’.

Also, ‘scripts/config –state CONFIG\_KALLSYMS’ and even ‘scripts/config –state CONFIG\_KALLSYMS\_ALL’ should be set to ‘y’.

Inorder to insert a probe in the middle of a function, compile the kernel with debug info i.e., CONFIG\_DEBUG\_INFO should also be set to ‘y’ inorder to be able to use ‘objdump -d -1 vmlinux’ to see the source-to-object code mapping.

NOTE: USAGE OF ‘typedef’ with FUNCTION POINTERS:

Ex: typedef int (\*funcptr)(int,int); 🡺This basically makes it so that using ‘funcptr var\_name’ is like telling that the ‘var\_name’ variable is a variable that holds the address of a function that takes 2 parameters with both being int and also returns an int.

Int add(int num1,int num2)  
{

Return num1+ num2;  
}

Int main()  
{

Funcptr addfptr=add;

Printf(“%d\n”,addfptr(10,40));

Return 0;  
}

Ex: typedef int (\*kprobe\_pre\_handler\_t) (struct kprobe\*, struct pt\_regs \*);

Struct kprobe  
{

/\* some code\*/

kprobe\_pre\_handler\_t pre\_handler;

/\*some code\*/

}

Then the ‘pre\_handler’ attribute can basically hold the address of a function that returns and ‘int’ value and takes the parameters ‘struct kprobe \*’, ‘struct pt\_regs \*’ in the same order. You can call the function by doing ‘struct\_var\_name.pre\_handler();’ where ‘struct\_var\_name’ is a variable of type ‘kprobe’ as it has the attribute ‘pre\_handler’

Ex: typedef int (\*hello\_t) (float, char); 🡺’hello\_t’ is like a datatype for storing a function’s address that only returns an ‘int’ and takes in the parameters ‘float’, ‘char’ in the same order.

Int this\_func(float val1, char val2)

{

Return val2;

}

Struct tmp\_struct

{

Int val1;

Hello\_t func\_name;

Float val2;

}

Int main()

{

float dum\_val = 12.3;

struct tmp\_struct my\_struct;

my\_struct.func\_name=this\_func;

printf(“value of character a: %d and using the function: %d\n”, ‘a’, my\_struct.func\_name(dum\_val,’a’)); 🡺Both will print 97.

return 0;

}

GOAL RIGHT NOW:

1. To create a module that will only track the first 100 functions to the pick\_next\_task’ function. We write the ‘timestamp function\_name(param1,param2) -> return\_value’ into the file named ‘my\_probe\_picknexttask.txt’.
2. A module that uses ‘kprobes’ to dynamically trace the entry and exit points of the first 5000 functions in kernel mode. It writes them into a file along with the return values of the function and the parameters sent. All the timestamps are also traced.

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.

I found a ‘samples’ directory which I think contains some sample code on how some of the kernel functions are called. For example ‘/samples/kprobes/kprobe\_example.c’ has a sample kernel module showing the use of kprobes to dump a stack trace and selected registers when kernel\_clone() is called. The trace data would be visible in ‘/var/log/messages’ and on the console whenever kernel\_clone() is invoked to create a new process. ‘For more information on theory of operation of kprobes, see ‘/Documentation/trace/kprobes.rst’.

<https://devarea.com/linux-kernel-development-kernel-module-parameters/>

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.

In file ‘/include/linux/genl\_magic\_func.h’, pr\_info is defined to be ‘fprintf(stderr, args)’ if not already defined. File ‘/include/linux/printk.h’ also has a definition for pr\_info to be a fprintf() call.

I GOT AN ERROR SAYING THAT ‘insmod: ERROR: could not insert kprobe\_kernelclone’. I CHECKED THE OUTPUT OF ‘lsmod | grep kernelclone’ AND SAW THAT THIS MODULE ALREADY EXISTED. SO I CHANGED THE NAME OF MY .C FILE TO ‘my\_VN\_kprobe\_kernelclone’ AND IT GOT LOADED.

