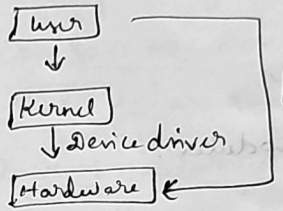
S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
KM :-		→ Writing LKM & insertion, removal		
		→ understand why Makefile & others		
		→ .c to .ko full understanding.		
		→ passing arguments to KM, as well as to builtin KM		
		→ Module stacking		
		→ Kernel tainting path		
		→ use case of src version [modinfo o/p]		
		→ strace		
		→ Kernel panic		
		→ memory map of process)		
		→ Kthread		
		→ Load modules automatically		
		→ Kernel module blacklist		

① Device driver :-

→ It is a piece of software that controls a particular type of device which is connected to the Computer system.

3 sides :- → ① one side talks to rest of Kernel.
② one talks to the hardware
③ one talks to user.



[also access directly without device driver is using "device file"

② Kernel module :-

Traditional way of adding code to Kernel was to recompile the Kernel and reboot the system.

→ Kernel modules are piece of code that can be loaded/inserted and unloaded/removed from the Kernel as per demand/need.

Other names of Kernel modules :-

1. Loadable Kernel module (LKM) 2. Modules

Extension :- .ko (Kernel object)

Standard location :-

Modules are installed in /lib/modules/<Kernel version> directory of rootfs by default.

cd /lib/modules

5.1.-generic 5.1.generic] => Kernel modules Installed.

To check the Current Kernel

$ uname -r

o/p: 5.4.0.41-generic

then cd 5.4.-generic

then cd Kernel

then if we want to Search all ko modules,

$ find . -name '*.ko' => shows all modules.

(space space space)

Count:-

$ find . -name '*.ko' | wc -l

o/p 5505

③ Kernel modules vs Device drivers:-

-> A Kernel module may not be a device driver at all.

-> A driver is like a Sub-class of module.

-> A driver is always a K.M.

modules are used for below:-

1. I.D	1. Network drivers: Drivers Implementing a network protocol (TCP/IP)
2. file system	5. TTY line disciplines: for terminal devices
3. system calls	

Flow:- -> always in ram.

First (Base Kernel) is loaded (booted), after it is booted K.M are loaded. K.M may be in separate memory.

Advantages of K.M:-

1. All parts of Base Kernel stay loaded all the time. Modules can save you memory, because you have to have them only when you are actually using them.
2. Users would need to rebuild and reboot the Kernel every time they would require a new functionality.
3. A bug in driver which is compiled as part of Kernel will stop system from loading, whereas module allows systems to load.
4. Faster to maintain & debug multiple
5. Makes it Easier to maintain machines on single Kernel base.

Disadvantages:-

1. Size: Module management Consumes unpageable Kernel memory. A Basic Kernel with no. of modules loaded will Consume more memory than an Equivalent Kernel with drivers compiled into Kernel image itself.

This can be very significant issues on machines with limited physical memory.

2. As the Kernel modules are Loaded very late in the boot process, hence core functionality has to go in base Kernel (Eg. Memory Management)

3. Security: If you build your Kernel statically and disable linux's dynamic module loading feature, you prevent run-time modification of Kernel Code.

Configuration:

In order to support modules, Kernel must have been built with following options enabled:

CONFIG_MODULES=y

To check that:-

$ cd /boot -> we can check by ls -> it shows lot:

$ cat config-[redacted] 'uname -r'

$ Cat Config-'uname -r' | grep CONFIG_MODULES

CONFIG_MODULES_USE_ELF_RELA=y
CONFIG_MODULES=y
CONFIG_MODULES_TREE_LOOKUP=y

output

[press button down key]

very important

④ Types of Modules:-

1. In-Source Tree:- Modules present in the linux Kernel Source Code. [linux source code -> kernel.org]

2. Out-of-Tree:- Modules not present in linux Kernel source code.

All modules start out as "out-of-tree" developments, that can be compiled using context of source-tree. Then went to review. once a module gets accepted to be included, it becomes an in-tree module.

⑤ Basic Commands:-

1. List modules:-

(lsmod) lsmod gets its information by reading the file /sys/module currently

2.6

/proc/modules previously

$ cd /boot

$ lsmod

↓
It shows all modules

$ lsmod | tail ⇒ shows latest installed module

o/p:

Module	size	used by	
ccm	20480	9	(1x) dependency
intel_rapl_msr	20482	0	
intel_rapl_common	24576	1	intel_rapl_msr

2. Module Information :-

(modinfo) : prints the information of the module

$ cd boot

$ modinfo modulename → Modulename is replaced while typing. It can be identified (or) Chosen from lsmod (previous Command)

↓
It shows Info.

Eg:-

$ modinfo psmouse | less

↓
It shows Info. [please check it in terminal.
One Intresting, we can find whether it is In-tree (or) out of tree:
"intree: Y" → will appear there]

⑥ Hello world Kernel module:-

In C/C++ programming, we have main() as entry point & exit point.

Kernel modules must have atleast two functions:

→ ① a "start" (Initialization) function: which is called when the module is loaded into kernel.

→ ② an "end" (cleanup) function called: which is called just before it is removed.

This is done with module_init() & module_exit() macros.

Licensing: Module should specify which license you are using MODULE_LICENSE() macro

Eg "GPL" [GNU public License V2 (or) later]

Header files:-

Every kernel module needs to include linux/module.h for macro expansion of module_init & module_exit linux/kernel.h only for macro expansion for the printk() log level.

steps in code:

① Include header file

② Include License

③ Include Initialization & exit fun-

$ vi world.c

```
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");
static int world_init(void)
{
  printk(KERN_INFO"%s: world_init\n", __fun__);
  return 0;
}
static void world_exit(void)
{
  printk(KERN_INFO"%s: world_exit\n", __fun__);
}
module_init(world_init);
module_exit(world_exit);
```

① Code is completed.

② Create Makefile.

Why Makefile?

In Case of C programming, we simply use

gcc -world.c -o a.out

But if we use above, it will not work. Because we have Linux header files so

↓

They are not userspace header files but Kernel header files.

In order to build Kernel source / module, we need to use Kernel make file which is in path

/lib/modules/`uname -r`/build/Makefile

↓

Current Kernel which we are going to build version

Then we can't directly use Kernel makefile. Because we have to include our object in that file (i.e) world.o

So, we are creating a Makefile in local path & ask Kernel to check this created Makefile.

$ vi Makefile → Caps 'M'

```
obj-m := world.o
```

Two options :-

① -m ⇒ It tells we are building modules

② -y ⇒ It builds Kernel source code

③ To Build Module :-

```
$make -C /lib/modules/`uname -r`/build M=${PWD} modules
```

(-C: caps C; PWD: caps PWD)

Check in local folder ⇒ It creates .ko file & others (i.e) world.ko

To check whether module is inserted (or) not,

$ lsmod | less

↓

Shows the module list where we can see world.ko

But it will not be available - Because we are not Inserted into Kernel. Let's do

④ Inserting /ko:

$ sudo insmod ./Hello.ko

⑤ Now check lsmod | less ; we can see that.

⑥ And also in /sys/module/(world) ⇒ folder is also created.

⑦ $ dmesg (or $ sudo dmesg

where we can see printf statements module-entry will be there.

⑧ Remove the module:

$ sudo rmmod ./Hello.ko

⑨ Clean the module [clean the files, folders related to Ko]:

$ make -C /lib/modules/`uname -r`/build M=$PWD clean.

⑦ printf vs printk:

printf() is a function in C standard library.

printk() is a Kernel level function.

printk() is called with one more argument than printf()

like below

printk (KERN_log_priority "hello world\n");

Here log priority is one of eight values (predefined in linux/kernel.h, similar to /usr/include/sys/syslog.h)

EMERG
ALERT,
CRIT,
ERR,
WARNING,
NOTICE,
INFO,
DEBUG

priority decreasing in this order.

⑦⑧ printk() writes to the kernel buffer, whereas printf() writes on the standard output.

⑨ Simplified Makefile:

$ man make

make [options] [target]

Eg: -C

Instead of giving whole command make -C /lib/module ....., we can simply give

① $ make

② $ make clean

Let's do

$ vi Makefile

```
obj-m := Wald.o

all: -> target
	make -C /lib/modules/`uname -r`/build M=$(PWD) modules

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

Eg:

$ make -> in current folder [which Contains makefile]

That's all. It will build modules

$ make clean -> To clean the modules.

① What happens when we do insmod on module?

What is KM?

KM is a piece of Kernel Code which Can be added (insmod) to the running Kernel when loaded and Can be removed (rmmod) from the Kernel when the functionality is removed.

When we do insmod on a module, it performs a series of steps:

a) It Calls init_module() to intimate the Kernel that a module is attempted to be loaded & transfers the Control to Kernel.

Kernel is running in super privileged users, so only way to access the Kernel by user space program is System Calls.

b) In Kernel, sys_init_module() is run. It does a sequence of operations as follows:

→ Verifies if the user who attempts to load the module has the permission to do so (or) not

→ After the verification, load module fn is Called.

→ The load module function assigns temporary memory and Copies the elf module (our module format) from user space to Kernel memory using Copy_from_user fn.

→ It then checks Sanity of ELF file (verification if it is a proper ELF file)

→ Then based on ELF file interpretation, it generates offset in temporary memory space allocated. This is Called Convenience Variables.

→ user arguments to the module are also Copied

to Kernel memory
→ System resolution is done.
→ load_module fn returns a reference to the Kernel module.
→ Reference to module returned by load_module is added to doubly linked list that has a list of all modules loaded in the system.

⑩ what happens if we return '-1' from the Kernel module?

In user space:

```
int main()
{
    return 0;
}
```

→ this means success value as 0 returning negative failure

But in Kernel space?

