- ## What is a kernel module?
  - Kernel modules are pieces of code that can be loaded and unloaded into the *kernel* upon demand. They extend the functionality of the *kernel* without the need to reboot the system.

- ## Kernel Module basics
  - Kernel modules must have at least two functions: a "start" (initialization) function called **init_module()** which is called when the module is insmod-ed into the kernel, and an "end" (cleanup) function called **cleanup_module()** which is called just before it is rmmod-ed.

  - *init_module()* either registers a handler for something with the kernel, or it replaces one of the kernel functions with its own code (usually code to do something and then call the original function). The *cleanup_module()* function is supposed to undo whatever init_module() did, so the module can be unloaded safely.

  - **The simplest kernel module**

```
/**
 * hello.c : Getting started with simple kernel module
 */

#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void)
{
        printk(KERN_INFO "Welcome to the world of kernel programming\n");
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_INFO "There is no way to exit. see you again!\n");
}
```

- ## Introducing printk()
  - printk works more or less the same way as printf in userspace, so if you ever debugged your userspace program using printf, you are ready to do the same with your kernel code, e.g. by adding:

```
printk("A program is never finished until the programmer dies.\n");
```

  - This wasn't that difficult, was it? Usually you would print out some more interesting information like,

```
printk("Var1 %d var2 %d\n", var1, var2);
```

  - just like in userspace. In order to see the kernel messages, just use the dmesg command in one of your shells - this one will print out the whole kernel log buffer to you.

- Most of the conversion specifiers supported by the user-space library routine printf() - are also available in the kernel; there are some notable additions, including "%pf", which will print the symbol name in place of the numeric pointer value, if available.
- The supported format strings are quite extensively documented in Documentation/printk-formats.txt
- ***However please note***: always use *%zu*, *%zd* or *%zx* for printing *size_t* and *ssize_t* values. ssize_t and size_t are quite common values in the kernel, so please use the *%z* to avoid annoying compile warnings.

- **Log Levels**
  - If you look into real kernel code you will always see something like:

```
printk(KERN_ERR "something went wrong, return code: %d\n",ret);
```

  - where *KERN_ERR* is one of the eight different log levels defined in *include/linux/kern_levels.h* and specifies the severity of the error message.
  - Note that there is *NO comma* between the *KERN_ERR* and the format string (*as the preprocessor concatenates both strings*)
  - **The log levels are:**

Name	String	Meaning	Alias function
KERN_EMERG	“0”	Emergency messages, system is about to crash or is unstable	pr_emerg
KERN_ALERT	“1”	Something bad happened and action must be taken immediately	pr_alert
KERN_CRIT	“2”	A critical condition occurred like a serious hardware/software failure	pr_crit
KERN_ERR	“3”	An error condition, often used by drivers to indicate difficulties with the hardware	pr_err
KERN_WARNING	“4”	A warning, meaning nothing serious by itself but might indicate problems	pr_warning
KERN_NOTICE	“5”	Nothing serious, but notably nevertheless. Often used to report security events.	pr_notice
KERN_INFO	“6”	Informational message e.g. startup information at driver initialization	pr_info
KERN_DEBUG	“7”	Debug messages	pr_debug
KERN_DEFAULT	"d"	The default kernel loglevel	
KERN_CONT	“c”	"continued" line of log printout (only done after a line that had no enclosing \n)	pr_cont

  - **Memorising kernel log levels:**

- **E**veryone **A**lways **C**omplains **E**ven **W**hen **N**othing **I**s **D**ifferent
- **E**very **A**wesome **C**isco **E**ngineer **W**ill **N**eed **I**cecream **D**aily
- **E**very **A**lley **C**at **E**ats **W**atery **N**oodles **I**n **D**oors
- **E**veryone **A**ttends **C**lass **E**ach **W**eek **N**ot **I**f **D**ead

Compiling kernel module

- Kernel modules need to be compiled a bit differently from regular user-space apps. Kernel Makefiles are part of the kbuild system, documented **here**.

Below is a simple makefile to compile a kernel module:

```
obj-m += hello.o

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

- Now you can compile the module by issuing the command **make**. You should obtain an output which resembles the following:

```
debian@beaglebone:~/test/hello$ make
make -C /lib/modules/4.14.108-ti-r113/build M=/home/debian/test/hello modules
make[1]: Entering directory '/usr/src/linux-headers-4.14.108-ti-r113'
  CC [M]  /home/debian/test/hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/debian/test/hello/hello.mod.o
  LD [M]  /home/debian/test/hello/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.14.108-ti-r113'
```

- From kernel 2.6 a new file naming convention has been introduced for kernel modules - i.e. .ko extension (*in place of the old .o extension*) which easily distinguish them from conventional object files. The reason for this is that they contain an additional .modinfo section where additional information about the module is kept. Use modinfo command to see what kind of information it is.

```
debian@beaglebone:~/test/hello$ modinfo hello.ko
filename:       /home/debian/test/hello/hello.ko
depends:
name:           hello
vermagic:       4.14.108-ti-r113 SMP preempt mod_unload modversions ARMv7 p2v8
debian@beaglebone:~/test/hello$
```

How do modules get into - out from the kernel?

- **modprobe**
  - modprobe utility is used to add loadable modules to the Linux kernel. You can also view and remove modules using the modprobe command.
  - modprobe is an intelligent command, it looks for dependencies while loading a module. Suppose, if we load a module, which has symbols defined in some other module (*this module path is given inside the main module*). so, modprobe loads the main module and the dependent module.

- modprobe looks through the file /lib/modules/$(uname -r)/modules.dep, to see if other modules must be loaded before the requested module may be loaded. This file is created by depmod -a and contains module dependencies. The requested module has a dependency on another module if the other module defines symbols (variables or functions) that the requested module uses.
- Linux maintains /lib/modules/$(uname -r) directory for modules and its configuration files (except /etc/modprobe.conf and /etc/modprobe.d).
  - **To insert module using modprobe:**
    1. sudo ln -s /path/to/your-kernel-module.ko /lib/modules/`uname -r`/

    OR
    1. sudo cp /path/to/your-kernel-module.ko /lib/modules/`uname -r`/
    2. sudo depmod -a (*Generate a list of kernel module dependencies and associated map files*.)
    3. sudo modprobe your-kernel-module
  - **To remove module using modprobe:**
    - Sudo modprobe -r your-kernel-module

- **insmod**
- insmod is similar to modprobe: it can insert a module into the Linux kernel. Unlike modprobe, however, insmod does not read its modules from a set location, automatically insert them, and manage any dependencies.
- insmod can insert a single module from any location, and does not consider dependencies when doing so. It's a much lower-level program; in fact, it's the program modprobe uses to do the actual module insertion.
  - **To insert module using insmod:**
    - sudo insmod /path/to/your-kernel-module.ko
- Insmod requires you to pass it the full pathname and to insert the modules in the right order, while *modprobe* just takes the name, without any extension, and figures out all it needs to know by parsing /lib/modules/version/modules.dep.

- **rmmod**
- rmmod is a simple program which removes (unloads) a module from the Linux kernel. In most cases, you will want to use modprobe with the -r option instead, as it is more robust and handles dependencies for you.
  - **To insert module using insmod:**
    - sudo rmmod your-kernel-module.ko

To examine loaded kernel modules

- **lsmod**
- lsmod is a simple utility that does not accept any options or arguments. What the command does is that it reads /proc/modules and displays the file contents in a nicely formatted list.
- Run lsmod at the command line to find out what kernel modules are currently loaded. Below is a snapshot of our loaded module:

