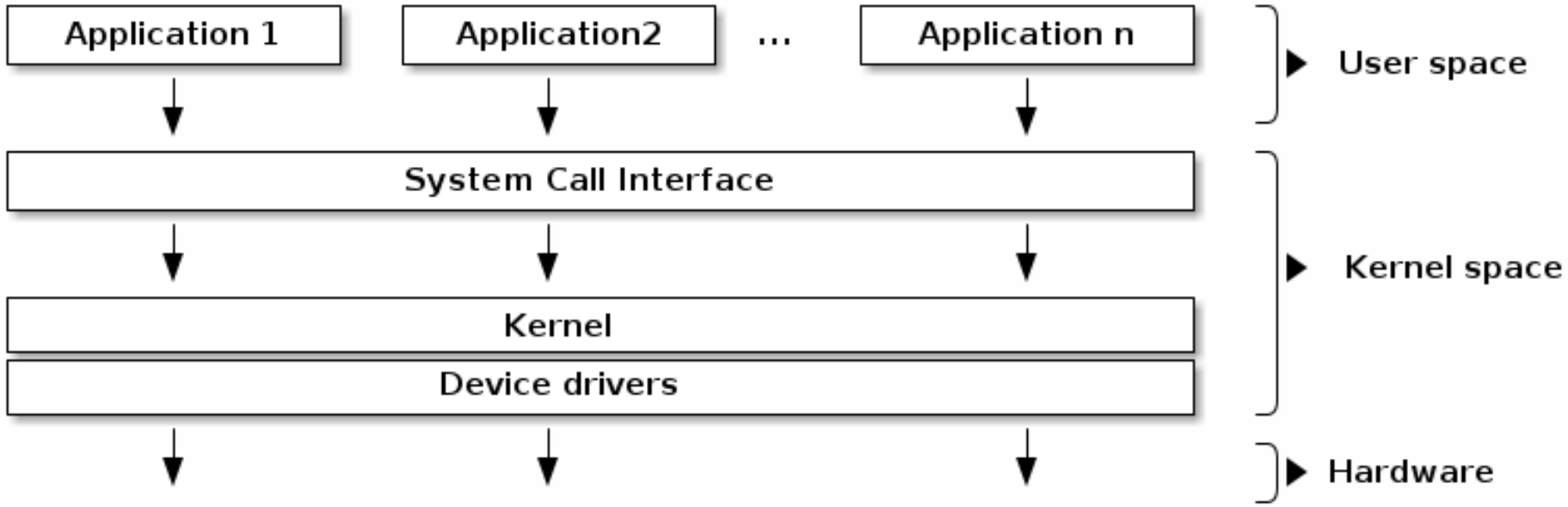
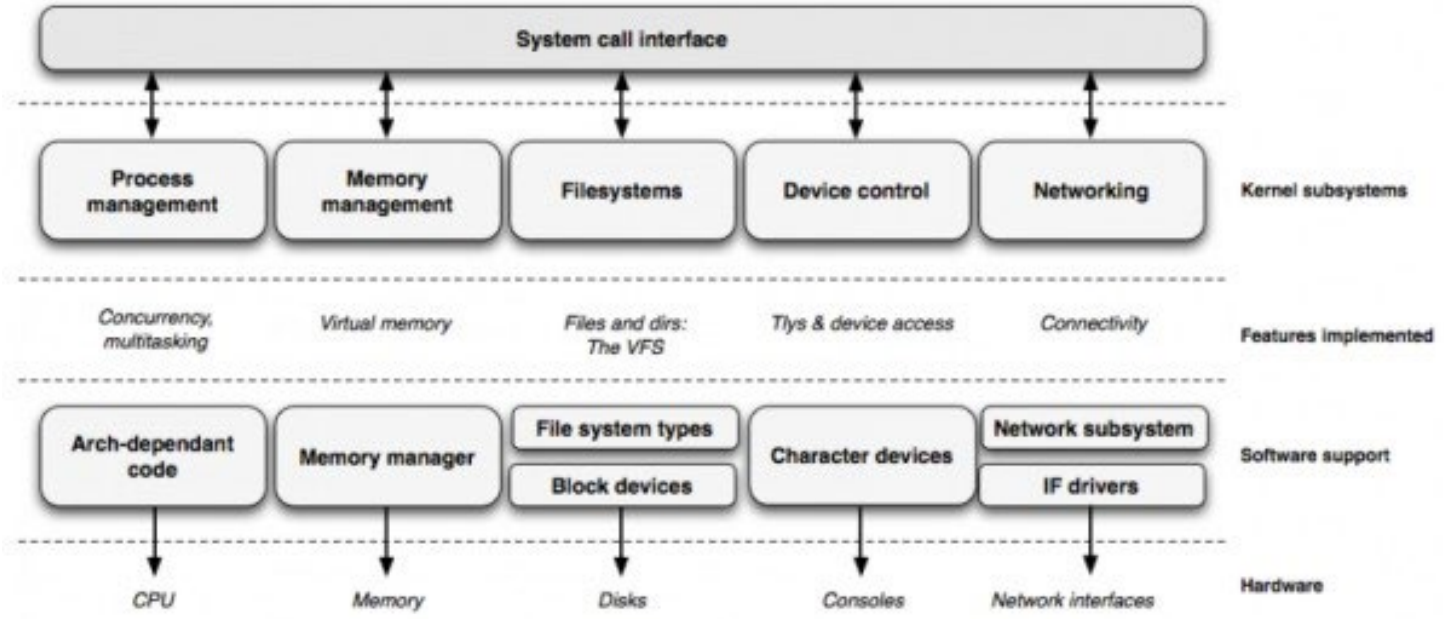
Linux for Devices