I GOT ANOTHER ERROR SAYING ‘vfs\_write’ WAS UNDEFINED ALTHOUGH HAVING DONE ‘#include<linux/fs.h>’ WHICH HAD THE LINE ‘extern ssize\_t vfs\_write()’ WITH THE PROPER PARAMETER LIST. I HAD ALSO PUT IN THE CORRECT PARAMETERS. SO, I SEARCHED FOR ANOTHER ‘write’ FUNCTION AND FOUND ‘kernel\_write()’ AND USED IT. IT HAS THE SAME PARAMETER LIST.

MINI-GOAL 1 COMPLETED: Finished a module named ‘kprobe\_100\_kernelclone\_functions’ which probes ‘kernel\_clone()’ functions and traces a max of the first 100 invocations to this call. Basic information such as ‘timestamp’, ‘func\_name’(here it would be ‘kernel\_clone’ only), ‘address’ (location of the probe point), ‘ip’(instruction pointer), ‘flags’ into a txt file. We could also write the ‘return value’ but right now, in this module, I’m using a static global counter to keep track of which trace record’s details I am filling and so there might be a race condition if I were to use this counter in the post handler as some other processe’s probe’s post handler might have incremented the counter after interrupting this current probe. Thus, it might lead to filling the wrong record. NOT SURE ABOUT HOW TO SOLVE THIS. Now that I think about it there might be a race condition even in the pre handler code since another process might interrupt one and increment the counter so ig we would have to get a lock or something.

CODE:

‘

#include<linux/kernel.h>

#include<linux/module.h>

#include<linux/kprobes.h>

#include<linux/kallsyms.h> //for macro ‘KSYM\_NAME\_LEN’, used for length of function name I think. 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 ‘kernel\_clone’, change by modifying this .C file’s value for that variable. The information is written into the file only once the module has been unloaded.”);

MODULE\_AUTHOR(“Vrajnandak Nangunoori”);

MODULE\_LICENSE(“GPL”);

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

#define MY\_MAX\_FUNCTIONS\_TRACED 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]=”kernel\_clone”; //Function name to be probed.

static struct kprobe kp; //This struct is used for probing.

static struct file \*my\_kernelclone\_trace\_file; //File in which we will be writing the information as described in ‘MODULE\_DESCRIPTOIN’.

static struct my\_trace\_record mytrace\_records[MY\_MAX\_FUNCTIONS\_TRACED];

static int my\_trace\_counter;

//This function will be called just before the probed instruction is executed. The reason for this parameter list is that the default pre\_handler has this parameter list and we are simply overwriting the function which will be called. Hence, following this parameter list is a must.

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=&mytrace\_records[my\_trace\_counter];

if(my\_trace\_counter< MY\_MAX\_FUNCTIONS\_TRACED)

{

printk(KERN\_INFO “In the pre handler of my\_kprobe\_100\n”);

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 the 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 my\_kprobe\_100\_kernelclone\n”);

}

else

{

printk(KERN\_INFO “Finished tracing 100 kernelclone functions, can’t trace more\n”);

//Can also unregister the kprobe here for optimization since you won’t have to needlessly call the pre-handlers and post-handlers. In my opinion, it definitely is better to unregister the kprobe over here.

}

return 0;

}

//This below function is called everytime after kernel\_clone() is done executing. The reason for this parameter list is the same reason as specified for the above pre handler.