```
debian@beaglebone:~/test/hello$ lsmod
Module                  Size  Used by
hello           16384  0
```

```
evdev                   24576  1
8021q                   32768  0
garp              16384  1 8021q
mrp               20480  1 8021q
stp               16384  1 garp
llc               16384  2 garp,stp
usb_f_mass_storage    53248  2
usb_f_acm               16384  2
u_serial          20480  3 usb_f_acm
```

- Each line has three columns:
  - **Module -** The first column shows the name of the module.
  - **Size -** The second column shows the size of the module in bytes.
  - **Used by -** The third column shows a number that indicates how many instances of the module are currently used. A value of zero means that the module is not used. The comma-separated list after the number shows what is using the module.

- **Using /proc/modules**
- This file displays a list of all modules loaded into the kernel in an unarranged list.

```
debian@beaglebone:~/test/hello$ cat /proc/modules
hello 16384 0 - Live 0xbf1f9000 (PO)
evdev 24576 1 - Live 0xbf1e7000
8021q 32768 0 - Live 0xbf1d9000
garp 16384 1 8021q, Live 0xbf1d2000
mrp 20480 1 8021q, Live 0xbf1c9000
stp 16384 1 garp, Live 0xbf1c2000
llc 16384 2 garp,stp, Live 0xbf1b9000
usb_f_mass_storage 53248 2 - Live 0xbf1a4000
usb_f_acm 16384 2 - Live 0xbf19b000
u_serial 20480 3 usb_f_acm, Live 0xbf192000
```

- **~~init_module() / cleanup_module()~~ : Use your own init/deinit functions**
  - As of Linux 2.4, you can rename the init and cleanup functions of your modules; they no longer have to be called init_module() and cleanup_module() respectively.
  - This is done with the module_init() and module_exit() macros.These macros are defined in **linux/init.h**.
  - The only caveat is that your init and cleanup functions must be defined before calling the macros, otherwise you'll get compilation errors.

```
/**
 * hello-2.c : Demonstrating the module_init() and module_exit() macros.
 *  This is preferred over using init_module() and cleanup_module().
 */

#include <linux/module.h>      /* Needed by all modules */
#include <linux/kernel.h>      /* Needed for KERN_INFO */
#include <linux/init.h>        /* Needed for the macros */

static int __init hello_init(void)
{
        printk(KERN_INFO "All computers wait at the same speed !\n");
        return 0;
```

```
}

static void __exit hello_exit(void)
{
        printk(KERN_INFO "Any program that runs right is obsolete !\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- So now we have two real kernel modules under our belt. Adding another module is as simple as this:

```
obj-m += hello.o
obj-m += hello-2.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
```

- **The __init* and __exit* Macros**
  - The init* and exit* macros are widely used in the kernel. These macros are defined in include/linux/init.h and serve to free up kernel memory.
  - When you boot your kernel and see something like `Freeing unused kernel memory: 236k freed`, this is precisely what the kernel is freeing.

```
#define __init          __attribute__ ((__section__ (".init.text")))
#define __initdata      __attribute__ ((__section__ (".init.data")))
#define __exitdata      __attribute__ ((__section__(".exit.data")))
#define __exit_call     __attribute_used__ __attribute__ ((__section__ (".exitcall.exit")))

#ifdef MODULE
#define __exit          __attribute__ ((__section__(".exit.text")))
#else
#define __exit          __attribute_used__ __attribute__ ((__section__(".exit.text")))
#endif
```

- **__init* macros**
  - It tells the compiler to put the variable or the function in a special section, which is declared in vmlinux.lds.h. *init* puts the function in the ".init.text" section and *initdata* puts the data in the ".init.data" section.
  - For example, the following declaration means that the variable md_setup_ents will be put in the init data section.

```
static int helloworld_data __initdata;
```

  - But why must you use these macros ?
    Let's take an example, with the following function, defined in mm/slab.c :

```
void __init kmem_cache_init(void)
```

- This function initializes the slab system: it's only used once, at the boot of the kernel. So the code of this function should be freed from the memory after the first call. It's the goal of *free_initmem()*.
- The function *free_initmem()* will free the entire text and data init sections and so the code of your function, if it has been declared as init.

- **__exit* macros**
  - The __exit macro causes the omission of the function when the module is built into the kernel, and like __exit, has no effect for loadable modules. Again, if you consider when the cleanup function runs, This makes complete sense; built-in drivers don't need a cleanup function, while loadable modules do.
  - The exit macro tells the compiler to put the function in the ".exit.text" section. The exit_data macro tells the compiler to put the function in the ".exit.data" section.
  - exit.* sections make sense only for the modules : exit functions will never be called if compiled statically. That's why there is an ifdef : exit.* sections will be discarded only if modules support is disabled.

Licensing and Module Documentation

- In kernel 2.4 and later, a mechanism was devised to identify code licensed under the GPL (and friends) so people can be warned that the code is non open-source.
- This is accomplished by the MODULE_LICENSE() macro. By setting the license to GPL, you can keep the warning from being printed. This license mechanism is defined and documented in *linux/module.h*:
- Similarly, MODULE_DESCRIPTION() is used to describe what the module does, MODULE_AUTHOR() declares the module's author, and MODULE_SUPPORTED_DEVICE() declares what types of devices the module supports.
- These macros are all defined in linux/module.h and aren't used by the kernel itself. They're simply for documentation and can be viewed by a tool like objdump.