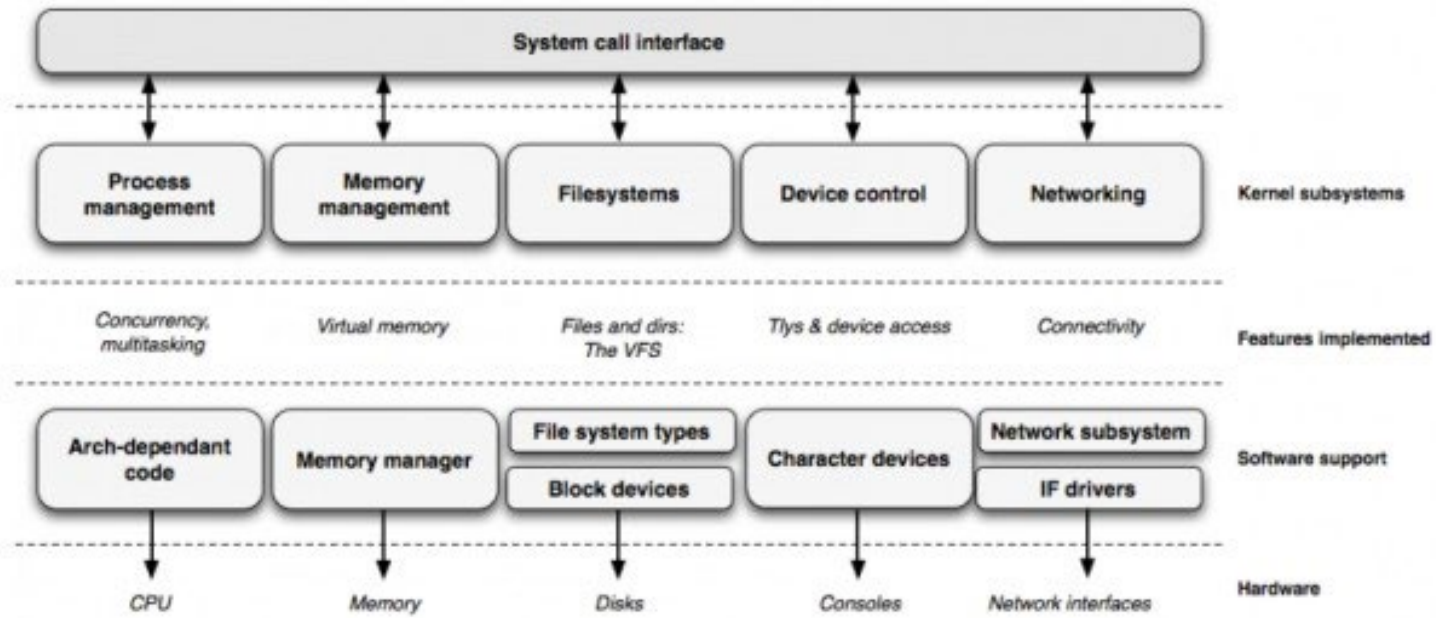
Module-5

By: Dr. A K Yadav (9911375598)
Dept of CSE, ASET, AUUP

About Linux Device Driver




System call interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: The VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependant code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks
Consoles
Network interfaces
Hardware