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

{

pid\_t my\_var\_pid;

my\_var\_pid=current->pid; //This statement can be executed only if you ‘#include<linux/sched.h>’ which has the ‘current’ , a task\_struct pointer, representing the currently running task.

printk(KERN\_INFO “In posthandler of my\_kprobe\_100\n”);

}

static int \_\_init my\_kprobe\_init(void)

{

int ret;

my\_trace\_counter=0;

printk(“Entered the initialization function of my\_VN\_kernelclone module\n”);

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

if(IS\_ERR(my\_kernelclone\_trace\_file))  
 {

printk(KERN\_ALERT “Failed to open file: %s\n”, MY\_KERNELCLONE\_TRACE\_FILENAME);

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

}

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

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

kp.symbol\_name=symbol;

ret=register\_kprobe(&kp);

if(ret<0)

{

pr\_err(“register\_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;

write\_error\_occured=0;

unregister\_kprobe(&kp);

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

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\_KERNELCLONE\_TRACE\_FILENAME);

}

module\_init(my\_kprobe\_init);

module\_exit(my\_kprobe\_exit);

‘

The makefile is the same as usual. Use ‘strings’ to use grep on the txt file as the txt file is a binary file.

GOAL 1 completed: The same module as the above ‘kprobe\_100\_kernelclone\_functions’ but this module will kprobe any one kind of function and trace it’s first 100 invocations only. The idea is to make the variable holding the function name to be probed, in the above code the variable would be the static global variable ‘symbol[KSYM\_NAME\_LEN]’ , a module parameter and thus we can dynamically set the function to be probed. Thus, we won’t have to make the module again and insert it.

we just simply have to add the line “module\_param\_string(symbol,symbol,sizeof(symbol), 0644);” which will make the ‘symbol’ variable that we’ve declared to hold the name of the function as a module parameter and we can pass any valid function name while insert this module.

Using ‘module\_param(symbol,charp,0660)’ gave me erros saying ‘returning ‘char (\*)’ from a function with incompatible return type ‘char \*\*’

ex: sudo insmod my\_VN\_kprobe\_any\_one\_function.ko symbol=vfs\_write

(or) sudo insmod my\_VN\_kprobe\_any\_one\_function.ko symbol=synchroniz\_rcu

I noticed that for some reason, it doesn’t work when u pass ‘pick\_next\_task’ function, but works well with ‘schedule’.

In order to view the .txt file, you can do ‘strings name\_of\_txt\_file.txt | grep pattern\_to\_search\_for | wc -l’ because grep doesn’t work on binary files and the txt is a binary file.

GOAL 2 COMPLETED: A module that uses ‘kprobes’ to dynamically trace the entry and exit points of the first 100 functions in kernel mode. It writes them into a file along with the return values of the function and the parameters sent. All the timestamps are also traced.

Since, the kprobe puts a breakpoint in the function, it becomes impossible to probe a function without knowing the name. Also, I couldn’t find any special wildcard character or syntax that allowed to probe a function regardless of it’s name. I tried to run it by putting ‘NULL’ as the value for ‘symbol’ but it didn’t work and gave an error saying that it couldn’t insert the module due to invalid parameters.

I have decided to use ‘ftrace’ inorder to trace the functions dynamically regardless of whether the names of the functions are known or not.

In file ‘/include/linux/ftrace.h’:

🡺saw that the function ‘static \_\_always\_inline struct pt\_regs \*ftrace\_get\_regs(struct ftrace\_regs \*fregs)’ uses ‘arch\_ftrace\_get\_regs’ but this is a function-like macro that is defined only when the MACRO “CONFIG\_HAVE\_DYNAMIC\_FTRACE\_WITH\_ARGS” is undefined. I checked using ‘scripts/config –state’ to see that this macro is defined. So I think the ‘ftrace\_get\_regs’ function will never be called or will be called only when the ‘fregs’ parameter is NULL.

🡺’rcu’ stands for ‘Read-Copy-Update’ I think. It’s a synchronization technique designed for read-mostly scenarios where reads significantly outnumber writes. It provides a way to allow concurrent reads without locking overhead, while ensuring that data remains consistent during updates.

🡺There is an ‘extern enum ftrace\_tracing\_type\_t ftrace\_tracing\_type’ which according to the comments specifies the current tracing type and is set to a default of FTRACE\_TYPE\_ENTER

enum ftrace\_tracing\_type\_t

{

FTRACE\_TYPE\_ENTER=0, /\*Hook the call of the function \*/

FTRACE\_TYPE\_RETURN, /\*Hook the return of the function \*/

} //A small doubt, does the ‘,’ after FTRACE\_TYPE\_RETURN not cause an error?