```
static int test_init(void)
{
    printk(KERN_INFO "1.S: Initialization \n", __func__);
    return -1;
}
```

Trying to build it using makefile.
Trying to insert module ---> error "permission denied".

But check Kernel buffer by "dmesg", there is printf statement "Initialization". But after it errors.

⑪ Giving another name to module :-

Eg: I have hello.c that contain all of my source code.
But I want module to be linux.ko.

$ vi Makefile

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

⋮

⇒ modified to

$ vi Makefile

```
obj-m := linux-o
linux-objs := hello.o
```

⋮

(;e) for linux.o, it needs some object (i.e hello.o
so first it generates hello.o → then linux.o → linux.ko.

⑫ Kernel Module spread across Multiple c files :-

Eg: wor.c

```
#include <linux/kernel.h>
...
void func(void);
static int init(void)
{
    print...,
    func();
}
```

func.c

```
#include <linux/kernel.h>
void func(void)
{
    printk(KERN_INFO, "Hello Linux");
}
```

modification in Makefile: (file wr.c)

obj-m := world.o → another name that we gave in previous

world-objs := wr.o func.o

Important Note:- (from Hands-on)

Before cleaning the module, must remove module from Kernel.

(13) Two Kernel modules from single Makefile

modification in Makefile:-

obj-m := hello1.o

obj-m+= hello2.o

all:

clear...

Note:- We can't insert two modules at a time

(14) What does dmesg Command do?

→ Kernel keeps all the logs in ring buffer.

→ This is done to avoid the boot logs getting lost until the syslog daemon starts and collects them and stores them in /var/log/dmesg.

We will lose the boot up logs if we don't store them in ring buffer.

dmesg Command is used to control (or) print Kernel ring buffer.

Default is to prints messages from the Kernel ring buffer on to Console.

$cd /var/log

$dmesg (or) $sudo dmesg

Important dmesg Commands:-

1. Clear Ring buffer:-

$dmesg -c → will clear the ring buffer after printing.

$dmesg -C → will clear the ring buffer but does not prints on the Console.

2. Don't print Timestamps:-

$dmesg -t → will not print timestamps

previously $dmesg

o/p [51432.781953] test_hello2_exit: In exit

after dmesg -t

o/p test_hello2_exit: In exit

⑤ Restrict dmesg Commands to List of levels :-

Eg: $ dmesg -l err

o/p: [2.93894] cpufreq_reg: failed to initialize policy for cpu

Eg: $ dmesg -l err, warn

o/p: "

[5.47667] systemd-journald[468]: File /var/log/journal/ System (or) uncleanly shut down.

④ print human readable timestamps:-

$ dmesg -T will print timestamps in readable format.

Note: Timestamp could be inaccurate.

⑤ Display the log level in the output:-

$ dmesg -x will add loglevel to the o/p

o/p: kern : info : [5143.2.781953] test_hello2_exit: exit

↓
log level

⑥ We can combine options, to dmesg -Tx will print both human readable time & log level.

⑮ Dmesg follow option:-

previously what are we doing? load the module → print the kernel msg (using dmesg) & unload the module → print the kernel msg.

Now, no need to give dmesg after ins & rmmod. now, we are going to run dmesg in background.

$ dmesg -W &

o/p: (log comes...)

$ jobs ——→ To check the background running.

o/p: [1]+ Running dmesg -W &

$ make

...

$ sudo insmod ./hello.ko

[22567.9912] test_hello_init: In init → immediately prints without giving dmesg.

⑯ Without exit function?

Same code without exit fn.

→ Tried, Code is compiled

→ Insmod → works → test_init → printed

→ rmmod work?

$ sudo rmmod hello

rmmod: Error ../../C793: Kmod - remove - module() Could not remove 'hello': Device (or) resource busy

rmmod: Error: Could not remove module hello: Device Source busy.

Why do we get -EBUSY when we remove the module?

Definition: Kernel/module.c

→ There is a check in implementation to strictly have the exit function if there is a init function. If it failed to find exit function. It throws -EBUSY

```
/* ... */
if (mod->init && !mod->exit)
{
    forced = try_force_unload(flags);
    if (!forced) {
        ret = -EBUSY;
        goto out;
    }
}
```

delete_module is the system call which implements the removing/unloading the module.

$ cd linux-5.2.8/

$ vi Kernel/module.c

:

:

/SYSCALL_DEFINE → search

→ It shows

( )

17. LKM with exit only?

→ Same code without init

→ tried → builded

→ Insmod → No error.

open the Kernel/module.c file

/sys call define Init_module → load_module → do_init_module

↓

Here if init function is not there, it does not return error.

18. Two line Linux LKM:

Src Code Contain

#include <linux/module.h>

MODULE_LICENSE ("GPL");

tried → Compiled → Ko file generated

Insmod → Works

Rmmod → Works

(18) overviewing of Compiling Kernel modules

- From .c to .ko :

*Kernel use Kbuild system to build the Kernel modules.

*Kbuild system reads the assignment of "obj-m := modulename.o" from makefile.

→ Now Kbuild system know that it has to build "modulename.ko" and will look for "modulename.c" for the source.

→ In case these files are not present in directory. it passed to "M", the Compiling will stop with an error.

→ If files are present in the source file is Compiled "modulename.o" and "modulename.mod.c" is created which is Compiled to "modulename.mod.o".

→ modulename.mod.c is a file that basically Contains the information about the module (version Information etc.)

→ The modulename.o and modulename.mod.o are linked together by modpost in the next stage to create "modulename.ko".

other files :- (Contain exported symbols in our modules)

"module.symvers": This will Contain any of external symbols that is defined in your module and hence not present in the module.symvers of Kernel.

(22) understanding Module.symvers and modules.order :

modules.order : In Case you are Compiling multiple modules together, it will Lost out the order in which Compilation and Creation of .ko takes.

(23) Insmod vs Modprobe

lsmod → List modules that loaded already

insmod → Insert Module into Kernel

modinfo → Display Module Info

rmmod → Remove module from Kernel

modprobe → Add (or) Remove modules from Kernel.

Difference b/n insmod vs modprobe

insmod :- Loads the module given 'insmod /path/to/module.ko' → (or) 'insmod ./v.ko

modprobe : Loads the module only in /lib/modules/$(uname -r)

note: 'modprobe /home/test/hello.ko' will not work.

insmod :- Dependencies if present are not loaded.

modprobe :- modprobe Calculates dependencies, loads the dependencies and then the main module.

(23) How modprobe Calculates dependencies?

Modprobe depends on depmod tool to Calculate dependencies.
depmod Calculates dependencies of all modules present in
/lib/modules/$(uname -r) folder, & places the dependency
information in /lib/modules/$(uname -r)/modules.dep file.
ex: $ man depmod → check in terminal.

Eg:
Kernel/drivers/.. adm8211.ko Kernel/.. mac80211 kernel/drivers/.. eeprom93.ko

when you say modprobe adm8211.ko, eeprom.ko, cfg80211.ko
is loaded first and then adm8211.ko.

(24) Modules are loaded right to left & removed left to right.

(25) understanding module_init & module_exit fn

We know on insmod, the function passed in module_init
macro is called, and on rmmod, argument passed in the
module_exit called.

Let's see definition of this macro, it is present in linux/module.h

```
#define module_init(initfn) \
static inline initcall_t __inittest(void)
{ return initfn; }
int init_module(void) __attribute__((alias(#initfn)));

#define module_exit(exitfn) \
static inline exitcall_t __exittest(void)
{ return exitfn; }
void cleanup_module(void) __attribute__((alias(#exitfn)));
```

stringify operator which converts into string

(*) purpose of defining __inittest function is to check during compile time, the function passed to module_init() macro is compatible with initcall_t type.

initcall_t is defined in linux/init.h:

```
typedef int (*initcall_t)(void);
```

If you declare module_init function which returns void instead of int, the compiler will throw warning.

That's why we pass,

```
static int myfun(void)          (ex) module_init(initmyfun)
{
    ...
    return 0; }
```

(26) Example of Gcc attribute alias :-

alias.c

```
#include <stdio.h>
static int myfunc(int a, int b)
{
printf("%s: Adding %d with %d : It Result: %d\n",
                    __func__, a, b, a+b);
return a+b;
}
static int add(int a, int b) __attribute__((alias("myfunc")));
int main()
{
  add(3,6);
  myfun(3,5);
  return 0;
}
```

↘ just declaring a fn. not defining.
(or) simply giving another name (add) to "myfunc".

Eg 2: alias_variable.c

```
#include <stdio.h>
int oldname=5;
extern int newname __attribute__((alias("oldname")));
int main()
{
  printf("value of new name is : %d\n", newname);
  return 0;
}
```

↘ If we don't use extern, it will become function definition declaration. We want declaration.

Build file

```
$ ls
$ vi alias.c
$ ls
alias.c
$ cc alias.c -o alias
$ ls
alias.c alias
$ ./alias
$ o/p: occurs.
```

(27) Linux Kernel Module example without module_init and module_exit :- check lecture.

(28) How Can I pass Parameters to the Linux Kernel modules :-

In C programs, we pass Command line parameters through argc/argv.

Kernel programming also allows us to pass Command line parameters.

Command line parameters provide a single kernel driver to do multiple things, for example

→ Instead of fixing to single I/O address for read/write, it can provide that as command line argument and allow user to read/write any address.

→ Enable/disable debug logs/printk

→ Allow user to set the mode if driver supports multiple modes.

How to pass parameters to module?

We can add parameters using module_param macro.

Declared in moduleparam.h file;

#define module_param(name, type, perm)
    module_param_named(name, type, perm)

name: name of variable

type: Type of variable. Supported types are charp (character pointer), bool, invbool, int, long, short, uint, ulong, ushort

perm: permission for the sysfs entry.

Eg: S_IRUGO: only read by all users

0: No sysfs Entry.

You can also use numeric values like 0644 for permission Entry.

Example: watch video.

Eg:

```
#include ...
int basic=0;
int hra=0;
int allow=0;

module_param(basic, int, S_IRUGO);
module_param(hra, int, S_IRUGO);
module_param(allow, int, S_IRUGO);

static int Salary_init(void)
{
  printk(KERN..., "Take home = %d", basic+hra+allow);
  return 0;
}
static void salary_exit(void)
{
  printk(KERN..., "exit");
}
module_init(salary_init);
module_exit(salary_exit);
```

Passing Parameters:

$ Sudo insmod ./arg.c basic=18000 hra=3000 allow=1000

o/p: Take home = 22000 ⇐ [Note previously we have to do $ sudo dmesg -wH] see this as soon as.

29. What happens if we pass incorrect values to module Parameters?

$: Sudo insmod ./arg.ko basic=0.005

o/p: insmod: Error: Could not insert module ./arg.ko: Invalid parameters (compile in note)

we can also see error in dmesg.

30. How to pass parameters to builtin modules :-

How to find out values of already loaded modules

$ cat /sys/modules/<module_name>/parameters/<parameter_name>

This is only possible, if the permission field in the module_parameter macro is not zero.

How can we pass arguments to modules which are built in?

Module parameters for built-in modules are passed through Kernel command line.

Syntax: <module_name>.<parameter_name>=value

So if the argument module was built-in, we can append argument. loop_count=5 (~~arg~~ in our example, arg.basic=18000) to the Kernel Command line for passing 18000 as basic.

How can we pass arguments which are called by modprobe?

modprobe reads /etc/modprobe.conf file for parameters.

31. How to pass string with multiple word as parameter?

Assume we have character variable (name) in our module. Let's see

$ Sudo insmod ./arg.ko name="Hello world".

$ dmesg → o/p: Hello
'World' is ignored.

try: $ Sud" " = 'Hello world';

$ dmesg → o/p: Hello
'world' is ignored

try:-

$ sudo ... name='"Hello world"'

dmesg -s 0f: Hello world.

Reason:-

This happens because shell removes double quotes and pass it to insmod, to avoid this add a single quotes over the string.

Run the following command: "insmod argument.ko name='"Linux world"'" to pass the whole string.

32. passing zero to permission argument of module - param macro

Code change in argue:- (lecturer code)

```
module_param(name, charp, 0);   // charactor pointer; passing zero
```

When we go & check,

$ ls /sys/module/arguments/parameters ~~loop_count~~

shows only one parameter. (by) loopcount

But "name" is not present because we pass zero to permission argument

But when we check modinfo, [because it can access]

$ modinfo ./arguments.ko

```
filename: ...
description: ...
:
parm: name:charp        // it shows here but not there
param: loop_count: int
```

(33) passing array as module parameters :-

To pass multiple parameters, we need to pass parameter array.

To pass array we need to use module_param_array() function instead of module_param() function

#define module_param_array (name, type, nump, perm)

Here nump: optional pointer filled in with number written.

$ vi array.c

```
#ind. ...
#include <linux/moduleparam.h>   -> (34)
int array[4];
static int arg_count=0;
```

```
module_param_array(array, int, &arg_count, S_IWUSR|S_IRUSR);

static int arg_test(void)
{
    ...
    int i=0;
    printk("Arg_count: %d\n", arg_count);
    printk("Array elements:\n");
    for(...)
    {
        print array elements
    }
    return 0;
}
```

$sudo insmod ./array.ko array=4

o/p: Arg Count: (1)
Array elements:
0: 4
1: 0
2: 0
3: 0

$" " array=4,5

o/p: Arg Count: (2)
...
0: 4
1: 5
2: 0
3: 0

34. invbool:

→ invbool is one type of parameters supported by module_param.

→ A boolean (true (or) false) value.

The invbool type inverts the value, so that true values become false & vice-versa

(35) symbol and symbol table:-

What is symbol?

A symbol is a name given to a space in the memory which stores

data (variables, for reading & writing)

Instructions (Functions, for executing)

So symbol in programming language is either a variable (or) function.

What is symbol table?

→ Data structure created by Compiler containing all symbols used by program.

→ Every kernel image that you build has a .symbol table with it.

Linux kernel symbol table contains names & addresses of all the kernel symbols.

when you install the Kernel, it will be present in /boot/System.map-<Linux_version>

$ cd /boot/system.map-4.18.0-25-generic

If try to cat, permission denied.

So, $ sudo cat

It will list lot. "kobj_ns_ops_tbl" → This symbol table.

(36) Exporting symbols:-

How to export your symbols?

When you define a new function in your module, the default behavior of this function is local, only the module in which the function is defined can access it, cannot be accessed by other modules.

To export this module, we need to use EXPORT_SYMBOL (or) EXPORT_SYMBOL_GPL.

once you export them, they will be available to other modules to use.

EXPORT_SYMBOL:- The exported symbol can be used by any Kernel module.

EXPORT_SYMBOL_GPL:- The exported symbol can be used by only GPL licensed code.

(37) System.map Vs /proc/kallsyms:-

/proc/kallsyms: Contains symbols of dynamically loaded modules as well as builtin modules.

System.map: Contains symbols of only builtin modules

Let's say we write a module & export it symbol. It will be available in /proc/kallsyms only. Because what our modules are "out of source tree".

38. Linux Kernel module example of exporting function:-

```
#include
#include <linux/jiffies.h>
MODULE_LICENSE ("GPL");
void print_jiffies(void);
{
    printk(KERN_INFO":).s: jiffies:-).ld\n", __func__, jiffies);
}
```

Kernel variable

static int ...
static void ... → for name

EXPORT_SYMBOL_GPL(print_jiffies);

module_init ...
module_exit ...

...

$make
$insmod ...
$sudo cat /boot/System.map-4.18.0-25-generic | grep print_jiffies
→ Caps

No o/p:

$ sudo cat /proc/kallsyms | grep print_jiffies

(39). Module stacking:-

What is it?

new modules using the symbols exported by old modules.

Examples of modules stacking in Linux Kernel?

① Msdos filesystem relies on symbol exported by fat module.

② parallel port printer driver (lp) relies on symbols exported by generic parallel port driver (parport)..

our intention of examples:

1. write module which exports a function myadd performing addition of two numbers:
2. Write another module which uses the exported function

mod1.c

```
#include ...

int myadd(int a, int b)
{
  pr_info("%s : Adding %d with %d\t Result : %d\n", __func__,
          a, b, a+b);
  return a+b;
}
EXPORT_SYMBOL(myadd);

static int module1_init(void)
{
...
}
static void module1_exit(void)
{
...
}
module_init( )
module_exit( ) ..
```

mod2.c

```
#include
extern int myadd(int a, int b);
static int module2_init(void)
{
  printf("%s: Add: %d\n", __func__, myadd(3,5));
  return 0;
}
```

① Create makefile which builds the both mod1 & mod2.c ft.
② Insert mod1
③ sudo cat /proc/kallsyms | grep myadd
④ Insert mod2
⑤ dmesg → result will appear

40. Linux Kernel module example of exporting Variable:

⑥ symbol_export.c :-

```
#include ..
#include "symbol_test.h"
Module_license...
struct test foo;
EXPORT_SYMBOL(foo);
```

variable

```
static int test_export_init(void)
{
  foo.a = 5; foo.b = 6; foo.c = 7. ....
  return 0;
}
Static void test_export_exit(.)
  ...
```

symbol_test.h

```
#ifndef _SYMBOL_H
#define _SYMBOL_H
struct test
 { int a; int b; int c. ....
 }; 
extern struct test foo;
#endif
```

① Insert symbol_export.ko
② sudo cat /proc/kallsyms | grep foo
③ Vi symbol1.c

symboll.c

```
#include "symbol_test.h"
...
--
```

```
static int test_symbol_init(void)
{
printk(KERN_INFO "%s: In init\n a: %d\t c: %c\n",
        __func__, foo.a, foo.c);

return 0;
}
static void. . .
```

Three steps ⊕

4. Sudo insmod ./symbol1.ko

4) Version Magic:-

It is a magic string present in linux kernel and added into the .modinfo section of Linux Kernel Modules.

→ This is used to verify whether Kernel module was/is Compiled for the particular Kernel version (or) not.

'VERMAGIC_STRING' is generated by Kernel Configuration.

#define VERMAGIC_STRING
UTS_RELEASE " "

$ vi vermagic.c

```
#include <linux/vermagic.h>
static int vermagic_init(void)
{
  pr_info("VERMAGIC_STRING = " VERMAGIC_STRING "\n");
  return 0;
}
static void vermagic_exit(void) {}
module_init(...)
```

$ Sudo insmod ./vermagic.ko

o/p: VERMAGIC_STRING = 4.18.0-25-generic SMP mod_unload

② $ uname -a

③ modinfo ./vermagic.ko

o/p: filename:
license: GPL
:
name: vermagic
vermagic: 4.18.0-25-generic SMP mod_unload
:

Eg2: $ vi vermagic2.c

#include <linux/vermagic.h>

static int .. (same fn in vermagic.c)

:

MODULE_INFO(vermagic, "1234567");

↳ additionally one statement in vermagic.c

Here we change vermagic to some "1234567" value.

$ Sudo insmod ./vermagic2.ko

$ sudo modinfo ./vermagic2.ko

filename: ..

vermagic: 1234567 ? How, is it conflict?

:

vermagic: 4.18.0-25-generic SMP mod_unload ;

Let's try to load,

$ insmod ./vermagic2.ko

o/p: Could not insert module.

$ dmesg

o/p: Vermagic 2: version magic '1234567' should be '4.10...generic SMP mod_unload'

42. Module licenses:-

What happens if we don't specify MODULE_LICENSE macro?

→ It will give warning, but it can be loaded successfully.

→ But if we check modinfo for that module, we can't find the license info.

43. What is tainted Kernel?

When the Kernel is tainted, it means that it is in state that is not supported by Kernel Community.

→ In addition, some debugging functionality and API calls may be disabled when Kernel is tainted.

Kernel may become tainted for any of several reasons, including (but not limited to) the following:

① use of proprietary (or non-GPL-compatible) Kernel module - this is the most common cause of tainted Kernels and usually results from loading proprietary NVIDIA & AMD drivers.

② use of staging drivers, which are part of Kernel source code but are not fully tested.

③ use of out-of-tree modules (we are creating) that are not included with Linux Kernel source code.

Certain critical error conditions, such as machine

check exceptions & Kernel oopses.

45. How to check whether Kernel is in tainted state or not?

→ on first time loading of Kernel (eg: it is tainted), it will print "loading out-of-tree module taints Kernel".

→ After that, it will not tell that it is tained.

At runtime, you can Query tainted state by reading

$ Cat /proc/sys/kernel/tainted

If that returns '0', Kernel is not tainted.

any other number indicates the reasons why it is.

Watch lecture → for more info.

46. What happens when you specify Invalid license (say "abc"):

- Almost same as others.

47. What happens when non GPL Kernel module trying to access GPL Module?

module1.c