System call interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: The VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependant code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks
Consoles
Network interfaces
Hardware

- In Linux system, several concurrent process perform different tasks
- Each process asks for the system resource, be it memory, networking or any other
- The kernel is the big chunk of executable code in charge of handling such request like:
  - Process Management
  - Memory Management
  - File System
  - Device Control
  - Networking

- Device Driver takes a special role in the Linux Kernel
  - Linux device drivers are distinct “black boxes” that make a particular piece of hardware respond to a well-defined internal programming interface
  - They completely hide the details of how the device works
  - What is the need of device driver?
- Role of device driver
  - User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver.
- System Memory
  - Memory consists of RAM cells, whose contents can be accessed (i.e. read and written to) at extremely high speeds but are retained only temporarily (i.e. while in use or atmost while the power supply remains on)
  - System memory in Linux can be divided into two distinct regions: Kernel space and User space
- Linux is primarily divided into two parts: User space and Kernel space
  - Kernel space is where the core of the operating system runs and provides its services
  - User space is that the set of memory locations in which user processes (i.e. everything other than the kernel) run. A process is an executing instance of a program. One of the roles of the kernel is to manage individual user processes within this space and to prevent them for interfering with each other.
  - Device driver run in which space?
- These two components interacts through system call interface
  - Kernel space can be accessed by user processes only through the use of system calls
- Driver may be integrated directly into the kernel or can be build separately and plugged in at run time when needed i.e. Loadable module
- Linux way of looking at devices distinguish between three types:
  - Character Device: Accessed as stream of bytes
  - Block Device: Accessed by the file system (/dev)
  - Network Device

Device Control

- Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called device driver.
- The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive.

Linux Kernel Modules

- A piece of code that can be added to the base kernel is called a module
- As we have device drivers in Windows operating system, we have modules in Linux
- In short, device drivers are called modules in Linux. Both terms can be used interchangeably.

Base Kernel

- Kernel code which is compiled and loaded along with basic device drivers as a single entity is called base kernel
- Note that all parts of base kernel will be loaded all the time
- Once you have working base kernel, it is good to leave it untouched as long as possible

User Space and Kernel Space

- Kernel runs in kernel space and normal programs run in user space
- Normal programs in user space can't mess with memory (and other resources) owned by other programs or by the Linux kernel. This limits their ability to do corrupt things like crashing the machine
- The kernel is the core of the operating system. It normally has full access to the all memory and machine hardware (and everything else on the machine). To keep the machine as stable as possible, you normally want only the most trusted, well tested code to run in kernel mode/kernel space

Kernel Modules

- Static modules
  - These are compiled as part of the kernel and are available at anytime
  - These make the kernel larger and has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality
- Dynamic Modules
  - These modules are the pieces of code that can be loaded and unloaded into the kernel upon demand
  - These are also called Loadable Kernel Modules (LKM)
  - They extend the functionality of the kernel without the need to reboot the system
  - The advantage that it uses the memory more efficiently than the statically linked drivers

How exactly the dynamic module is loaded?

- Step 1: Assume you have written a dynamic module xyz. You will also write an application which tries to access xyz functionalities
- Step 2: Once Linux is up on running on your PC or device, you will dynamically insmod your xyz module using “insmod” command to RAM
- Step 3: Your application in userspace will try to access functionalities of your hardware through xyz module by calling Linux provided system calls. Normally they are called ioctl calls
- Once the required hardware functionality is retrieved by the user application, when xyz module is no more needed, we can request kernel to remove xyz module using rmmod command