MODULE_INFO	Generic info of form tag = "info"
MODULE_ALIAS	For userspace: you can also call me...
MODULE_SOFTDEP	Soft module dependencies. See man modprobe.d for details.
MODULE_LICENSE	Indication for module license - Free(GPL, GLP v2, GPL and additional rights, Dual BSD/GPL, Dual MIT/GPL, Dual MPL/GPL)/Proprietary
MODULE_AUTHOR	Author(s), use "Name <email>" or just "Name", for multiple authors use multiple MODULE_AUTHOR() statements/lines.
MODULE_DESCRIPTION	What your module does

MODULE_VERSION	Version of form [<epoch>:]<version>[-<extra-version>] **<epoch>** : A (small) unsigned integer which allows you to start versions a new. If not mentioned, it's zero. eg. "2:1.0" is after "1:2.0". **<version>** : The <version> may contain only alphanumerics and the character `.'. Ordered by numeric sort for numeric parts, ascii sort for ascii parts. **<extraversion>** : Like <version>, but inserted for local customizations , eg "rh3" or "rusty1".
MODULE_DEVICE_TABLE	

- ## Modules Spanning Multiple Files
  - Sometimes it makes sense to divide a kernel module between several source files. Here's an example of such a kernel module.

```
/* 0103_module_spanning_init.c : Illustration of multi-file modules */

#include <linux/module.h>
#include <linux/kernel.h>

int __init module_split_init(void)
{
    printk(KERN_DEBUG "Welcome to the universe of linux!\n");
    return 0;
}

module_init(module_split_init);
```

```
/* 0103_module_spanning_exit.c : Illustration of multi-file modules */

#include <linux/module.h>
#include <linux/kernel.h>

void module_split_exit(void)
{
    printk(KERN_DEBUG "Remember! There is no exit once you entered!\n");
}

module_exit(module_split_exit);
```

```
/* Makefile */

obj-m += 0103_module_spanning.o
0103_module_spanning-objs := 0103_module_spanning_init.o 0103_module_spanning_exit.o

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

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

Passing a command line arguments to a module

- Modules can take command line arguments, but not with the argc/argv you might be used to.
- To allow arguments to be passed to your module, declare the variables that will take the values of the command line arguments as global and then use the module_param() macro, (defined in *linux/moduleparam.h*) to set the mechanism up.
- **module_param(name, type, perm);**
  - where, **name** is the name of both the parameter exposed to the user and the variable holding the parameter inside your module.
  - The **type** argument holds the parameter's data type; it is one of byte, short, ushort, int, uint, long, ulong, charp, bool, or invbool. These types are, respectively, a byte, a short integer, an unsigned short integer, an integer, an unsigned integer, a long integer, an unsigned long integer, a pointer to a char, a Boolean, and a Boolean whose value is inverted from what the user specifies.
  - The byte type is stored in a single char and the Boolean types are stored in variables of type int. The rest are stored in the corresponding primitive C types.
  - Finally, the **perm** argument specifies the permissions of the corresponding file in sysfs.
  - The permissions can be specified in the usual octal format, for example 0644 (owner can read and write, group can read, everyone else can read), or by ORing together the usual S_I* defines, for example S_IRUGO | S_IWUSR (everyone can read, a user can also write). (*defined in include/linux/stat.h*)
  - The macro does not declare the variable for you. You must do that before using the macro. Therefore, typical use might resemble

```
/* module parameter controlling the capability to allow live bait on the pole */
static int allow_live_bait = 1;             /* default to on */
module_param(allow_live_bait, bool, 0644);  /* a Boolean type */
```

  - This would be in the outermost scope of your module's source file. In other words, allow_live_bait is global.
- **module_param_named(name, variable, type, perm);**
  - It is possible to have the internal variable named differently than the external parameter. This is accomplished via *module_param_named()*.
  - where **name** is the externally viewable parameter name and **variable** is the name of the internal global variable. For example,

```
static unsigned int max_test = DEFAULT_MAX_LINE_TEST;
module_param_named(maximum_line_test, max_test, int, 0);
```

- **module_param_string(name, string, len, perm);**
  - Normally, you would use a type of *charp* to define a module parameter that takes a string. The kernel copies the string provided by the user into memory and points your variable to the string. For example,

```
static char *name;
module_param(name, charp, 0);
```

  - If so desired, it is also possible to have the kernel copy the string directly into a character array that you supply. This is done via *module_param_string()*.

- where, **name** is the external parameter name, **string** is the internal variable name, **len** is the size of the buffer named by string (or some smaller size, but that does not make much sense), and **perm** is the sysfs permissions (or zero to disable a sysfs entry altogether). For example,

```
static char species[BUF_LEN];
module_param_string(specifies, species, BUF_LEN, 0);
```

- **module_param_array(name, type, nump, perm);**
  - You can accept a comma-separated list of parameters that are stored in a C array via *module_param_array()*.
  - where, **name** is again the external parameter and internal variable name, **type** is the data type, and **perm** is the sysfs permissions. The new argument, **nump**, is a pointer to an integer where the kernel will store the number of entries stored into the array.
  - Note that the array pointed to by name must be statically allocated. The kernel determines the array's size at compile-time and ensures that it does not cause an overrun. Use is simple. For example,

```
static int fish[MAX_FISH];
static int nr_fish;
module_param_array(fish, int, &nr_fish, 0444);
```

- **module_param_array_named(name, array, type, nump, perm);**
  - You can name the internal array something different than the external parameter with *module_param_array_named().*
  - The parameters are identical to the other macros.
- **MODULE_PARM_DESC(name, description)**
  - Finally, you can document your parameters by using MODULE_PARM_DESC():

```
static unsigned short size = 1;
module_param(size, ushort, 0644);
MODULE_PARM_DESC(size, "The size in inches of the fishing pole connected to this computer.");
```

- All these macros require the inclusion of <linux/moduleparam.h>.
- At runtime, insmod will fill the variables with any command line arguments that are given, like ./insmod mymodule.ko myvariable=5. The variable declarations and macros should be placed at the beginning of the module for clarity.
- A good use for this is to have the module variable's default values set, like a port or IO address. If the variables contain the default values, then perform auto-detection. Otherwise, keep the current value.