🡺 Functions ‘int register\_ftrace\_function(struct ftrace\_ops \*ops);’ and ‘int unregister\_ftrace\_function(struct ftrace\_ops \*ops);’ and there was a comment saying that “the ‘ftrace\_ops’ must be a static and should also be read\_mostly. These functions do modify read\_mostly variable so use them sparely. Never free an ftrace\_op or modify the next pointer after it has been registered. Even after unregistering it, the next pointer may still be used internally.”

scripts/config –state CONFIG\_FUNCTION\_TRACER 🡺should be set to ‘y’. Mine was already set to ‘y’ and hence I would have access to the above specified enum and the register, unregister functions.

🡺I also have the macro “CONFIG\_DYNAMIC\_FTRACE\_WITH\_DIRECT\_CALLS” and hence

have the functions ‘int register\_ftrace\_direct(unsigned long ip, unsigned long addr);’ and ‘int unregister\_ftrace\_direct(unsigned long ip, unsigned long addr);’ and ‘int modify\_ftrace\_direct(unsigned long ip, unsigned long old\_addr, unsigned long new\_add);’ and more.

🡺I also have the macro “CONFIG\_STACK\_TRACER” and hence have ‘extern int stack\_tracer\_enabled’, ‘int stack\_trace\_sysctl(struct ctl\_table \*table, int write, void \*buffer, size\_t \*lenp, loff\_t \*ppos);’ and more. There are also functions ‘static inline void stack\_tracer\_disable(void)’ and ‘static inline void stack\_tracer\_enable(void)’ and according to the comments, the disable function temporarily disables the stack tracer and is used to disable stack tracing during those critical sections where stack tracing can’t be executed( mainly in RCU). The disable function must be called with pre-emption or interrupts disabled and stack\_tracer\_enable() must be called shortly after while pre-emption or interrupts are still disabled. the enable function is used to re-enable the stack tracer and when stack\_tracer\_disable() is called, stack\_tracer\_enable() must be called shortly afterward.

🡺The main things to focus on are perhaps the register, unregister, enable, disable functions and the ‘struct ftrace\_ops’. The member ‘flags’ of this struct can take multiple values like “FTRACE\_OPS\_FL\_SAVE\_REGS”, “FTRACE\_OPS\_FL\_SAVE\_REGS\_DYNAMIC” etc.

->In file ‘/include/linux/kallsyms.h’. In order to get the function’s name that is going to be calling our trace\_handler() function, I needed to get the function’s name. I found a function called ‘const char \*kallsyms\_lookup(unsigned long addr, unsigned long \*symbolisize, unsigned long \*offset, char \*\*modname, char \*namebuf);’ which according to the comments, looks up an address and the ‘modname’ is set to NULL if it’s in the kernel. Inorder to know how this function actually works, I did a ‘cscope -d’ and searched for where the ‘kallsyms\_lookup’ function is called and in file ‘/kernel/debug/kdb/kdb\_support.c’ saw the following piece of code:

“

int kdbnearsym(unsigned long addr, kdb\_symtab\_t \*symtab)

{

int ret=0;

unsigned long symbolsize=0;

unsigned long offset=0;

static char namebuf[KSYM\_NAME\_LEN];

symtab->sym\_name=kallsyms\_lookup(addr, &symbolsize, &offset, (char \*\*)(&symtab->mod\_name), namebuf);

//rest of the code.

}

“

Thus, I could do the same and get the name of the function. I checked the definition of the ‘kallsyms\_lookup’ :

const char \*kallsyms\_lookup(unsigned long addr, unsigned long \*symbolisize, unsigned long \*offset, char \*\*modname, char \*namebuf)

{

return kallsyms\_lookup\_buildid(addr,symbolsize,offset,modname,NULL,namebuf);

}

I checked the definition of this kallsyms\_lookup\_buildid() function and saw that it changed the value of modname to NULL if it wasn’t already in the case that ‘is\_ksym\_addr(addr)’ evaluated to true. So, I guess I’ll be putting ‘NULL’ for modname because I’m assuming that doing this will trace only kernel functions.