```
#include...
MODULE_LICENSE("GPL");
int myadd(int a, int b)
{
...
}
```

Export_SYMBOL_GPL(myadd);

module2.c

```
// ...
MODULE_LICENSE("proprietary");
extern int myadd(int a, int b);
static int module2_init(void)
{
p... myadd(2,3);
}
```

→ Gives the warning before building .ko.

(i.e) GPL-incompatible module module2.ko uses GPL-only Symbol 'myadd'.

$ make V=1 (or GPS) → This gives verbose info. (What happens internally in Kernel?)

48. How to find out Kernel version from .ko?

modinfo → gives vermagic. [Where we can find].

$ uname -r ⇒ also gives.

49. Module Metadata:

MODULE_DESCRIPTION can be short synopsis of what your module is trying to accomplish.

MODULE_AUTHOR, declares module's author.

MODULE_VERSION, macro sets the version of the module.

We can add all these in our c file.

Eg: $modinfo ./hello.ko

filename: /home/linux....
version : 1.0.1
License : GPL
author : Linux trainer
description: Hello world
srcversion : A60LFE68 ....
depends :
retpoline : Y
name : hello
vermagic : 5.0.0-23-generic ....

understanding modinfo output :-

vermagic :- when loading a module, strings in vermagic value are checked, if they match. If they don't match, you will get an error and Kernel refuses to load the module.

intree :- All Kernel modules start their developments as out-of tree. Once a module gets accepted to be included, it becomes an in-tree-module.

srcversion :- is an MD4 hash of the source code used to compile the Kernel module. It is calculated automatically at build time from modpost script.

Retpoline :- "Retpoline" was introduced to be a solution to mitigate the risk of (spectre) → bug. Intel has to be provided patch
avoid

Use Case of srcversion :-

Let's say we provide a ko to client. But client reported that has a bug. We have released new ko which fixes the bug. But still client explains it doesn't fix. But we are pretty sure that it should fix. we can check whether client has loaded fix ko(os) not - we can check "srcversion" of ko file that loaded by using modinfo. Then we can compare with fix file & confirm.

5a. MODULE_INFO macro :-

$ cd linux-5.2.8/
$ cd include
$ grep -nr 'MODULE_VERSION'.
o/p: #define MODULE_VERSION(_version) MODULE_INFO(version, _version)
$ grep -nr 'MODULE_AUTHOR'.
o/p: #define MODULE_AUTHOR(_version) MODULE_INFO(author, _author)

5b. objdump on Kernel module

$ file ./mod_info.ko
o/p:
./mod_info.ko: ELF 64-bit LSB relocatable, ... not stripped.
(executable linkable format)

An ELF object file consist of various named sections.
Some of them are basic parts of an object file, for the .text Section Contains executable code that a loader loads.

To see all the sections,

$ objdump --section-headers ./mod-info.ko → any ko

To see the modinfo section:

$ objdump --section-headers --section=.modinfo --full-contents
(space) (space) (space)
./mod-info.ko

Eg:
$ objdump --section-headers ./mod-info.ko | less
o/p:
list all sections of ko.

Eg: To see modinfo section,

$ objdump --section-headers --section=.modinfo --full-Contents
./mod-info | less

o/p
Sections:
Idx Name
6 .modinfo

Contents of section .modinfo:
0000 ...
license=GPL version=1.0
os=linux

52. what happens if we Continuously call printK ?

printK → writes into Kernel buffer of some size, dmesg → print the Contents of buffer.

Eg: #include...
static void fun(void)
{
int i;
for(i=0; i<50000L; i++) → Here we Continuously write into Kernel buffer.
printK(KERN_INFO "Hello linux cnt: %d\n", i);
}
static int test_hello_init
{
?...
Start...
- - - -

→ So, Kernel buffer is circular one, so it is overwriting.

53. printK Kernel ring buffer size:-

printK is implemented by using ring buffer in Kernel with size of __LOG_BUF_LEN bytes where __LOG_BUF_LEN Equals (1 << CONFIG_LOG_BUF_SHIFT)

Command: cat boot/config-`uname -r` | grep CONFIG_LOG_BUF_SHIFT

o/p: CONFIG_LOG_BUF_SHIFT = 18

So 2^{18} is buffer size

It can be changed by 2 options.

① make menuconfig → General setup → Kernel log buffer

Note: dmesg reads by default a buffer of max 16392 bytes, if you use a larger logbuffer you have to invoke dmesg with -s parameter

Eg: ### Config-log-BUF-SHIFT 18 = 256K

$ dmesg -s 256000

first method need to rebuild the Entire Kernel.

② By using Kernel Command line:

log_buf_len=4M

→ sets the Kernel buffer length to 4 Megabytes

54. log levels (KERN_INFO, KERN_ALERT, ...)

There are Eight possible loglevel strings, defined in the header <linux/kern_levels.h>

#define KERN_SOH "\001" /* ASCII start of header */

#define KERN_SOH_ASCII '\001'

55. Default printk log level

what happens if we don't specify log level?

Eg: static mod_init(void)
{
  printk("%s: Init\n", __func__);
  return 0;
}
static mod_exit(void)
{
  printk("%s: In exit\n", __func__);
}

Both doesn't have log levels.

When Compile, no errors & warnings.

$ sudo insmod hello.ko (or) sudo rmmod

$ dmesg -X

o/p: => Default log level

kern: warn [77117.364] mod_init: In init.

Go to cat /proc/sys/kernel/printk

They are associated with following variables:

- console_loglevel: level under which messages are on the console device

default_message_loglevel: priority level that is associated by default with messages for which no priority value is specified

minimum_console_loglevel: Minimum level to allow a message to be logged on the console device.

Maximum_console_loglevel: maximum level

$echo 8 > /proc/sys/kernel/printk

will change console_loglevel

$cat /proc/sys/kernel/printk

4 4 1 7

↓ ↓ ↓ ↓

Conde default min max

56. Kernel messages on Console

What is Console?

→ for linux, a console is a device to which we can write text data & read text data.

→ By default, Console is the (Text mode) screen & Keyboard. One can switch to that Console by pressing Ctrl+Alt+Fn where n is 1 to 6.

→ When one boots his pc, Kernel prints lots of messages, like "Initialising this ...", "initializing that ...". These all get printed via printk that sends message to the console driver. (normally printf → prints in Gnome, why?) (over terminal)

so, why don't Kernel message appear when in graphics mode?

In linux, graphics mode is implemented not inside the Kernel (& thus it cannot print messages in graphics mode), but as a usermode process called X, (or sometimes X server.

→ Every pgm that wants to say, display a window, sends a message via IPC to X server and says it how it (x-server) should draw the window.

→ of course, this message passing is implemented in shared library, so from the application writer point of view, it is just a call to function that displays the window.

→ xterm is one of many graphical applications (Konsole and gnome-terminal are other two - well known pgms that emulates a terminal)

$ ps -ef | grep 'X' → Caps

gdm ..... usr/bin/Xwayland: ← x server

(ie) x-server → implemented in userspace not in Kernel.

↑ very simple understanding

→ Xterm passes all keystrokes to the shell it runs and the resultant text is drawn graphically using the selected font (by using some functions in shared library that send messages to X server & tell it to draw the characters)

→ Kernel messages are not output from shell, so xterm does not receive them, and therefore does not print (err... draw) them

To Console:

Switch to CTRL+ALT+F4 (or) any other.

② To Return to GUI :-

CTRL+ALT+F2

$ pwd →

ask user → "welcome" > my pc

→ Try to load any module

→ check dmesg

→ check $ cat /proc/sys/kernel/printk

o/p 4 4 1 7 → gives error, alert... not "warn" (i.e) kern_warn

Console level

→ $ sudo dmesg -n 5 → this also gives kern_warn

→ $ sudo command ./ko

→ dmesg → shows exit message

57. short printk macros :-

There are other fns that we can use in place of printk() in order to have more efficient messages and compact & readable code.

```
#define pr_emerg(fmt, ...) \
    printk(KERN_EMERG pr_fmt(fmt), ##__VA_ARGS__)
#define pr_info(fmt, ...) \
    printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
...
```

So instead of printk(KERN_INFO "%s:%i: In init\n", __func__); we can use directly pr_info("%s: In init\n", __func__);

Likewise we can use macro directly

58. Enable pr_debug message :-

Let's say

```
static int test_hello_init(void)
{
    printk(KERN_DEBUG "%s: In init 1\n", __func__);
    pr_debug("%s: In init 2\n", __func__);
}
```

In above, only printk will print, but pr_debug is not printing.

By default, it is not enabled.

To enable:

In makefile,

obj-m := hello.o

ccflags-y := -DDEBUG

Now, make the makefile, build → pr_debug will print.

59. LKM which prints floating point number ?