Disadvantages of building your module as static

- Assume you have written a static module “xyz” for a device driver. When you compile this as static module along with the kernel, you will be adding extra size to the kernel image permanently
- Whenever you modify your “xyz” device driver, you need to recompile your entire kernel to build it
- Also machine need to be rebooted for the changes to take effect
- Especially while the device driver is not stable, when you build it as static module, you will face issues which takes more time to debug
- Take a scenario when there is a memory overflow bug in your “xyz” static module. Once it is statically compiled and the kernel is loaded, at some point of time, the kernel may crash
- In order to debug it people may take hours or days to fix it. Also, your kernel which includes this “xyz” part of it will be useless till you fix the crash issue
- This doesn’t happen when you build your “xyz” as dynamic module

Advantages of building your module as dynamic/LKM

- Assume you built your “xyz” module as dynamic
- With this, you don’t have to rebuild your kernel as often. It can be compiled separately
- It saves your time and spares you the possibility of introducing an error in rebuilding and reinstalling the base kernel
- It can be loaded onto kernel at run time without having the machine to reboot
- It saves you memory because you load them when you are actually using them
- It can be unloaded anytime and hence no permanent affect on kernel size since these are compiled and built separately from kernel, they are much faster to maintain and debug
- Since these are compiled and built separately from kernel, they are much faster to maintain and debug
- Each dynamic module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program

Implementing & Running Driver Module

- Setting up environment: Install necessary packages required for compiling a kernel module
  - Switch on to the super user mode
  - Zypper install kernel-devel kernel-headers dkms make bzip2
    - kernel-devel: are files that are required for compiling kernel code that will run as part of the kernel, such as kernel modules
    - kernel-headers: contains files that describe the system environment and are used for compiling normal programs that run in userspace
    - dkms: Dynamic kernel module support package
    - make: make the tool necessary for kernel code build system
    - bzip2: zipping the tool
  - reboot the system