MY CURRENT FILE:

‘

#include<linux/init.h>

#include<linux/kernel.h>

#include<linux/module.h>

#include<linux/ftrace.h>

#include<linux/sched.h>

MODULE\_DESCRIPTION(“Using ftrace to print the information of only the first 100 function invoked regardless of whether the names of the function are known or not. I think only those functions will be chosen that have hookpoints.”);

MODULE\_AUTHOR(“Vrajnandak Nangunoori”);

MODULE\_LICENSE(“GPL”);

#define MAX\_FUNCTIONS\_TRACED 100

static struct ftrace\_ops my\_ftrace\_ops;

static int counter;

static void my\_trace\_handler(unsigned long ip, unsigned long parent\_ip, struct ftrace\_ops \*ops, struct ftrace\_regs \*regs)

{

if(counter<MAX\_FUNCTIONS\_TRACED)  
 {

counter+=1;

printk(KERN\_INFO “Entered the handler code from my\_ftrace\_module, function name: %s\n”,current->comm);

}

}

static int \_\_init my\_VN\_ftrace\_module\_init(void)  
{  
 int ret;

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

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

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

if(ret)

{

printk(KERN\_INFO “Failed to register ftrace function tracer module\n”);

return ret;

}

printk(KERN\_INFO “My tracing module initialized\n”);

return 0;  
}

static void \_\_exit my\_VN\_ftrace\_module\_exit(void)  
{

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

printk(KERN\_INFO “My tracking module has exited after unregistering the trace function.\n”);

}

module\_init(my\_VN\_ftrace\_module\_init);

module\_exit(my\_VN\_ftrace\_module\_exit);

I’ve tried to do ‘sudo insmod’ on this but for some reason my entire screen freezes up. Not sure why though.

I tried commenting all the functions except for the register\_trace(), unregister\_trace(), .func=my\_trace\_handler but the screen still freezes.

I added the ‘\_\_read\_mostly’ to the declaration of the ‘my\_trace\_ops’ variable since the ‘/include/linux/ftrace.h’ file said to add this to the struct declaration.

After having this reference: <https://www.kernel.org/doc/html/v5.15/trace/ftrace-uses.html> i think the reason my computer was being frozen was because my callback function ended up being traced and hence calls the same callback leading to infinite recursion. Thus, I added my code in the callback function as shown:

my\_trace\_handler()

{

int bit;

bit =ftrace\_test\_recursion\_trylock(ip,parent\_ip);

if(bit<0)

return;

//my original code in the above code.

ftrace\_test\_recursion\_unlock(bit);

}

But this still didn’t help.

I now deleted all the code inside the callback function by simply not putting a single line of code in it and surprisingly I was able to hover my cursor on the screen with it being visible. So I really think there might be an infinite recursion of some kind going on.

I tried putting the line ‘module\_put(THIS\_MODULE)’ which will decrement the module usage count and hence unload the module but it did not work. The screen froze and I couldn’t see my cursor. The definition is in ‘/kernel/module.c’ file where it does a preempt\_disable() then decrements then preempt\_enable().

I did a preempt\_disable(), module\_put(THIS\_MODULE), and then a preempt\_enable() but the issue persists.

**I ADDED ‘EXPORT\_SYMBOL(kallsyms\_lookup)’ line in the ‘/kernel/kallsyms.c’ file thinking this would work but it still doesn’t work.**

I tried checking out the ‘/samples/ftrace/’ directory and running the files there but I didn’t understand anything.