```
static int test_hello_init(void)
{
    float f = 2.56;
    printk("%s: In init, value of f is %f\n", __func__, f);
    return 0;
}
```

1) make file
2) sudo insmod
3) dmesg → See error

(ix). printk will not print floating numbers.

%f ⇒ format specifier will work in printf not in printk.

60. No use of floating point in Kernel.

When a user-space process uses floating-point instructions, Kernel manages the transition from integer to floating point mode.

Why is floating point mode off?

→ Many pgms don't use floating point (or) don't use it on any given time slice; and saving FPU registers & other FPU states takes time; therefore an OS Kernel may simply turn FPU off. presto, no state to save and restore, & therefore faster Context-Switching.

If a program attempts an FPU op,

→ the pgm will trap into Kernel,
Kernel will turn FPU on,
restore any saved state that may already exist and
then return to re-execute FPU op.

At Context switch time, it knows to actually go through the state save logic. (And then it may turn FPU off again.)

The reason that Kernel doesn't particularly need FPU ops and also needs to run on architectures without an FPU at all. Therefore it simply avoids the Complexity & runtime required to manage its own FPU Context by not doing ops for which there are always other Software Solutions.

61. Limiting printk messages - printk_ratelimit :-

printk is used in kernel programs to log messages and is very useful while debugging the kernel pgm's.

But kernel log being through big, is restricted & unnecessary logging at times can lead to loosing relevant messages.

Thus kernel provides a function, printk_ratelimit to restrict the logging using which we can set a limit on the no. of prints that we want our pgm to do.

The limit on no. of prints is set in file /proc/sys/kernel/printk_ratelimit_burst

```
$ cat /proc/sys/kernel/printk_ratelimit_burst
10
```

The printk_ratelimit function will allow 10 prints before it starts blocking further prints.

The printk_ratelimit fn returns 1 as long as no. of prints do not exceed the limit. Once the limit is reached, it start returning 0.

Thus fn can be used as a condition for an "if" statement to decide whether to print a message (or) not.

printk will be enabled again after time interval in 5ecs mentioned in file /proc/sys/kernel/printk_ratelimit

```
$ cat /proc/sys/kernel/printk_ratelimit
5
```

Eg: To understand

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/delay.h>
Module_like ...
static int hello_init(void)
{
  int i;
  for(i=0;i<20;i++) {
  if(printk_ratelimit()) {
   pr_info("printing %d\n", i+1) }
  else {
   pr_info("Sleeping for 5 Seconds\n");
  }
}
static void hello_exit(void)
{
   pr_info("Done\n");
}
```

→ return 1 for first 10 times then return 0.

Generally it needs 5 seconds to reset printk_ratelimit fn - here we don't provide any time, so else loop runs for 10 times

o/p: printing 1
printing 2
.
.
printing 10
sleeping for 5 seconds } 10 times.
sleeping for 5 seconds

Let's try the same with below changes.

```
use {
pr_info("sleeping for 5 seconds");
msleep(5000);
}
```

o/p: printing 1
printing 2
.
.
printing 10
sleeping for 5 second
printing 12
printing 13
.
.
printing 20

62. Limiting printk messages - printk_once:

printk_once is fairly trivial - no matter how often if we call it, it prints once & never again.

```
#include..
static int hello_init(void) {
int i;
for(i=0; i<20; i++)
{
  printk_once(KERN_INFO "printing %d \n",i);   → print only once.
}
return 0;
}
```

If we change,

```
for(int i=0; i<20; i++)
{
  printk_once(KERN_INFO "printing %d \n",i);   → only once
}
printk_once(KERN_INFO "printing %d \n",i);   → This also prints
return 0;
}
```

(1) This is for line by line, not for the whole module.

(2) If we use, printk_once(KERN_WARN, "...)
↓
will error. We can't use KERN_WARN with once.

63. Avoiding default newline behaviour of printk

Eg:

```
Static int hello_init (void)
{
printk(KERN_INFO "%s : Hello", __func__);
printk(KERN_INFO "World\n");
return 0;
}
```

o/p: hello_init : Hello
World

Why "World" doesn't print with Hello? since we didn't add '\n' to Hello?

(ix) Default behaviour of printk is adding newline.

How to print a message in one single line in Linux Kernel?

To prevent a new line from being started, use

KERN_CONT → Kernel Continue

(ix)

```
Static . . .
{
printk(KERN_CONT "%s : Hello", __func__);
printk(KERN_CONT "World\n");
}
```

Not log level (x)

64. printing hex dump - print_hex_dump :-

```
void print_hex_dump(const char *level, const char *prefix_str,
int prefix_type, int rowsize, int groupsize, const void *buf,
size_t len, bool ascii);
```

Given a buffer print_hex_dump prints a hex + ASCII dump to the kernel log at specified kernel log level with an optional leading prefix.

level: kernel log level (e.g. KERN_DEBUG)

prefix_str: string to prefix each line with;

prefix_type: Controls whether prefix of an offset, address or none is printed (%DUMP_PREFIX_OFFSET, \"DUMP_PREFIX_ADDRESS" \"DUMP_PREFIX_NONE"\)

rowsize: no. of bytes to print per line; must be 16 (or) 32

groupsize: no. of bytes to print at a time (1, 2, 4, 8; default = 1)

buf: data blob to dump

len: no. of bytes in buf

ascii: include ASCII after hex o/p

Eg:

```
#include
```

```
static int test_init(void)
{
char buffer[] = "Hello world. Linux is the best os";
print_hex_dump(KERN_INFO, "buf: ", DUMP_PREFIX_ADDRESS,
    16, 1, buffer, sizeof(buffer), 1);
```

(include ASCII after o/p; 16 bytes)

o/p:

```
[1702...] buf: 00000fde : 48 65 6c ... 6e Helloworld.Linu
[1710...] buf: 00000056 : 75 78 20 ... 48
```

If we change

```
print_hex_dump(..., ..., ..., 0);
```

(0 → ASCII)

o/p:

```
[1702]: buf: 0000 ... 6e
```

(NO Helloworld Linu)

65. printing hex-dump - print_hex_dump_bytes:

```
void print_hex_dump_bytes(const char *prefix_str, int prefix_type,
const void *buf, size_t len)
```

shorthand form of print_hex_dump with default params.

log level of KERN_DEBUG, rowsize of 16, groupsize of 1 & ASCII output included.

66. Dynamic debug:-

```
$cd /boot
$boot$ uname -r
5.0.0-23-generic
boot$ cat Config-5.0.0-23-generic | grep CONFIG_DYNAMIC_DEBUG
CONFIG_DYNAMIC_DEBUG=Y
```

→ Using pr_debug() globally will cause a tremendous amount of logging so it is not very practical. To make the debug level more manageable, dynamic debugging was introduced.

→ We can activate it with CONFIG_DYNAMIC_DEBUG

→ DYNAMIC debug is designed to allow you to dynamically enable/disable kernel code to obtain additional kernel information.

→ Currently, if CONFIG_DYNAMIC_DEBUG is set, then all pr_debug() / dev_dbg() and print_hex_dump_debug() calls can be dynamically enabled per-call-site [per module (or) file (or) folder].

Kindly watch lecture for more info.

68. What happens if i try to load non-ko file with insmod?

```
$ touch hello.ko
$ chmod +x hello.ko
$ file hello.ko
o/p: hello.ko: Empty
$ modinfo ./hello.ko
```

→ Error: Couldn't get modinfo from 'hello': Invalid argument

$ Sudo insmod ./hello.ko

O/p: insmod: ERROR: Could not insert module ./hello.ko: Invalid parameters

69. strace on insmod Command :-

$ sudo -s

$ strace -o /tmp/strace.txt insmod ./hello.ko

Shows lot -

Last - "finit_module( ) ..

System Calls:

$ man finit_module

70. Find out name of module from .ko

$ vi modname_test.c

#include

Char * modname = __Stringify(KBUILD_MODNAME); (convert to string)

static int test_(void)

{

printk("%s : In init \n", __func__);

printk("%s : Module Name : %s loaded \n", __func__, modname);

}

O/p:

In init

Module Name: "modname_test" loaded

How it works?

$ make V=1 | grep KBUILD_MODNAME

We can find it on terminal.

71. How to dump Kernel stack?

Calling dump_stack() will cause a stack trace to be printed at that point.

static int myinit(void)

{

pr_info("dump_stack myinit\n");

dump_stack();

pr_info("dump_stack after\n");

return 0;

}

$ sudo insmod ...

$ dmesg | less

It shows Call trace..

72. Kernel panic :-

-> Kernel panic is an error in Kernel Code.

-> On Kernel panic, Kernel stop running immediately to avoid the data loss (or) other damage.

The reason to stop running is to protect your Computer. for example, if the module which is Controlling the fan fails to load, Kernel generates a panic & freezes the system.

What are reasons for Kernel panic?

→ HW (or) SW issue (Eg. unable to start init process)
→ Bug in Kernel driver
→ Defective (or) Incompatible RAM

What happens decides to panic, it calls panic() function which dumps some debug information & depending on configuration reboots the system.

Configure Kernel to reboot on Kernel panic

By default, Kernel will not reboot on Kernel panic. There are two ways, by which you can instruct Kernel to reboot.