Program

```
#include<linux/init.h>
#include<linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
        printk(KERN_ALERT "Hello World\n");
        return 0;

}

static void hello_exit(void)
{
        printk(KERN_ALERT "Goodbye");

}

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

Kernel headers used in the module

- #include<linux/init.h>
- This header contains the definition of the functions used in this module
- module_init() and module_exit() are defined in this file
- module_init() is a macro which defines the function to be called at module insertion time or at system boot time
- module_exit() is a macro which defines the function to be called at module removal time. This function is called cleanup function
- Kernel modules must always contain these two functions int_module and cleanup_module

Kernel headers used in module

- #include<linux/module.h>
- This header is included to add support for dynamic loading of module into the kernel
- It also contains the definitions of the special macros used in this module
- Special macros like MODULE_LICENSE is defined in this header
  - MODULE_LICENSE: This macro is used to tell the kernel that this module bears a free license, without such a declaration, the kernel complains when the module is loaded
- printk: This function is similar to printf in standard C. This is implemented to make kernel have its own printf function instead of having decency on C library
  - When using printk() when syslogd and klogd daemons are running, the output of printk() will be appended to /*var*/log/messages. Printk() if called with low priority will only be appended to /*var*/log/messages. To ensure that it prints to the console use KERN_ALERT priority.
- KERN_ALERT: This string is the priority of message. KERN_ALERT is the highest priority set to specify kernel to print the message on console.

Hello World Kernel module functions

- hello_init and hello_exit
- hello_init()
  - This particular function is called when the module is loaded
- hello_exit()
  - This particular function is called when module removed

Makefile Code

- obj-m := hello.o
- KDIR := /lib/modules/uname -a/build
- PWD := $(shell pwd)
- default:
- 	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD)

Makefile

- Obj-m:=hello.o
  - This assignment states that there is one module to be built from object file hello.o

- The kernel build system in Linux kernel is designed in such a way to handle the rest of the parameters
  - KDIR := /*lib/*modules/uname -a/build Kernel build directory path
  - PWD := $(shell pwd)
    - Current directory path
- Default:
  - $(MAKE) -C $(KDIR) SUBDIRS=$(PWD)
    - This is the main make command which switches to your kernel build directory and invokes the make command to compile helloworld kernel module present in the sub directory
    - -C option tells make command to switch to $(KDIR) which is kernel directory

Loading Kernel Modules

- Two kernel utilities to load the modules
  - insmod
  - modprobe
- insmod
  - Is a kernel utility that installs loadable kernel modules into the kernel
  - It actually loads the module code and data into the kernel land, links any unresolved symbols in the module to the symbol table of the kernel
  - insmod accepts a number of command line options and it can assign a value to parameters in a module before linking it to the current
  - Note that if a module is correctly designed, it can be configured at load time by passing arguments to insmod
  - Syntax: insmod [filename] [module-options…]
- For example: insmod hello.ko
- insmod /*path/*to/my_module.ko param1=0x330
- In order to know if your insmod command has caused any errors, try using “dmesg” command to see the output
  - Note: insmod only accepts one file at a time

Modprobe

- It's a linux utility which offers more features than basic insmod
- Advantages include
  - It has the ability to decide which modules to load
  - */lib*/modules/$(uname -r)
  - It's aware of module dependencies
  - It supports resolution of recursive module dependencies
  - e.g module x dependent on module y which is dependent on module z
- Syntax:
  - modprobe module_name
  - modprobe hello
  - Note: no need to use .ko extension
  - If syntax is not working then you need to run depmod to re-create the module dependency list. Run:
  - sudo depmod

Difference between insmod and modprobe

- While loading a module, insmod fails with “Unresolved symbol” error when an unresolved symbol is present in the loading module
- Modprobe looks at the module to be loaded to see whether it references any symbols that are not currently defined in the kernel. If any such references are found, modprobe looks for other modules in the current module search path that define the relevant symbols