The reason has to be something related to the declaration of the ‘ftrace\_ops’ variable. The comment in the ‘struct ftrace\_ops’ definition says “Note, ftrace\_ops can be referenced outside of RCU protection, unleass the RCU flag is set. If ftrace\_ops is allocated and not part of kernel core data, the unregistering of it will perform a scheduling on all CPUs to make sure that there are no more users. Depending on the load of the system that may take a bit of time.

Any private data added must also take care not to be freed and if private data is added to a ftrace\_ops that is in core code, the user of the ftrace\_ops must perform a schedule\_on\_each\_cpu() before freeing it.

“

And there was a comment on top of the ‘register\_ftrace\_function’ saying “The ftrace\_ops must be a static and should also be read\_mostly. These functions do modify read\_mostly variables so use them sparely. Never free an ftrace\_op or modify the next pointer after it has been registered. Even after unregistering it, the next pointer may still be used internally.”

Checked out ‘schedule\_on\_each\_cpu’ in ‘kernel/workqueue.c’ file, is defined as ‘extern int schedule\_on\_each\_cpu(work\_func\_t func);’ in the ‘include/linux/workqueue.h’ file. the ‘work\_func’ is defined as ‘typedef void (\*work\_func\_t)(struct work\_struct \*work);’ in this same header file. ‘struct work\_struct’ is defined as :

‘struct work\_struct { 🡺In the ‘/include/linux/workqueue.h’ file itself.

atomic\_long\_t data;

struct list\_head entry;

work\_func\_t func;

struct lockdep\_map lockdep\_map;

};

‘

One thing I don’t understand is that ‘work\_struct’ uses ‘work\_func\_t’ and ‘work\_func\_t’ also uses work\_strct. A bit hard to understand what’s doing what.

I HAVE STOPPED WORKING ON THE FIX.

I WAS SEARCHING FOR “ERROR: modpost: “kallsyms\_lookup\_name” undefined”. THE TABS I HAD OPENED ARE:

<https://github.com/anbox/anbox-modules/issues/84>

<https://github.com/anbox/anbox-modules/issues/67>

<https://github.com/xcellerator/linux_kernel_hacking/issues/3>

<https://github.com/gravit0/changepid/blob/master/module/main.c>

https://docs.kernel.org/RCU/listRCU.html

**Trythe below code later with the usual makefile:**  
#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/ftrace.h>

#include <linux/kallsyms.h>

#include <linux/seq\_file.h>

#define MAX\_FUNCTIONS 100

static int call\_count = 0;

// Define a structure for seq\_file context

struct kallsyms\_context {

unsigned long address;

char \*name;

size\_t name\_size;

};

static int kallsyms\_callback(void \*data, const char \*name, struct module \*mod, unsigned long address)

{

struct kallsyms\_context \*ctx = data;

if (address == ctx->address) {

strlcpy(ctx->name, name, ctx->name\_size);

return 1; // Stop iteration

}

return 0;

}

static void lookup\_symbol\_name(unsigned long addr, char \*name, size\_t name\_size)

{

struct kallsyms\_context ctx = {

.address = addr,

.name = name,

.name\_size = name\_size,

};

kallsyms\_on\_each\_symbol(kallsyms\_callback, &ctx);

}

static void notrace function\_trace\_callback(unsigned long ip, unsigned long parent\_ip,

struct ftrace\_ops \*op, struct ftrace\_regs \*regs)

{

char fname[KSYM\_NAME\_LEN];

if (call\_count < MAX\_FUNCTIONS) {

lookup\_symbol\_name(ip, fname, sizeof(fname));

pr\_info("Function %d: %s\n", call\_count + 1, fname);

call\_count++;

}

if (call\_count >= MAX\_FUNCTIONS) {

unregister\_ftrace\_function(op);

}

}

static struct ftrace\_ops ops \_\_read\_mostly = {

.func = function\_trace\_callback,

.flags = FTRACE\_OPS\_FL\_SAVE\_REGS,

};

static int \_\_init trace\_init(void)

{

int ret;

ret = register\_ftrace\_function(&ops);

if (ret) {

pr\_err("Failed to register ftrace function\n");

return ret;