1. Kernel Command line: Add "panic=N" to the Kernel Command line, for the Kernel to reboot after N seconds. (Don't explain)
2. Proc file system: echo N > /proc/sys/kernel/panic for Kernel to reboot after N seconds on reboot. Note this setting is not persistent on reboot (i.e cleared after reboot)

2nd method:

$ cat /proc/sys/kernel/panic
o/p:-
$ echo '5' > /proc/sys/kernel/panic
'If' permission denied

$ sudo -s
$ echo '-5' > /proc/sys/kernel/panic

73. Kernel panic Example :- (↑ set 5 seconds, must read)

```
#include ---
static int test_panic_init(void)
{
    printk(KERN_INFO "%s: In init \n", __func__);
    panic("Hello Kernel I am Causing the panic\n");
    return 0;
}
```

Once we insert this module, linux gets restarted.
After restart, we can check value, it will be '5'.
→ Yes, it works ☺ with 10 s.

What is oops?

An oops is similar to segfault in user space. Kernel throws oops message when an exception such as accessing invalid memory location happens in the Kernel code.
upon oops, the Kernel performs the following operations:
① Kills the offending process
② Prints the information which can help developers to debug
③ Continuous execution.

Note: After oops, system cannot be trusted further as the some of the locks (or) structures may not be cleaned up.

An oops Message Contains the following information:

→ processor States
→ Contents of CPU Registers at time of exception.
→ Stack trace
→ Call trace

75. oops Example

Eg:

```
static int --
{
   *(int *)(0x12) = 'a';
   :
}
```

$ make
$ sudo insmod ./oops.ko
o/p: Killed ⟶ shows killed the offending process
$ sudo dmesg
o/p: ... error (we can see in red colour)

75. What is BUG & example?

The most frequently found macro in linux device drivers is BUG_ON/BUG and WARN_ON

BUG_ON(Condition)
(or)
if (Condition) ⟶ if this condition true, then it call BUG. (i.e) we understand that there is some prblm happened in source code.
BUG()

→ grep in drivers folder for BUG_ON (gave 5000 plus and for WARN_ON gave more than 6000 count)

What does BUG() macro do?

→ prints the contents of registers, → prints stack Trace
→ Current process dies

What does WARN() macro do?

→ print the contents of registers → prints stack Trace.

Eg:

```
static int ...
{
   BUG(); ⟶ should not be directly called, just to simulate.
}
```

o/p: Segmentation fault (Core dumped)
$ sudo dmesg

77. Can we remove module after bug/oops?

Please note that after loading this module using insmod, you cannot unload this. If you try to call rmmod, you will get "Module in use" error.

78. How to define preprocessing symbol in Makefile?

(i.e.)

```
Static int test_hello(void)
{
#ifdef DEBUG
printk(KERN_INFO "%s: In init\n", __func__);
#endif
 return 0;
}
Static void Test_hello_exit(void)
{ --
 }
```

→ How we enable & tell to makefile?

makefile

```
obj-m := symbol.o
```

#Kbuild understands a make variable named CFLAGS_modulename.o to add specific C flags when Compiling this unit

```
CFLAGS_symbol.o := -DDEBUG
```

#This will be applied to all of the source files Compiled for your module with Makefile.

```
#ccflags-y := -DDEBUG
all:
 ---
clean:
 ---
```

80. How to find out how many CPU's are present from userspace & Kernel space?

```
$ Cat /proc/cpuinfo
$ Cat /proc/cpuinfo | grep processor
$ Cat /proc/cpuinfo | grep processor | WC-l
```

From Kernel space:

```
Eg:
Static int --
{
 pr_info("number of online CPUs is %d\n", num_online_cpus());
 return 0;
}
```

① make
② sudo insmod
③ check dmesg.

81. process representation in Linux kernel and process states

→ Linux kernel internally refers processes as tasks.
→ Kernel stores the list of processes in Circular doubly linked list called the task list.
→ Each task/process is represented in Kernel with struct task_struct (defined in <linux/sched.h>).
→ This data structure (task_struct) is huge (1.7 kilobytes)

Containing all the information about a specific process.

Let's write a module/driver which reads the circular linked list & prints the following information for us:

→ process name → process ID → process state.

Before that, we should know what are different states a process can be:

TASK_RUNNING(R): process is either currently running or on a run-queue waiting to run.

TASK_INTERRUPTIBLE(S): process is sleeping/blocked. Can be runnable/awaken by a signal.

TASK_UNINTERRUPTIBLE(D): Similar to TASK_INTERRUPTIBLE, but does not wakeup on a signal.

__TASK_STOPPED(T): process execution has stopped. This happens when task receives SIGSTOP, SIGTSTP, SIGTTIN (or) SIGTTOU signal (or) if it receives any signal while it is being debugged.

We can find the states using ps -el command.

This ps -el → takes directly from file/proc system, does not circular buffer.

82. LKM example demonstrating process name, process id and process s :-

(i) Now w, we don't use proc file system to get. we will access circular double linked list.

print_task.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched/signal.h>
char buffer[256];
char * get_task_state (long state)
{
switch (state) {
  case TASK_RUNNING:
        return "TASK_RUNNING";
  case TASK_INTERRUPTIBLE:
        return "TASK_INTERRUPTABLE";
  case TASK_UNINTERRUPTIBLE:
        return "TASK_UNINTERRUPTIBLE";
  case __TASK_STOPPED:
        return "__TASK_STOPPED";
  case __TASK_TRACED:
        return "__TASK_TRACED";
```

```
default:
{
    sprintf(buffer, "unknown type : %ld\n", state);
    return buffer;
}
}
static int test_init(void)
{
struct task_struct *task_list;
unsigned int process_count=0;
pr_info("%s: In init\n", __func__);
/* define for_each_process(p) \ <-- definition:
for (p=&init_task; (p=next_task(p))!=&init_task;)
*/
for_each_process(task_list)
{
pr_info("process: %s \t PID: [%d] \t state: %s\n",
    task_list->comm, task_list->pid,
    get_task_state(task_list->state));
    process_count++;
```

```
$ sudo
$ sudo insmod
$ dmesg | less
```

83. LKM example demonstrating Current Macro

While writing a module if we want to get information about the current process that is running in kernel, we need to read the "task_struct" of the corresponding process.
The kernel provides a easy way to do this by providing a macro by the name "current", which always return a pointer to the "task_struct" of the current executing process.
This macro has to be implemented by each architecture.
Some architecture stores this in a register, some stores them in bottom of kernel stack of process.

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <asm/current.h>
static void current_exit(void)   <-- this will call, when we use rmmod. So, current is, rmmod
{
printk("Current pid: %d, Current process: %s\n",
        current->pid, current->comm);
}
```

```
Static int Current_init(void)
{
  printk("Current pid : %d, Current process: %s \n",
         Current->pid, Current->comm);
  return 0;
}
```

84. LKM Current macro part-2:-

Little modification:

Now only few tasks will be printed. (i.e) from current task to init_task (i.e) again from circular linked list)

```
struct ...
for (task= Current; task!= &init_task; task = task->parent)
{
pr_info("process: %s \t PID: [%d] \t state : %s \n",
        task->comm, task->pid, get_task_state( task->state));
process_count++;
}
```

85: LKM which accepts PID as argument & print process & parent

```
Static unsigned int PID= 1; -> default
module_param (PID, uint, 0400);
void print_task( struct task_struct * task)
{
printk("process: %s, parent process is: %s \n", task->comm, task->parent->comm);
}
```

```
Static int init_find_task(void)
{
  struct task_struct *task=NULL;
  for_each_process (task) {
  if ( task -> pid == (pid_t) PID) {
   print_task (task);
  }
}
return 0;
}
```

$ Sudo insmod ./task.ko

o/p process : systemd , parent process : swapper/0

Let's find another PID to check

$ ps -ef | less (we choose 45)

$ Sudo insmod ./task.ko PID=45

o/p process : netns, parent process: kthreadd

86. process Memory Map:-

struct mm_struct -> Contains list of process VMAs, page tables, etc...

All info related to process address space is included in an object called memory descriptor of type mm_struct

accessible via Current->mm

```
struct mm_struct {
/* pointer to head of list of memory region objects */
  struct vm_area_struct *mmap;
/* pointer to root of red-black tree of memory region objects */
  struct rb_root mm_rb;
/* pointer to last referenced memory region object */
  struct vm_area_struct *mmap_cache;
};
```

Linux implements a memory region by means of an object of type vm_area_struct.

$ cat /proc/2047/maps → any process id

87. Example for printing process memory map.

88. Introduction to Kernel Threads:

$ ps -ef

o/p:

[ ] → this is kernel threads.

→ Kernel thread is a linux task running only in kernel mode. It is not created by running fork() (or) clone() system calls.

What is the use of Kernel Thread?

Kernel Threads helps kernel to perform operations in background.

Examples of Kernel Thread:-

1. ksoftirqd is per CPU kernel runs processing softirqd.
2. kworker is a kernel thread which processes work queues.

Difference between Kernel Thread & user Thread?

Both Kernel thread & user thread are represented by task_struct. The main difference is that there is no address space in kernel threads. mm variable of task_struct is set to NULL.

How to create Kernel thread?

Api for creating Kernel thread.

```
#include <linux/kthread.h>
struct task_struct *kthread_create(int (*threadfn)(void *data),
void *data, const char name[], ...)
```

Parameters:-

threadfn → the fn which thread should run.
data → Argument for thread fn.
name → printf style format for name of the kernel thread.
Return value → pointer to struct task_struct

Note:- kthread_create only creates the thread but doesn't run the thread, we need to call wake_up_process() with

return value of kthread_create as an argument to the wake_up_process for thread fn to run.

Alternative:- Linux provides API which creates the Kernel thread & calls wake_up_process().

struct task_struct * kthread_run (int (*threadfn)(void *data), void *data, const char name[], ...)

To stop the Kernel thread:-

int kthread_stop (struct task_struct *k);

Note:- If you don't stop the Kernel thread in your module_exit function, you will get oops message.