- When modprobe finds these modules which are needed by the module being loaded, it loads them into the kernel as well
- In short, while loading a module into the kernel, if that module has any dependency on another module, modprobe find those modules from its search path, it loads them into the kernel
- Note modprobe is aware of default location of module knows how to figure out the dependencies and load the modules in the right order
- We need to have root permissions for executing both insmod and modprobe commands
- When a module is loaded, any symbol exported by the module becomes part of kernel symbol table

Kernel symbol table

- The kernel symbol table contains the addresses of global kernel items, functions and variables that are needed to implement modularized drivers

Listing modules currently loaded in the kernel

- We use a linux utility called lsmod
- lsmod command lists the modules currently loaded in the kernel. lsmod works by reading the */proc*/modules virtual file
- Information on currently loaded modules can also be found in the sysfs virtual file system under */sys*/modules
- Syntax:
- lsmod

Unloading kernel module

- Two kernel utilities to unload the modues are:
  - rmmod
  - modprobe
- rmmod
  - Rmmod is a linux utility which unloads the loadable module
  - Module should be already loaded to execute rmmod command
  - Module fails if the kernel believes that the module is still in use or if the kernel has been configured to disallow module removal
  - Syntax
    - rmmod [module_name]
    - e.g rmmod hello
    - Note: we need to be a superuser for executing this command
- Modprobe
  - Syntax
    - modprobe -r [module_name]
    - e.g modprobe -r hello

Important question

- Is it possible to forcefully remove the kernel modules when they are in use?
- Yes, its possible to configure the kernel to allow forced removal of modules, even when they appear to be busy
- Commands normally used are
  - rmmod -f module_name
  - modprobe -rf module_name
  - These commands only work when you enable the kernel configuration CONFIG_MODULE_FORCE_UNLOAD

Revise: Loadable Kernel Modules

- Linux has the ability of run time extension which means you can add functionality to the kernel while the system is up or running
- Each piece of code that can be added to the kernel at run time is called a module
- The module defines two functions: One to be invoked when the module is loaded into the kernel and other when the module is removed
- The module_init() and module_exit() special kernel macros to indicate the role of the two functions
- MODULE_LICENSE() macro is used to inform the kernel that this module bears a free license
- printk() is used to display output
- cat /*var*/log/syslog or dmesg will display your message

Revise: Loadable Kernel Modules

- To build your module, you need to create Makefile which will give the .ko file as the resulting module
- After Makefile, insmod program loads the module code and data into the kernel
- insmod typically allocates kernel memory to hold the module. It then copies the module text into the memory region
- It performs the similar function to that of GNU ld(linker) and links any resolved symbol into the module to the symbol table of linux kernel and calls module initialization function
- rmmod() is used to unload the module from the kernel
- lsmod program produce a list of modules currently loaded in the kernel
- The __init and __initdata is used to tell the kernel that function is used only initialization, it will drop the init function after loading, making its memory available for another use

Revise: User Space Vs Kernel Space

- Module runs in the kernel space, whereas application runs in the user space
- The kernel executes in the highest level, where everything is allowed and application executes at the lowest level, where the processor regulates direct access to hardware and unauthorized access to memory
- System calls are the interface where user space and kernel space can communicate with each other
- Kernel code executing a system call is working in the context of a process-it operates on behalf of the calling process and is able to access data in the process's address space
- Code that handles interrupts, on the other hand, is asynchronous with respect to processes and is not related to any particular process
- Some functions in the module are executed as part of the system calls, and some are in charge of interrupt handling

Kernel modules Vs Applications

- Initialization
  - Most small and medium sized application programs perform a single task from beginning to end