}

pr\_info("Kernel function tracing initialized\n");

return 0;

}

static void \_\_exit trace\_exit(void)

{

unregister\_ftrace\_function(&ops);

pr\_info("Kernel function tracing stopped\n");

}

module\_init(trace\_init);

module\_exit(trace\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("A simple Linux kernel module to trace first 100 functions");

WRITING A LINUX DRIVER:

<https://www.youtube.com/watch?v=x1Y203vH-Dc&list=PLCGpd0Do5-I3b5TtyqeF1UdyD4C-S-dMa>

Linux MODULE TO TRACE SOME BASIC SCHEDULING EVENTS WITH THE AVAILABLE PROBES:

REFERENCE: <https://www.infradead.org/~mchehab/rst_conversion/trace/tracepoints.html>

REFERENCE: <https://lwn.net/Articles/379903/> (THIS has a good explanation of entry points)

REFERENCE: <https://lwn.net/Articles/381064/> (This is the part2)

REFERENCE: <https://lwn.net/Articles/383362/> (This is the part3)

<https://www.kernel.org/doc/Documentation/trace/tracepoints.txt>

In file, ‘/include/trace/events/sched.h’:

* the ‘TRACE\_SYSTEM’ macro at the start of the file defines what gropu the ‘TRACE\_EVENT’ macros in this file belong to, and is also the directory name that the events will be grouped under in the debugfs tracing/events/ directory. This grouping let’s user to enable or disable events by group.
* I found a lot of usage of macros ‘TRACE\_EVENT’, ‘DECLARE\_EVENT’, ‘DEFINE\_EVENT’, ‘DEFINE\_EVENT\_SCHEDSTAT’, ‘DECLARE\_EVENT\_CLASS’, ‘’ etc. There was also a ‘TRACE\_EVENT(sched\_pi\_setprio,’ thing at line 525, which acc to comments is the tracepoint for showing priority inheritance modifying a tasks priority.
* The file has ‘#include<trace/define\_trace.h>’ at the end of the file. On checking out this file ‘include/trace/define\_trace.h’, the comments say that the trace files that want to automate creation of all tracepoints defined in their file should include this file. The comments also mention the purpose of different macros: TRACE\_SYSTEM(defines the system the tracepoint is for), TRACE\_INCLUDE\_FILE (if file name is something other than TRACE\_SYSTEM.h, then this macro may be defined to tell define\_trace.h what file to include), TRACE\_INCLUDE\_PATH(if path is something other than core kernel include/trace then this macros can define the path to use. The path is relative to ‘define\_trace.h’ and not the file including it. Full path names for out of tree modules must be used.

I found in file ‘/include/linux/tracepoint.h’ the macros being defined and at the end of the file, it showed the usage of the ‘TRACE\_EVENT’ macro for ‘sched\_switch’.

TRACE\_EVENT(name, proto, args, struct, assign, print)

In file ‘/kernel/sched/core.c’ , there is a call to ‘ret = register\_trace\_sched\_switch(ftrace\_graph\_probe\_sched\_switch, NULL);’ so ig I can use this function to replace the first argument with my own callback. The 1st argument was a function:

“”””

static void ftrace\_graph\_probe\_sched\_switch(void \*ignore, bool preempt, struct task\_struct \*prev, struct task\_struct \*next)

{

unsigned long long timestamp;

int index;

if(fgraph\_sleep\_time)

return;

timestamp=trace\_clock\_loca();

prev->ftrace\_timestamp=timestamp;

if(!next->ftrace\_timestamp)

return;

timestamp-=next->ftrace\_timestamp;

for(index=next->curr\_ret\_stack;index >=0; index--)

next->ret\_stack[index].calltime+=timestamp;

}

“”””

In the same file, I also saw a call being made to ‘unregister\_trace\_sched\_switch(ftrace\_graph\_probe\_sched\_switch,NULL);’ so Ig I can use this function to replace the 1st argument with my own callback.