K_thread_stop is a blocking call, it waits until the fn executed by thread exits. kthread_stop() flag sets a variable in task_struct variable which function running in while(1) should check in each of its loop.

```
int threadfun (void *data)
{
  while (!kthread_should_stop())
  {
  } // perform operations here.
  return 0;
}
```

89. Kernel thread Example - kthread_create :-

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>          //Kmalloc()
#include <linux/uaccess.h>       //copy_to/from_user()
#include <linux/kthread.h>       //Kernel threads
#include <linux/sched.h>         //task_struct
#include <linux/delay.h>

static struct task_struct *my_thread;
int thread_function(void *pv)
{
  int i=0;
  while (!kthread_should_stop())
  {
  printk(KERN_INFO "In thread fun:%d\n", i++);
  msleep(1000);
  }
  return 0;
}
static int my_driver_init(void)
{
  my_thread = kthread_create(thread_function, NULL, "myTHREAD");
```

```
if(my_thread) {
wake_up_process(my_thread);
return 0;
} else {
printk(KERN_ERR "Cannot Create Kthread\n");
return -1;
}
}
void my_driver_exit(void)
{
Kthread_stop(my_thread);
printk(KERN_INFO "Device driver DONE - Removed !!!\n");
}
```

90. Kernel Thread example Kthread_fun :-

```
Kthread_run(thread_function, "NULL", "MYThread");
```

Same example in (89).

91. Two Kernel threads example :-

Let's launch two threads and see if they actually run in parallel.

Outcome: Two threads count to dones from 0 to 9 in parallel.

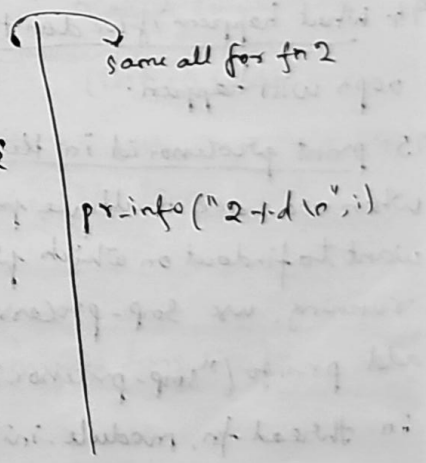
Each line has o/p of form:

<thread id><count>

E.g. <1><2>

The threads almost always interleaved nicely, thus confirming that they are actually running in parallel.

```
static int myinit(void)
{
Kthread1 = Kthread_Create(fn1, NULL, "mythread1");
Kthread2 = Kthread_Create(fn2, NULL, "mythread2");
wake_up_process(Kthread1);
wake_up_process(Kthread2);
return 0;
}
static int fn1(void *data)
{
int i=0;
while(!kthread_should_stop()) {
pr_info("1->%d\n", i);
usleep_range(1000000, 1000001);
i++;
if(i==10)
i=0;
}
```



92. Can we have two Kernel threads with same name?

Yes.

93. What happens if we don't use kthread_should_stop() in thread function?

kthread_stop(thread_fn); → This is blocking call (ie) It waits until thread_fn is executed fully. But thread fn doesn't have kthread_should_stop(), so it is already exited before we call/remove module [module_exit → call kthread_stop].

→ It creates oops in Kernel./warning.

94. What happens if we don't call kthread_stop() in module_exit?

oops will happen.

95. print processor id in Kernel module?

When we have multiple processors present in System, and want to find out on which processor your driver code is running, use smp_processor_id().

add printk("smp_processor_id : %d\n", smp_processor_id());

in thread fn, module_init, module_exit fns.

We can see, module_init runs in one processor, thread fn is running & changing in processors & module_exit in another pr

96. Linux Kernel thread example of race condition:-

Eg: 2 threads,

```
static int inc_thread(void *data)
{
int i=0;
printk("Thread started\n");
for(i=0; i<LoopCount; i++) {
Counter++;
}
printk("Thread Completed\n");
return 0;
}
```

① If we pass LoopCount as 10000

o/p: Thread started } 1st
Thread completed }
Thread started } 2nd one
Thread Completed }
Counter => 20000
→ (ie) No race condition

② If we pass L.C as 900000

o/p: Thread started } 1st
Thread started } 2nd
Thread Completed } 1st
Thread Completed } 2nd
Counter => (8--;-
(ie) race condition occurs.

97: Linux_Version_CODE macro:-

We have seen function arguments changes b/n Linux Kernel versions for example "access_ok".

Before 5.0 Linux version:-

int access_ok(int type, void *addr, unsigned long size);

After 5.0:-

int access_ok(void *addr, unsigned long size);

If you want to support multiple Kernel versions, you will find yourself having to code conditional compilation directives. The way to do this to compare macro LINUX_VERSION_CODE to the macro KERNEL_VERSION.

LINUX_VERSION_CODE:- This macro expands to the binary representation of the Kernel version. One byte for each part of the version release number. Eg: 5.0.0 = 0x050000 = 327680.

Header file: <linux/version.h>

from Kernel Top Level Makefile.

LINUX_VERSION_CODE = $(VERSION) * 65536 + $(PATCHLEVEL) * 256 + $(SUBLEVEL)

Eg 5.0.0 = 5 * 65536 + 0 * 256 + 0 = 327680

If we open "top level makefile" in linux source code, we can find

$ vi lib/modules) 'uname -r'/build

press Enter for opening anyone.

Eg:

```
#include <linux/version.h>
static int test_init(void)
{
  pr_info("Linux_version_code:%d\n", LINUX_VERSION_CODE);
}
```

98. Kernel Version Macro: #include <linux/version.h>

Eg:

```
static int test_init(void)
{
  pr_info("Kernel version for 5.0.18 is %d\n", KERNEL_VERSION(5,0,18));
  pr_info("Kernel version for 2.6.2 is %d\n", KERNEL_VERSION(2,6,2));
}
```

This macro is used to build an integer code from individual numbers that build up a version number.

#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))

Header file: <linux/version.h>

99. UTS_RELEASE

This macro expands to a string describing the version of this Kernel tree.

Header file: #include <generated/utsrelease.h>

Eg:

```
#include <generated/utsrelease.h>
static int test_init(void)
{
  pr_info("UTS Release: %s\n", UTS_RELEASE);
  return 0;
}
```

100. Linux Kernel Module example supporting multiple versions :-

```
static int test_init(void)
{
 //Now depends on our kernel version, we could log a different
  Comment in kernel log
#if LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,10)
 printk(KERN_INFO "KERNEL VERSION: Hello old Kernel %s\n",
        UTS_RELEASE);
#elif LINUX_VERSION_CODE >= KERNEL_VERSION(5,0,0)
 printk(KERN_INFO" KERNEL VERSION: Hello New Kernel %s\n",
        UTS_RELEASE);

#else
    printk(KERN_INFO"Kernel version: Hello moderate Kernel %s\n",
    UTS_RELEASE);
#endif
 return 0;
}
```

101. o/p of pre-processed stage :-

If we don't want to create ko, but need to check preprocessing stage o/p,

```
$make -C /lib/modules/`uname -r`/build M=$(pwd) hello.i
$ vi hello.i
```

(M=$(pwd) → small letter)

102. --init macro :-

This macro informs the Compiler to place this function into special section (.init.text) and this memory is discarded after the function call.

What's benefit of adding this macro?

Functions specified using module_init are Initialization functions runs only once.

Kernel can free RAM by removing memory used by this particular function.

Conclusion :- Functions marked using --init macro are discarded after the function call.

Ex: static int --init test_hello_init(void)

① Once we load the module, [without --init macro]

```
$ sudo cat /proc/kallsyms | grep hello1
```

we can see test_hello_init in memory

② Once we load the module, [with --init macro]