  - Whereas every kernel module registers itself with kernel in order to serve future requests, and its initialization function terminates immediately
  - In other words, the task of the module's initialization function is to prepare for later invocation of the module's function when required

Kernel modules Vs Applications

- Which are event driven KM or AP?
  - When a kernel module is loaded, it invokes init function (like hello_init) which conveys kernel “Here I am, and this is what I can do”
  - When kernel module is unloaded, it invokes exit function (like hello_exit) which conveys kernel “I am not there anymore; don’t ask me to do anything else”
  - This kind of approach to programming is similar to event driven programming, but while not all applications are event-driven, each and every kernel module is event driven

Kernel modules Vs Applications

- About exit functions in KM and AP
  - Major difference between event-driven applications and kernel code is in the exit function
  - When an application terminates, it can be lazy in releasing resources or avoids clean up together, the exit function of a module must carefully undo everything otherwise there is a chance that few pieces remain around until the system is rebooted

Kernel modules Vs Applications

- Ability to unload a module
  - Incidentally, the ability to unload a module is one of the features of modularization that you will most appreciate, because it helps cut down development time; you can test successive versions of your new driver without going through the lengthy shut down/reboot cycle each time
  - With application, you need to either exit it completely and return the application. You will not be sure if the memory allocated is deallocated or the task initiated is exited cleanly. It all depends on how you implement the application

Kernel modules Vs Applications

- Handling faults
  - Segmentation fault is harmless during application development and a debugger can always be used to trace the error to the problem in the source code; a kernel fault kills the current process atleast, if not the whole system

Kernel symbol tables

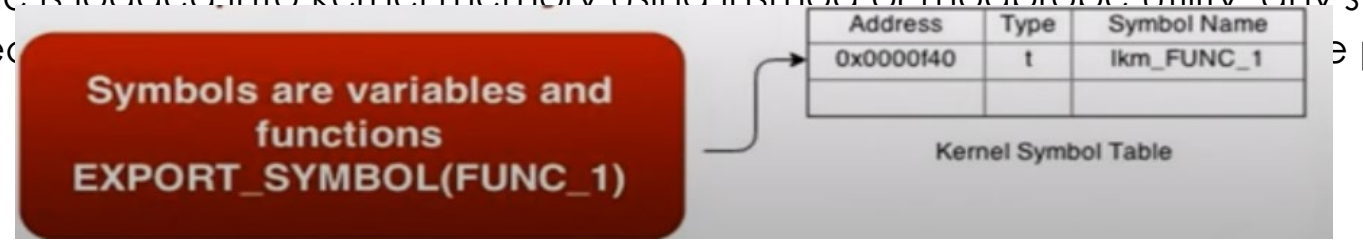
- Symbol in Linux
  - In Linux, symbols are nothing but variables and functions that are needed to implement modularized drivers. Note that each symbol has its address in memory

How symbols are exported?

- Exporting kernel symbols is typically done with
  - EXPORT_SYMBOL()
  - EXPORT_SYMBOL_GPL()
- EXPORT_SYMBOL(), which exports a given symbol to all loadable modules
- EXPORT_SYMBOL_GPL(), which exports a given symbol to only those modules that have a GPL compatible license
  - So whenever you write a driver, it should be GPL licensed. That will be great facility in case your module wants to use any other module's functions that are already exported using GPL

Symbol and Symbol table relation

- Kernel symbol table is nothing but a look up table between symbol names and their address in memory
- When a module is loaded into kernel memory using insmod or modprobe utility, any symbol exported by the module bec e public kernel symbols



More about symbol table

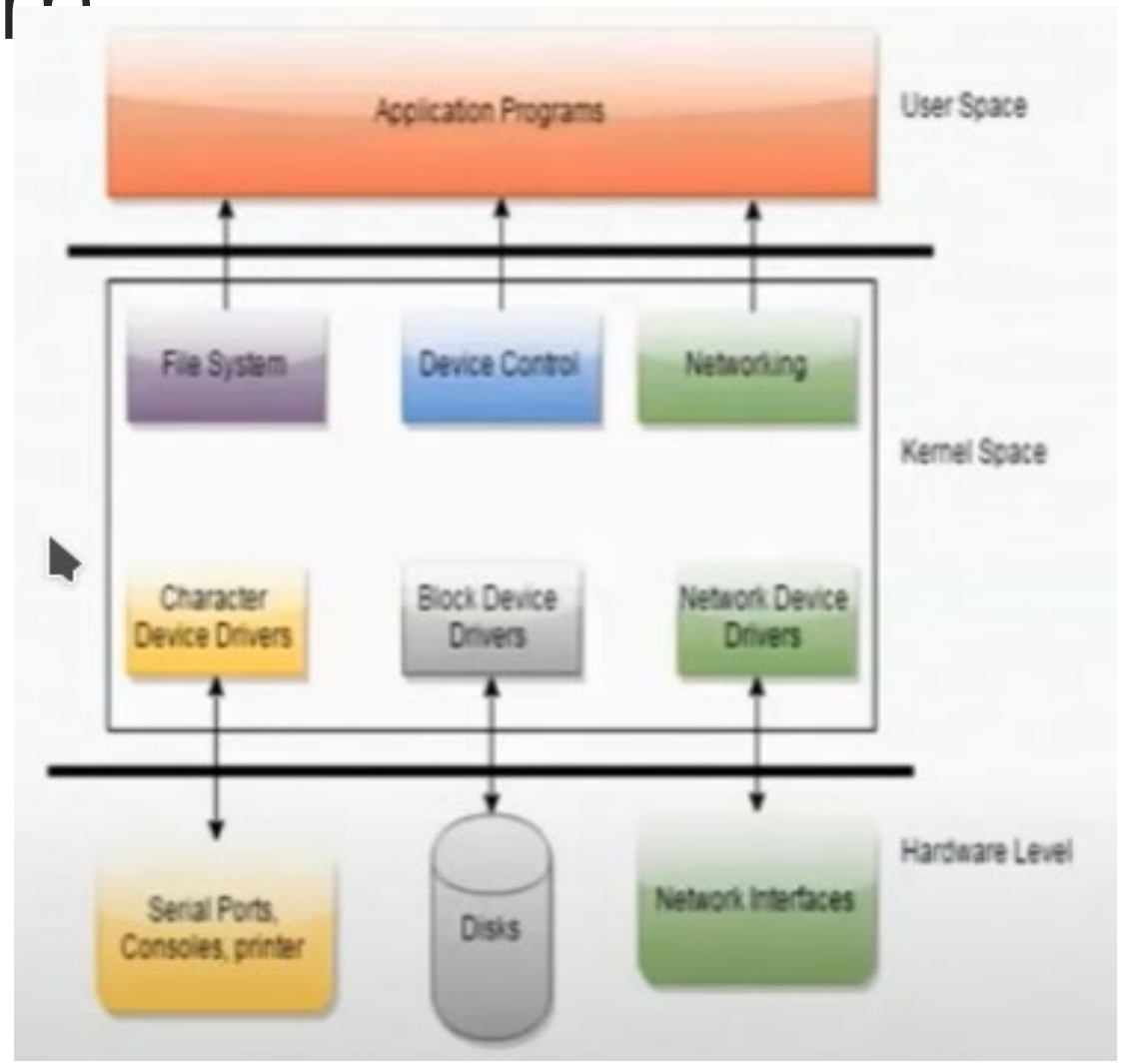
- While insmoding, the insmod utility resolves undefined symbols against the table of public kernel symbols
- Kernel symbol tables hold all the information needed to find program symbols, assign value to them and relocate them
- Primary task of symbol is to associate a string with a value. For ex, printk symbol represents the address of the printk function in virtual address space where the machine code resides
- This kernel symbol table is loaded into memory as part of the kernel boot process
-

Basics of device driver types

- About /dev directory
  - The /dev directory on a Linux machine is where all the device files for the system are located
  - Using this device file, a user program or application can access a specific hardware device
  - /dev is a very interesting directory that highlights one important aspect of the Linux filesystem
  - Take an example of a Printer driver in your Linux system. The device file of printer is /*dev*/lp0. Any data written to this file by your application will be redirected to your printer
  - Similarly, data read/written to /*dev*/ttyS0 will allow you to communicate with a device like serial terminal or a modem device
- Device driver types
  - In Linux, we have following device types
    - Character devices
    - Block devices
    - Network devices

Device types diagram

- As shown in the diagram, all three device drivers fall under kernel space
- Each device has it's device driver
- These device driver has an entry of its device file under /dev special directory