```
/* 0104_command_param.c : Demonstrates command line argument passing to a module */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/stat.h>
#include <linux/init.h>

#define COLORS_NAMES_LEN    40
#define ARRAY_LEN           5

static short int cmd_short = 10;
```

```
static int cmd_int = 100;
static long int cmd_long = 10000;
static char* cmd_charp = "Avengers! assemble";
static char cmd_string[COLORS_NAMES_LEN] = "red, green, blue";
static int cmd_int_array[ARRAY_LEN] = {1,2,3,4,5};
static int cmd_int_array_idx = 0;
static char cmd_char_array[ARRAY_LEN] = {'a', 'b', 'c', 'd', 'e'};
static int cmd_char_array_idx = 0;

/* user and group can read/write */
module_param(cmd_short, short, 0660);
MODULE_PARM_DESC(cmd_short, "A short integer number");
/* user and group can read/write */
module_param_named(lucky_number, cmd_int, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(lucky_number, "A lucky number");
/* user and group can read only */
module_param(cmd_long, long, 0);
MODULE_PARM_DESC(cmd_long, "A readonly long number");
/* anyone can (user,group,others) read only */
module_param(cmd_charp, charp, S_IRUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(cmd_charp, "A readonly string");
/* only user can read and write */
module_param_string(fav_colors, cmd_string, COLORS_NAMES_LEN, 0600);
MODULE_PARM_DESC(fav_colors, "Favourite colors");
/* only user can read */
module_param_array(cmd_int_array, int, &cmd_int_array_idx, 0640);
MODULE_PARM_DESC(cmd_int_array, "An int array of 5");
/* only user can read and write */
module_param_array_named(fav_letters, cmd_char_array, byte, &cmd_char_array_idx, 0600);
MODULE_PARM_DESC(fav_letters, "5 favourite letters");

int __init cmdline_demo_init(void)
{
    int i = 0;
    printk(KERN_INFO "command line demo init\n");
    printk(KERN_INFO "===============================\n");
    printk(KERN_INFO "=> cmd_short : %hd\n", cmd_short);
    printk(KERN_INFO "=> lucky_number : %d\n", cmd_int);
    printk(KERN_INFO "=> cmd_long : %ld\n", cmd_long);
    printk(KERN_INFO "=> cmd_charp : %s\n", cmd_charp);
    printk(KERN_INFO "=> fav_colors : %s\n", cmd_string);
    for(i = 0; i < ARRAY_LEN; i++)
    {
        printk(KERN_INFO "=> cmd_int_array[%d] : %d\n", i, cmd_int_array[i]);
    }
    for(i = 0; i < ARRAY_LEN; i++)
    {
        printk(KERN_INFO "=> faourite letter[%d] : %c\n", i, cmd_char_array[i]);
    }
    printk(KERN_INFO "===============================\n");
    return 0;
}

void __exit cmdline_demo_exit(void)
{
    printk(KERN_INFO "command line demo exit\n");
}

module_init(cmdline_demo_init);
module_exit(cmdline_demo_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sunil Vaghela");
```

```
debian@beaglebone:~/0104-cmdline-params$ sudo insmod 0104_cmdline_param.ko
debian@beaglebone:~/linux-device-drivers/0104-cmdline-params$ dmesg | tail -n 18
[ 2294.865925] command line demo init
[ 2294.865941] ================================
[ 2294.865950] => cmd_short : 10
[ 2294.865955] => lucky_number : 100
[ 2294.865960] => cmd_long : 10000
[ 2294.865965] => cmd_charp : Avengers! assemble
[ 2294.865970] => fav_colors : red, green, blue
[ 2294.865976] => cmd_int_array[0] : 1
[ 2294.865981] => cmd_int_array[1] : 2
[ 2294.865986] => cmd_int_array[2] : 3
[ 2294.865990] => cmd_int_array[3] : 4
[ 2294.865995] => cmd_int_array[4] : 5
[ 2294.866001] => faourite letter[0] : a
[ 2294.866006] => faourite letter[1] : b
[ 2294.866010] => faourite letter[2] : c
[ 2294.866015] => faourite letter[3] : d
[ 2294.866019] => faourite letter[4] : e
[ 2294.866024] ================================
debian@beaglebone:~/0104-cmdline-params$ dmesg | tail -n 5
[ 2294.866010] => faourite letter[2] : c
[ 2294.866015] => faourite letter[3] : d
[ 2294.866019] => faourite letter[4] : e
[ 2294.866024] ================================
[ 2299.019158] command line demo exit
debian@beaglebone:~/0104-cmdline-params$ sudo insmod 0104_cmdline_param.ko cmd_short=10
lucky_number=456 cmd_long=47 cmd_charp='"Marvel vs DC"' fav_colors='"yellow black white"'
cmd_int_array=10,20,3,40,50 fav_letters=0x41,0x42,0x43,0x44,0x45
debian@beaglebone:~/linux-device-drivers/0104-cmdline-params$ dmesg | tail -n 18
[ 4493.585918] command line demo init
[ 4493.585932] ================================
[ 4493.585942] => cmd_short : 10
[ 4493.585948] => lucky_number : 456
[ 4493.585953] => cmd_long : 47
[ 4493.585958] => cmd_charp : Marvel vs DC
[ 4493.585963] => fav_colors : yellow black white
[ 4493.585969] => cmd_int_array[0] : 10
[ 4493.585974] => cmd_int_array[1] : 20
[ 4493.585979] => cmd_int_array[2] : 3
[ 4493.585983] => cmd_int_array[3] : 40
[ 4493.585988] => cmd_int_array[4] : 50
[ 4493.585994] => faourite letter[0] : A
[ 4493.585999] => faourite letter[1] : B
[ 4493.586004] => faourite letter[2] : C
[ 4493.586008] => faourite letter[3] : D
[ 4493.586013] => faourite letter[4] : E
[ 4493.586017] ================================
```

- while passing a string with space, you have to put the string between '""', else only the first word will be assigned to the module parameter and words after space will be discarded.
- Also notice how the int and byte array has passed to the module.
- Value of the readonly variable also changed - why ?

- ## Modules vs Programs
  - ### How modules begin and end
    - A program usually begins with a *main()* function, executes a bunch of instructions and terminates upon completion of those instructions.
    - Kernel modules work a bit differently. A module always begins with either the *init_module* or the function you specify with the *module_init* call. This is the entry function for modules; it tells the kernel what functionality the module provides and sets up the kernel to run the module's functions when they're needed. Once it does this, the entry function returns and the module does nothing until the kernel wants to do something with the code that the module provides.
    - All modules end by calling either cleanup_module or the function you specify with the module_exit call. This is the exit function for modules; it undoes whatever entry function did. It unregisters the functionality that the entry function registered.
  - ### Functions available to modules
    - Programmers use functions they don't define all the time. A prime example of this is *printf().* You use these library functions which are provided by the standard C library, libc. The definitions for these functions don't actually enter your program until the linking stage, which ensures that the code (*for printf() for example*) is available, and fixes the call instruction to point to that code.
    - Kernel modules are different here, too. In the previous all examples, you might have noticed that we used a function, *printk()* but didn't include a standard I/O library. That's because modules are object files whose symbols get resolved upon insmod'ing.
    - The definition for the symbols comes from the kernel itself; the only external functions you can use are the ones provided by the kernel. If you're curious about what symbols have been exported by your kernel, take a look at /proc/kallsyms.
    - One point to keep in mind is *the difference between library functions and system calls*. Library functions are higher level, run completely in user space and provide a more convenient interface for the programmer to the functions that do the real -- *system calls*.
    - *System calls* run in kernel mode on the user's behalf and are provided by the kernel itself. The library function *printf()* may look like a very general printing function, but all it really does is format the data into strings and write the string data using the low-level system call *write()*, which then sends the data to standard output.
    - Would you like to see what system calls are made by *printf()*? It's easy! Compile the following program:

```
/* hello.c - Demonstrate printf -> write connection using strace */
#include <stdio.h>
int main(void)
{
printf("hello");
return 0;
}
```

- with **gcc -Wall -o hello hello.c**. Run the executable with **strace ./hello**.

```
sunil@sunil-Inspiron-N4050:Desktop ⚡ strace ./hello
execve("./hello", ["./hello"], 0xbffe6aa0 /* 59 vars */) = 0
brk(NULL)                               = 0x1905000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7f03000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=117247, ...}) = 0
mmap2(NULL, 117247, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7ee6000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\20\220\1\0004\0\0\0"..., 512) =
512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1942840, ...}) = 0
mmap2(NULL, 1948188, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb7d0a000
mprotect(0xb7edf000, 4096, PROT_NONE)   = 0
mmap2(0xb7ee0000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1d5000) = 0xb7ee0000
mmap2(0xb7ee3000, 10780, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7ee3000
close(3)                                = 0
set_thread_area({entry_number=-1, base_addr=0xb7f040c0, limit=0x0fffff, seg_32bit=1,
contents=0, read_exec_only=0, limit_in_pages=1, seg_not_present=0, useable=1}) = 0
(entry_number=6)
mprotect(0xb7ee0000, 8192, PROT_READ)   = 0
mprotect(0x475000, 4096, PROT_READ)          = 0
mprotect(0xb7f30000, 4096, PROT_READ)   = 0
munmap(0xb7ee6000, 117247)              = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
brk(NULL)                               = 0x1905000
brk(0x1926000)                          = 0x1926000
brk(0x1927000)                          = 0x1927000
write(1, "hello", 5hello)                       = 5
exit_group(0)                           = ?
+++ exited with 0 +++
```

- Are you impressed? Every line you see corresponds to a system call. *strace* is a handy program that gives you details about what a system calls a program is making, including which call is made, what its arguments are and what it returns. It's an invaluable tool for figuring out things like what files a program is trying to access. The highlighted line is the face behind the printf() mask.
- You may not be familiar with *write*, since most people use library functions for file I/O (like fopen, fputs, fclose). If that's the case, try looking at *man 2 write*. The 2nd man section is devoted to system calls (*like kill() and read()*). The 3rd man section is devoted to library calls.

- **User Space vs Kernel Space**
  - A kernel is all about access to resources, whether the resource in question happens to be a video card, a hard drive or even memory. Programs often compete for the same resource. As I just saved this document, *updatedb* started updating the locate database.

- My vim session and updatedb are both using the hard drive concurrently. The kernel needs to keep things orderly, and not give users access to resources whenever they feel like it.
- To this end, a CPU can run in different modes. Each mode gives a different level of freedom to do what you want on the system. The Intel 80386 architecture has 4 of these modes, which are called *rings*. Unix uses only two rings; the highest ring (ring 0, also known as `supervisor mode' where everything is allowed to happen) and the lowest ring, which is called `user mode'.
- Recall the discussion about library functions vs system calls. Typically, you use a library function in user mode. The library function calls one or more system calls, and these system calls execute on the library function's behalf, but do so in supervisor mode since they are part of the kernel itself. Once the system call completes its task, it returns and execution gets transferred back to user mode.

- **Name space**
  - When you write a small C program, you use variables which are convenient and make sense to the reader. If, on the other hand, you're writing routines which will be part of a bigger problem, any global variables you have are part of a community of other peoples' global variables; some of the variable names can clash.
  - When a program has lots of global variables which aren't meaningful enough to be distinguished, you get *namespace pollution*. In large projects, effort must be made to remember reserved names, and to find ways to develop a scheme for naming unique variable names and symbols.
  - When writing kernel code, even the smallest module will be linked against the entire kernel, so this is definitely an issue. The best way to deal with this is to declare all your variables as static and to use a well-defined prefix for your symbols. By convention, all kernel prefixes are lowercase. If you don't want to declare everything as static, another option is to declare a symbol table and register it with a kernel. We'll get to this later.
  - The file /proc/kallsyms holds all the symbols that the kernel knows about and which are therefore accessible to your modules since they share the kernel's codespace.
- **Code space**
  - If you haven't thought about what a segfault really means, you may be surprised to hear that pointers don't actually point to memory locations. Not real ones, anyway.
  - When a process is created, the kernel sets aside a portion of real physical memory and hands it to the process to use for its executing code, variables, stack, heap and other things. This memory begins with 0x00000000 and extends up to whatever it needs to be.
  - Since the memory space for any two processes don't overlap, every process that can access a memory address, say 0xbffff978, would be accessing a different location in real physical memory! The processes would be accessing an index named 0xbffff978 which points to some kind of offset into the region of memory set aside for that particular process.
- The kernel has its own space of memory as well. Since a module is code which **can** be dynamically inserted and removed in the kernel, it shares the kernel's

codespace rather than having its own. Therefore, if your module segfaults, the kernel segfaults. And if you start writing over data because of an off-by-one error, then you're trampling on kernel data (or code). This is even worse than it sounds, so try your best to be careful.

- **Device-Drivers**
  - One class of module is the device driver, which provides functionality for hardware like a TV card or a serial port. On unix, each piece of hardware is represented by a file located in **/dev** named a device file which provides the means to communicate with the hardware.
  - The device driver provides the communication on behalf of a user program. e.g the es1370.o sound card device driver might connect the */dev/sound* device file to the Ensoniq IS1370 sound card. A userspace program like VLC can use /dev/sound without ever knowing what kind of sound card is installed.
  - **Major and Minor Numbers**
    - The kernel needs to be told how to access the device. This is accomplished by the major number and the minor number of that device.
    - Let’s look at some device files. Here are device files which represent the first five partitions of the hard disk or SCSI - Small Computer System Interface (*pronounced "skuzzy"*) disk drive:

```
sunil@sunil-Inspiron-N4050:~ ⚡ ls -l /dev/sda[1-5]
brw-rw---- 1 root disk 8, 1 Apr 11 14:01 /dev/sda1
brw-rw---- 1 root disk 8, 2 Apr 11 14:01 /dev/sda2
brw-rw---- 1 root disk 8, 3 Apr 11 14:01 /dev/sda3
brw-rw---- 1 root disk 8, 4 Apr 11 14:01 /dev/sda4
brw-rw---- 1 root disk 8, 5 Apr 11 14:01 /dev/sda5
```

    - Notice the column of numbers separated by a comma? The first number is called the device's major number. The second number is the minor number. The major number tells you which driver is used to access the hardware.
    - Each driver is assigned a unique major number; all device files with the same major number are controlled by the same driver. All the above major numbers are 11, because they're all controlled by the same driver.
    - The minor number is used by the driver to distinguish between the various hardware it controls. Returning to the example above, although all five devices are handled by the same driver they have unique minor numbers because the driver sees them as being different pieces of hardware.
    - **Devices are divided into two types: character devices and block devices.** The difference is that block devices have a buffer for requests, so they can choose the best order in which to respond to the requests. This is important in the case of storage devices, where it's faster to read or write sectors which are close to each other, rather than those which are further apart. Another difference is that block devices can only accept input and return output in blocks (*whose*

size can vary according to the device), whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character, because they don't need this type of buffering, and they don't operate with a fixed block size.

- You can tell whether a device file is for a block device or a character device by looking at the first character in the output of **ls -l**. If it's `b' then it's a block device, and if it's 'c' then it's a character device. The devices you see above are block devices.
- Here are some character devices:

```
sunil@sunil-Inspiron-N4050:~ ⚡ ls -l /dev/ttyS0 /dev/ttyUSB0 /dev/i2c-0 /dev/media0
crw------- 1  root root   89,  0 Apr 11 14:01 /dev/i2c-0
crw-rw---- 1 root video   243,  0 Apr 11 14:01 /dev/media0
crw-rw---- 1 root dialout   4, 64 Apr 11 14:01 /dev/ttyS0
crw-rw---- 1 root dialout 188,  0 Apr 11 18:56 /dev/ttyUSB0
```

- If you want to see which major numbers have been assigned, you can look at [Documentation/admin-guide/devices.txt]().
- When the system was installed, all of those device files were created by the **mknod** command. To create a new char device named `coffee' with major/minor number 12 and 2, simply do **mknod /dev/coffee c 12 2**.
- You don't *have* to put your device files into /dev, but it's done by convention. Linus put his device files in /dev, and so should you. However, when creating a device file for testing purposes, it's probably OK to place it in your working directory where you compile the kernel module. Just be sure to put it in the right place when you're done writing the device driver.
- **When a device file is accessed, the kernel uses the major number of the file to determine which driver should be used to handle the access. This means that the kernel doesn't really need to use or even know about the minor number. The driver itself is the only thing that cares about the minor number. It uses the minor number to distinguish between different pieces of hardware.**

- **Dynamic allocation of Major numbers**
  - Some major device numbers are statically assigned to the most common devices. A list of those devices can be found in *Documentation/devices.txt* within the kernel source tree. The chances of a static number having already been assigned for the use of your new driver are small, however, and new numbers are not being assigned. So, as a driver writer, you have a choice: you can simply pick a number that appears to be unused, or you can allocate major numbers in a dynamic manner.
  - Picking a number may work as long as the only user of your driver is you; once your driver is more widely deployed, a randomly picked major number will lead to conflicts and trouble.
  - Thus, for new drivers, we strongly suggest that you use dynamic allocation to obtain your major device number, rather than choosing a number randomly from the ones that are currently free. In other words, your drivers should almost certainly be using *alloc_chrdev_region* rather than *register_chrdev_region*.

- The disadvantage of dynamic assignment is that you can't create the device nodes in advance, because the major number assigned to your module will vary. For normal use of the driver, this is hardly a problem, because once the number has been assigned, you can read it from */proc/devices*.

- **Character Device Files**
  - **The file_operations Structure**
    - The file_operations structure is defined in *include/linux/fs.h*, and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation.
    - For example, every character driver needs to define a function that reads from the device. The file_operations structure holds the address of the module's function that performs that operation. Here is what the definition looks like for kernel 5.0.3:

```
struct file_operations {
	struct module *owner;
	loff_t (*llseek) (struct file *, loff_t, int);
	ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
	ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
	ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
	ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
	int (*iterate) (struct file *, struct dir_context *);
	int (*iterate_shared) (struct file *, struct dir_context *);
	__poll_t (*poll) (struct file *, struct poll_table_struct *);
	long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
	long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
	int (*mmap) (struct file *, struct vm_area_struct *);
	unsigned long mmap_supported_flags;
	int (*open) (struct inode *, struct file *);
	int (*flush) (struct file *, fl_owner_t id);
	int (*release) (struct inode *, struct file *);
	int (*fsync) (struct file *, loff_t, loff_t, int datasync);
	int (*fasync) (int, struct file *, int);
	int (*lock) (struct file *, int, struct file_lock *);
	ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
	unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned
long, unsigned long);
	int (*check_flags)(int);
	int (*flock) (struct file *, int, struct file_lock *);
	ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
	ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
	int (*setlease)(struct file *, long, struct file_lock **, void **);
	long (*fallocate)(struct file *file, int mode, loff_t offset,
	loff_t len);
	void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
	unsigned (*mmap_capabilities)(struct file *);
#endif
	ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
	loff_t, size_t, unsigned int);
	loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
```

```
                    struct file *file_out, loff_t pos_out,
                    loff_t len, unsigned int remap_flags);
        int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

- It can be noticed that the signature of the functions differs from the system call that the user uses. The operating system sits between the user and the device driver to simplify implementation in the device driver.
- **open** does not receive the parameter path or the various parameters that control the file opening mode. Similarly, **read, write, release, ioctl, lseek** do not receive as a parameter a file descriptor. Instead, these routines receive as parameters two structures: file and inode. Both structures represent a file, but from different perspectives.
- **Most** parameters for the presented operations have a direct meaning:
  - *file* and inode identifies the device type file;
  - *size* is the number of bytes to be read or written;
  - *offset* is the displacement to be read or written (to be updated accordingly);
  - *user_buffer* user buffer from which it reads / writes;
  - *whence* is the way to seek (the position where the search operation starts);
  - *cmd* and *arg* are the parameters sent by the users to the ioctl call (IO control).
- Some operations are not implemented by a driver. For example, a driver that handles a video card won't need to read from a directory structure. **The corresponding entries in the file_operations structure should be set to NULL.**
- There is a gcc extension that makes assigning to this structure more convenient. You'll see it in modern drivers, and may catch you by surprise. This is what the new way of assigning to the structure looks like:

```
struct file_operations fops = {
        read: device_read,
        write: device_write,
        open: device_open,
        release: device_release
};
```

- However, there's also a C99 way of assigning to elements of a structure, and this is definitely preferred over using the GNU extension. The version of gcc the author used when writing this, 2.95, supports the new C99 syntax. You should use this syntax in case someone wants to port your driver. It will help with compatibility:

```
struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};
```

- The meaning is clear, and you should be aware that any member of the structure which you don't explicitly assign will be initialized to NULL by gcc. An instance of struct *file_operations* containing pointers to functions that are used to implement read, write, open... syscalls is commonly named **fops**.