```
$sudo cat /proc/kallsyms | grep hello2
```

We can't see test_hello_init in memory, because once fn. called is completed, Kernel freeup the memory.

103. Can we use --init macro for builtin modules?

Modules can be a) Loadable b) Built-in.

Kernel build system looks for all functions flagged with --init, across all pieces of Kernel & arranges them so that they will all be in same block of memory.

Then when Kernel boots, it can free that one block of memory all at once.

When you boot your Kernel & see something like 'Freeing unused Kernel memory: 236K freed' this is precisely what Kernel is doing.

$ dmesg | grep 'Freeing unused'.

104. --exit macro :-

--exit is a macro defined in linux/init.h?

What is benefit?

When your Kernel module is configured as built-in instead of loadable, there is no need of exit function.

Adding --exit macro causes the compiler to discard this function when module is configured as 'built-in'. This saves on code space as the exit functions won't be called.

For loadable modules, this macro doesn't make any effect

106. --initdata & --exitdata macro :-

--init / --exit is for functions & --initdata / --exitdata is for variables.

ex: static char buf[] --initdata = "LWL";

Static int --init test_hello_init (void)

{ pr_info ("%s: In init: %s\n", --func--, buf);

return 0;

}

After inserted module, this buffer will also not in RAM.

107. How do you list builtin modules?

Kernel modules are pieces of code that can be loaded & unloaded into Kernel upon demand.

It can be configured in 2 ways.

1. Built-in (part of Kernel) : Always present
2. Loadable : Can be loaded / unloaded as per the need.

lsmod command lists all modules that are built into Kernel.

$ cd /lib/modules/'uname -r'.

$ vi modules.builtin.

108. How to load modules automatically?

① $ pwd → take the path. (Ko file)

② $ sudo ln -s /home .... /hello.ko (/lib/modules/`uname -r`/kernel/drivers/misc)

↓ creating symbol

③ $ ls -l ( )

④ Sudo depmod -a → update module

⑤ Sudo modprobe hello1 → [just to check], BCoz we want to load automatically.

⑥ Sudo modprobe -r hello1 ← remove

⑦ Sudo vi /etc/modules

⑧ add ~~hello~~ hello1

⑨ reboot

⑩ lsmod | less ⇒ So, we can see hello in built in module

109. Blacklisting Kernel module.

Blacklisting is a mechanism to prevent KM from loading

Why do we need to blacklist kernel modules?

→ Associated hardware is not needed.

→ If loading that module could cause problems.

Ex: There may be two KMs to control same piece of HW & loading them together would result in conflicts.

How to blacklist?

Create a .conf file inside /etc/modprobe.d/ and append a line for each module you want to blacklist using blacklist keyword.

blacklist <module name>

finally run, the following commands to rebuild initramfs

$ sudo update-initramfs -u.

$ $ lsmod | grep e1000 → ethernet driver [we are going to blacklist it]

$ cd /etc/modprobe.d/

$ sudo -s

$ ls

$ touch bwl.conf

$ vi bwl.conf

↓ In this file,

blacklist e1000

$ lsmod | grep e1000

$ modinfo e1000 | less

$ sudo update-initramfs -u

NO space — space

$ reboot

after reboot

$ lsmod | grep e1000

o/p: No output.

II. Blacklisting using Kernel Command Line :-

We can also blacklist modules from the bootloader.
Simply add module_blacklist=modname1, modname2, modname3
to your bootloader's kernel line
Note: This can be very useful if a broken module makes it impossible to boot your system.

Eg: $ lsmod | grep e1000
o/p: e1000
$ sudo vi /etc/default/grub → bootloader
(or need to update)

inside that

GRUB_CMDLINE_LINUX="... find_preseed=/preseed.cfg auto noprompt priority=Critical locale=en_US module_blacklist=e1000"
(1) part added

$ sudo update-grub --nopath
$ reboot
$ lsmod | grep e1000
o/p: no o/p.

III. Passing parameters to LKM loaded using modprobe

Setting module options:-

To pass options to modules which are automatically loaded using modprobe.
→ Create a file in /etc/modprobe.d/ with .conf extension
syntax: options module_name parameter_name = parameter_value

Note: Configuration files in the directory can have any name, given that they end with the .conf extension.

$ cd /etc/modprobe.d/
$ ls
$ grep -nr 'options'
o/p:
... : options bt87x index=2
:
:

II2. systool:-

systool can be used to print values of files present in module's sysfs directory.

/sys/module/<module name>

(or) $ systool -vm <modulename>
Eg: $ systool -vm <e1000>

Linux booting

BIOS → MBR → GRUB → Kernel → Init → Runlevel

1. BIOS :-
1. stands for Basic Input/output system
2. performs some system Integrity checks.
3. Searches, loads & executes the boot loader program.
4. It looks for bootloader in floppy, cd-rom (or) hard drive.
We can press a key (F12 (or) F2) but it depends on system during BIOS startup to change the boot sequence
5. once the bootloader is detected & loaded into the memory, BIOS gives the control to it.
6. So, in simple terms BIOS loads & executes MBR boot loader.

2. MBR :-
1. MBR stands for Master Boot Record.
2. It is located in 1st sector of the bootable disk. Typically /dev/hda (or) /dev/sda.
3. MBR is less than 512 bytes in size. This has 3 components
1) primary boot loader info in 1st 446 bytes
2) partition table info in next 64 bytes
3) mbr validation check in last 2 bytes.
4. It contains information about GRUB.
5. So in simple terms MBR loads & executes GRUB boot loader.

In my pc:

$ cd /dev/

$ ls | grep sda

o/p: $ /sda

$ /sda1

$ /sda2 ...

3. GRUB:-

1. It stands for Grand Unified Bootloader
2. If you have multiple kernel images installed on your system, you can choose which one to be executed.
3. GRUB displays a splash screen wait for few seconds, if you don't enter anything it loads the default kernel image as specified in the grub configuration file.
4. GRUB has the knowledge of the file system
5. It contains kernel & initrd images.

$ cd /boot

$ ls

6. So in simple terms GRUB just loads & executes kernel & initrd images.

4. Kernel:-

→ Mounts the root file system as specified in the "root=".

→ Kernel executes /sbin/init program.

→ Since init was the 1st program to be executed by Linux kernel, it has process id (PID) of 1. Do 'ps -ef | grep init' & check the pid.

→ initrd stands for initial Ramdisk.

→ initrd is used by kernel as temporary root file system until kernel is booted and real root file system is mounted. It also contains necessary drivers compiled inside which helps it to access the hard drive partitions & other hardware.

5. Init:-

→ $ cd /etc

Init file was there.

→ Init identifies the default initlevel & used that to load all appropriate pgm.

→ Typically we would set default run level to either 3 (or) 5. ubuntu has moved away from sysv init to upstart instead.

6. Runlevel programs:-

→ When Linux system is booting up, you might say "starting sendmail ... OK". Those are runlevel pgms executed from the run level directory as defined by your run level.

↔ These files can be seen in /etc/rc.d
→ under /etc/rc1 (or) rc2.. rc6.d, you would see the pgms that start with "S" and "K".
S → startup. K → Kill.
→ There are numbers right next to S & K in the programs names. Those are sequence number in which programs should be started (or) killed.
$ runlevel
o/p: 5
$ cd /etc/rc5.d
$ ls
... Here you can see programs start with S & K.

Networking

1) ping www.google.com
→ Confirms packets transferring
2) ifconfig → ethernet, local, router (wifi) ip.
To analyse packets,
$ sudo apt get install tcpdump
$ sudo tcpdump
↳ Capturing all packets from / to computer.
To Capture Certain,
$ sudo tcpdump -c 10 ↳ Capture 10 packets
To Capture packets from specific device
$ ifconfig
(wlo1) → name wifi provider which connects to system.
$ sudo tcpdump -c 5 -i wlo1

netstat → status of network

$ netstat -nr → shows Kernel Ip routing table
$ netstat -i → shows Kernel Interface table.
$ netstat -ta → shows active Internet Connections (servers & established)
↓
shows with domainname.

3.
1. $ netstat -tan → Same as above Command. one difference
2. is that shows with Ip address.

hosts:-

3. When we search for www.google.com → first Ip address is given to Computer, if it is found. It will run by local not in internet. then it checks dns server in internet - That's it.

4. Let's try
5. go to any webpage like indianbank & get the ip of indian bank. from some website

$ sudo su
6. $ vi /etc/hosts
edit there,
61.246.225.227 ib indianbank.

4 ↓
→ This should be edited before the below statement.
" # The following lines are desirable for IPV6 Capable host

Then open browser.
1) type "ib/" & press enter
2) It will show first time error like no page → Go to advance & proceed.
3) now Indianbank page opens.

tracing:

$ sudo apt install traceroute
$ traceroute www.google.com
: o/p occurs & it shows each & every step to connect google.com

Network mapper for tracking activity on network:-

$ sudo install nmap
$ nmap -v "ip-ourwifi" ⇒ $ nmap -v 192.168.43.78

To find range of Ip, Connected to device

$ nmap 192.168.43.78,1-100

otherwise

$ nmap 192.168.43.*

o/p' :

~~Along with~~ for lot of info,

$ nmap -A 192.168.43.78

'A' = "with os also"

Git

① Go to git hub, create one repository. take that link. → eg: Audioteam

② Go to Linux terminal, create one directory.

$ ~~cd~~ mkdir git-hub

$ cd git-hub

$

③ $ git init → Contains Employee

④ $ vi list.txt

⑤ $ git add list.txt

⑥ $ git commit -m "list added project started".

⑦ $ git branch -M main

⑧ $ git remote add origin https://github.com//vijs

⑨ $ git push origin main → push branch to repo

⑩ It will ask for username & password.

username → github username

password → not git hub ~~password~~ password, token has to be generated → click git hub page.

⑪ git branch release → Creating new branch in local m/c

⑫ git push origin release ← push to remote m/c
→ remote name

⑬ you can see changes in git hub

⑭ ~~just mode~~ git checkout release → switch to release

⑮ vi list.txt → do modify

⑯ git add list.txt → Sankar Cane

⑰ git commit -m "new list". → Commited in local m/c release branch

⑱ git push origin release → pushing ~~update~~ update release in remote

⑲ U can see "release had recent pushes 11 minutes ago" in git hub ⇒ Compare & pull request

⑳ click Compare & pull request

㉑ There they ask to merge [base: main] ← [Compare: release]

㉒ click merge. so new change in release is merged to main branch. All in remote.

Note: In local m/c, release has only new change, main is old list.

㉓ In remote, after merge, it will ask "Safe to delete release branch"

㉔ Let's delete remote branch release from terminal:

㉕ $ git push origin --delete release.

㉖ U can see no release branch in remote. But

it will be available in local m/c.

(27) Let's also try to delete release in local m/c.

(28) Note:- Main has old change, release has new change. new change in release is already pushed to remote release then it is merged to remote main, then remote release is deleted.

(29) $ git branch --delete release → try to delete release in local m/c

(30) o/p: error: The branch 'release' is not fully merged. If you are sure, you want to delete it, run 'git branch -D release'.

(31) :- Here we are not going to merge locally, we will do something different. → Intention:- At last main should update

(32) $ git branch -D release → Delete forcely

(33) deleted. But main has old changes.

(34) Let's do one thing. [main has no data of SankarCame

(35) $ git checkout main

(36) $ vi resign.txt → abdul

(37) $ git add resign.txt

(38) $ git commit -m "resign" → In local main m/c.

Note:- remote main is not updated.

(39) $ git push origin main → try to update resign on remote m/c.

(40) Error:- failed to push some ref

(41) Reason:- Because main local m/c doesn't has "SankarCame" update. But Remote main has the update

Fix → local m/c has to in up-to-date with remote before commiting new change in local branch to remote branch.

(42) $ git pull origin main

(43) $ cat list.txt → Here SankarCame → updated.

(44) now push resign.txt to remote. :)

(45) $ git push origin main

(46) U can see resign.txt in remote.

All work done.

Delete repository in remote:-

→ click repo name [Audioteam].

→ Go to settings [code Issues ... Settings]

→ Go to Danger zone [Delete this repository]