- **The file structure**
  - Each device is represented in the kernel by a file structure, which is defined in include/linux/fs.h. Be aware that a file is a kernel level structure and never appears in a user space program.
  - It's not the same thing as a FILE, which is defined by glibc and would never appear in a kernel space function. Also, its name is a bit misleading; it represents an abstract open `file', not a file on a disk, which is represented by a structure named inode.
  - An instance of a struct file is commonly named *filp*. You'll also see it referred to as *struct file file*. Resist the temptation.

```
struct file {
        union {
        struct llist_node   fu_llist;
        struct rcu_head           fu_rcuhead;
        } f_u;
        struct path       f_path;
        struct inode      *f_inode;   /* cached value */
        const struct file_operations       *f_op;

        /*
        * Protects f_ep_links, f_flags.
        * Must not be taken from IRQ context.
        */
        spinlock_t        f_lock;
        enum rw_hint              f_write_hint;
        atomic_long_t             f_count;
        unsigned int      f_flags;
        fmode_t           f_mode;
        struct mutex      f_pos_lock;
        loff_t            f_pos;
        struct fown_struct  f_owner;
        const struct cred   *f_cred;
        struct file_ra_state       f_ra;

        u64       f_version;
#ifdef CONFIG_SECURITY
        void                *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to this file */
        struct list_head f_ep_links;
        struct list_head f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
        errseq_t          f_wb_err;
} __randomize_layout
  __attribute__((aligned(4)));  /* lest something weird decides that 2 is OK */
```

- **Registering / Unregistering a Device**
  - As discussed earlier, char devices are accessed through device files, usually located in **/dev**. The major number tells you which driver handles which device

file. The minor number is used only by the driver itself to differentiate which device it's operating on, just in case the driver handles more than one device.

- Adding a driver to your system means registering it with the kernel. This is synonymous with assigning it a major number during the module's initialization.
- Before kernel 2.6, we were using *register_chrdev()* to allocate and register the file operations structure. But this method is history now.
- **Why don't we use register_chrdev()?**
  - *register_chrdev()* still works in kernel 2.6, but the problem in kernel 2.4 was, we have limited major number allocation. We had only 8-bit for both major and minor numbers.
  - So, in total we could have only 255 major or minor numbers. Whereas in the new methods, we have 32 bit for major and minor numbers. (*12 bits for major, 20 bits for minor*).
- Below are the type and type/functions introduced in kernel 2.6

1. dev_t

- *dev_t* is the data type introduced in kernel 2.6. It's 32 bit, first 12 bits holds the major number and remaining 20 bits holds the minor number.

  Example:

  dev_t dev;
  dev = 0x0F800001 (*For Major number = 248 and Minor number = 1*)
  MAJOR(dev) = 0x0F800001 >> 20 = 0x0F8 = 248
  MINOR(dev) = 0x0F800001 & 0x000FFFFF = 0x1 = 1
- *Available* macros for translation into *dev_t* :

  a. MKDEV(int major, int minor);

  - Given two integers - major and minor numbers, MKDEV combines them into one 32 bit dev_t compatible number.

  b. MAJOR(dev_t dev);

  - The macro MAJOR accepts a dev_t type number which is 32 bits, and returns a major number

  c. MINOR(dev_t dev);

  - The macro MINOR accepts a dev_t type number which is 32 bits, and returns a minor number

```
/* linux-src/include/linux/kdev_t.h */

#define MINORBITS   20
#define MINORMASK   ((1U << MINORBITS) - 1)

#define MAJOR(dev)  ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)  ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)(((ma) << MINORBITS) | (mi))
```

2. int register_chrdev_region (dev_t first, unsigned count, const char * name);

- One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. The necessary function for this task is *register_chrdev_region*, which is declared in <linux/fs.h>.

- Here, *first* is the beginning device number of the range you would like to allocate. The minor number portion of first is often 0, but there is no requirement to that effect.
- *count* is the total number of contiguous device numbers you are requesting. Note that, if count is large, the range you request could spill over to the next major number; but everything will still work properly as long as the number range you request is available.
- Finally, *name* is the name of the device that should be associated with this number range; it will appear in */proc/devices and sysfs*.
- As with most kernel functions, the return value from register_chrdev_region will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region.
- *register_chrdev_region* works well if you know ahead of time exactly which device numbers you want. Often, however, you will not know which major numbers your device will use; there is a constant effort within the Linux kernel development community to move over to the use of dynamically-allocated device numbers.
- The kernel will happily allocate a major number for you on the fly, but you must request this allocation by using a different function - alloc_chrdev_region().

3. **int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);**
   - With this function, *dev* is an output-only parameter that will, on successful completion, hold the first number in your allocated range.
   - *firstminor* should be the requested first minor number to use; it is usually 0. The *count* and *name* parameters work like those given to *request_chrdev_region()*.
4. **void unregister_chrdev_region(dev_t first, unsigned int count);**
   - Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with - *unregister_chrdev_region()*.
   - The usual place to call *unregister_chrdev_region* would be in your module's cleanup function.
   - We can't allow the kernel module to be rmmod'ed whenever root feels like it. If the device file is opened by a process and then we remove the kernel module, using the file would cause a call to the memory location where the appropriate function (read/write) used to be.
   - If we're lucky, no other code was loaded there, and we'll get an ugly error message. If we're unlucky, another kernel module was loaded into the same location, which means a jump into the middle of another function within the kernel. The results of this would be impossible to predict, but they can't be very positive.
   - Normally, when you don't want to allow something, you return an error code (a negative number) from the function which is supposed to do it. With *cleanup_module* that's impossible because it's a void function.

- However, there's a counter which keeps track of how many processes are using your module. You can see what it's value is by looking at the 3rd field of /proc/modules. If this number isn't zero, rmmod will fail.
- Note that you don't have to check the counter from within *cleanup_module* because the check will be performed for you by the system call *sys_delete_module*, defined in linux/module.c. You shouldn't use this counter directly, but there are functions defined in *linux/module.h,* which let you increase, decrease and display this counter:
    a. try_module_get(THIS_MODULE): Increment the use count.
    b. module_put(THIS_MODULE): Decrement the use count.
- It's important to keep the counter accurate; if you ever do lose track of the correct usage count, you'll never be able to unload the module; it's now reboot time, boys and girls. This is bound to happen to you sooner or later during a module's development.
- *The above functions allocate device numbers for your driver's use, but they do not tell the kernel anything about what you will actually do with those numbers. Before a user-space program can access one of those device numbers, your driver needs to connect them to its internal functions that implement the device's operations.*

5. cdev

- The association of device numbers with specific devices happens by way of the *cdev* structure, found in <linux/cdev.h>.
- *cdev* is newly introduced in kernel 2.6. This cdev structure is an internal representation of a char device.
- It has a member field as a struct file operations pointer. You have to create and populate your file operations structure and assign the address of that structure to this member.

```
/* <kernel-src>/include/kernel/cdev.h */

struct cdev {
        struct kobject kobj;
        struct module *owner;
        const struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
} __randomize_layout;
```

6. struct cdev *cdev_alloc(void);

- Allocates and returns a cdev structure, or NULL on failure. Then, you need to give *an ops* pointer.
  Example:

```
struct cdev *my_dev = cdev_alloc();

 if (my_dev != NULL)
     my_dev->ops = &my_fops;  /* The file_operations structure */
     my_dev->owner = THIS_MODULE;
 else
     /* No memory, we lose */
```

- The owner field of the structure should be initialized to THIS_MODULE to protect against ill-advised module unloads while the device is active.

7. **void cdev_init(struct cdev *cdev, struct file_operations *fops);**
   - In the more common usage pattern, however, the *cdev* structure will be embedded within some larger, device-specific structure, and it will be allocated with that structure. In this case, the function to initialize the cdev is *cdev_init().*
   - Initializes *cdev*, remembering fops, making it ready to add to the system with *cdev_add*.

     Example:
     ```
     struct mycdev {
             struct cdev cdev;
             int  flag;
     };
     mycdev f ;
     f= kmalloc(100);
     cdev_init(f.cdev, &fops);
     ```
8. **int cdev_add(struct cdev *cdev, dev_t dev, unsigned int count);**
   - Once you have the structure set up, it's time to add it to the system using *cdev_add.*
   - *cdev_add* adds the device represented by *dev* to the system, making it live immediately, and can be used by the user.
   - *cdev* is, of course, a pointer to the cdev structure; *dev* is the first device number handled by this structure, and *count* is the number of devices it implements.
9. **void cdev_del(struct cdev *cdev);**
   - *cdev_del* removes *cdev* from the system. This function should only be called on a cdev structure, which has been successfully added to the system with *cdev_add()*.
   - If you need to destroy a structure which has not been added in this way (*perhaps cdev_add() failed*), you must, instead, manually decrement the reference count in the structure's kobject with a call like:
     ```
     kobject_put(&cdev->kobj);
     ```
   - Calling a *cdev_del()* on a device which is still active (*if, say, a user-space process still has an open file reference to it*) will cause the device to become inaccessible, but it will not actually delete the structure at that time.
   - The reference count in the structure will keep it around until all the references have gone away. That means that your driver's methods could be called after you have deleted your cdev object - a possibility you should be aware of.
   - The reference count of a cdev structure can be manipulated with:
     ```
     struct kobject *cdev_get(struct cdev *cdev);
     void cdev_put(struct cdev *cdev);
     ```
   - Note that these functions change two reference counts: that of the cdev structure, and that of the module which owns it. It will be rare for drivers to call these functions, however.

Questions/Answers:

1. **What is Modversioning ?**

- A module compiled for one kernel won't load if you boot a different kernel unless you enable CONFIG_MODVERSIONS in the kernel.

2. **Difference between dmesg output and /var/log/messages**

- We can say that dmesg is the subset of /var/log/messages and is maintained in a ring buffer.
- This information is also sent in real time to syslogd or klogd, when they are running, and ends up in /var/log/messages; when dmesg is most useful is in capturing boot-time messages from before syslogd and/or klogd started, so that they will be properly logged.
- The /var/log/messages contains global system messages, including the messages that are logged during system startup. There are several things that are logged in /var/log/messages including mail, cron, daemon, kern, auth, etc.

3. **Kbuild goal definitions**

- Goal definitions are the main part (heart) of the kbuild Makefile.These lines define the files to be built, any special compilation options, and any subdirectories to be entered recursively. The most simple kbuild makefile contains one line:
  Example: obj-y += hello.o
  This tells kbuild that there is one object in that directory, named hello.o. hello.o will be built from hello.c or hello.S.
- If foo.o shall be built as a module, the variable obj-m is used. Therefore the following pattern is often used:
  Example: obj-$(CONFIG_HELLO) += hello.o
  $(CONFIG_HELLO) evaluates to either y (for built-in) or m (for module). If CONFIG_HELLO is neither y nor m, then the file will not be compiled nor linked.

Important files in kernel:

include/linux/module.h	Required by all modules, module_* macros,
include/linux/kernel.h	Kernel debug macros and many more
include/linux/kern_levels.h	Kernel log levels macros
include/linux/printk.h	Kernel log level alias function
include/linux/init.h	Declarations of init* and exit* macros
include/asm-generic/vmlinux.lds.h	Declaration of init.data, exit.data etc sections
include/linux/moduleparam.h	Declarations of module_param* macros required to pass command line arguments to a module
include/linux/stat.h	S_IWUGO, S_IRUGO, S_IXUGO... etc permissions needed in module_param()
/proc/kallsyms	All symbols exported by the kernel
Documentation/admin-guide/devices.txt	To see which major numbers have been assigned
include/linux/fs.h	File_operations structure
include/linux/kdev_t.h	MAJOR, MINOR and MKDEV macros
include/linux/cdev.h	Character device init/add/del, struct cdev declaration

TODO:

- Add syslog and logd tutorial
- Add all links tutorial in the pdf itself and give its link
- Kernel makefile tutorial
- Explore more in include/asm-generic/vmlinux.lds.h --> .init, .initdata and .exit sections
- MODULE_DEVICE_TABLE explanation
- Go through Major/Minor numbers document (Documentation/admin-guide/devices.txt)