- An application which resides in User space, communicates through device files under /dev and access the specific device it needs



Introduction to Character Device Driver

- Character devices can be compared to normal files in which we can read/write arbitrary bytes at a time. Character device is one that can access as a stream of bytes like a file
- The difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially
- Character devices are accessed through names in the file system called Device files
- However, character devices are not limited to performing one character at a time(despite the name "character device"). For example, tape drivers (which is character device) frequently performs I/O operations in 10K chunks. You can also use a character device which it is necessary to copy data directly to or from a user process

Character Device Driver

- Examples of character device drivers are:
  - Serial port(/dev/ttyS0)
  - Text console(/dev/console)
  - Mice
  - Parallel printer ports
- Character devices are accessed by means of filesystem nodes, such as */dev*/tty1 and */dev*/lp0
- Since /dev contains all devices, how can we identify character device drivers under this directory?

Block Devices

- Operate on blocks of data, not on arbitrary bytes. Usual block size is 512 bytes or larger powers of two
- Linux allows the application to read and write a block device like a char device. It permits the transfer of any number of bytes at a time
- As a result, block and char devices differ only in the way data is managed internally by the kernel
- Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user
- Examples of block device drivers include
  - Hard drives
  - CD-ROM drives
- Since /dev contains all devices, how can we identify block device under this directory

Network device

- Network device drivers receive and transmit data packets on hardware interface that connect to external systems, and provide a uniform interface that network protocol can access
- The difference between block device driver(disk) and Network device driver (packet-delivery interface) is that the block device exists as a special file in the /dev directory whereas network interface has no such entry point
- These network interfaces exist in their own namespace and export a different set of operations

Minor and Major Numbers

- These are also called device numbers
- If you issue “ls -l” command in /dev directory, two numbers (separated by a comma) in the device file entries just before the date of the last modification, where the file length normally appears

Major Number

- Is an identifier which identifies the driver associated with the device. So, each device will have a major number
- These major numbers can be defined by us (only if we know exactly which major number is available) or you can leave this job to kernel. Either way kernel knows to which major number our device is associated with

Minor Number

- Is an identifier which is used by the kernel to determine exactly which device is being referred to
- Allocating minor number is same as major number but only difference is the kernel knows nothing about minor numbers beyond the fact that they refer to devices implemented by your drivers
- Note that there can be number of devices which can be controlled by a single driver. Multiple disks with a single block device driver is a good example
- Example ls ~~l */dev/sda*~~

Representation of device numbers

- In order to represent major and minor numbers in specific format, Linux kernel developers have embedded major and minor numbers in a structure dev_t type which is defined in <linux/types.h>

20 for the

Revise: Major Number & Minor Number

- The major number identifies the driver associated with the device. Kernel uses the major number to invoke the driver.
- The minor is used by the driver to check exactly which device is referred to
- Within the kernel, the dev_t type (defined in <linux/types.h>) is used to hold device numbers-both the major and minor parts

Allocating Major & Minor Number

- We can allocate the Major and Minor number in two ways:
  - Statically
    - int register_chrdev_region(dev_t first, unsigned int count, char *name);
    - The dev_t type(defined in <linux/types.h>) is used to hold device numbers-both the major and minor parts.
    - If you want to create the dev_t structure variable for your major and minor number:
      - MKDEV(int major, int minor);
    - To get your major number and minor number from dev_t, use below method
      - MAJOR(dev_t dev);
      - MINOR(dev_t dev);
  - Dynamically
    - int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

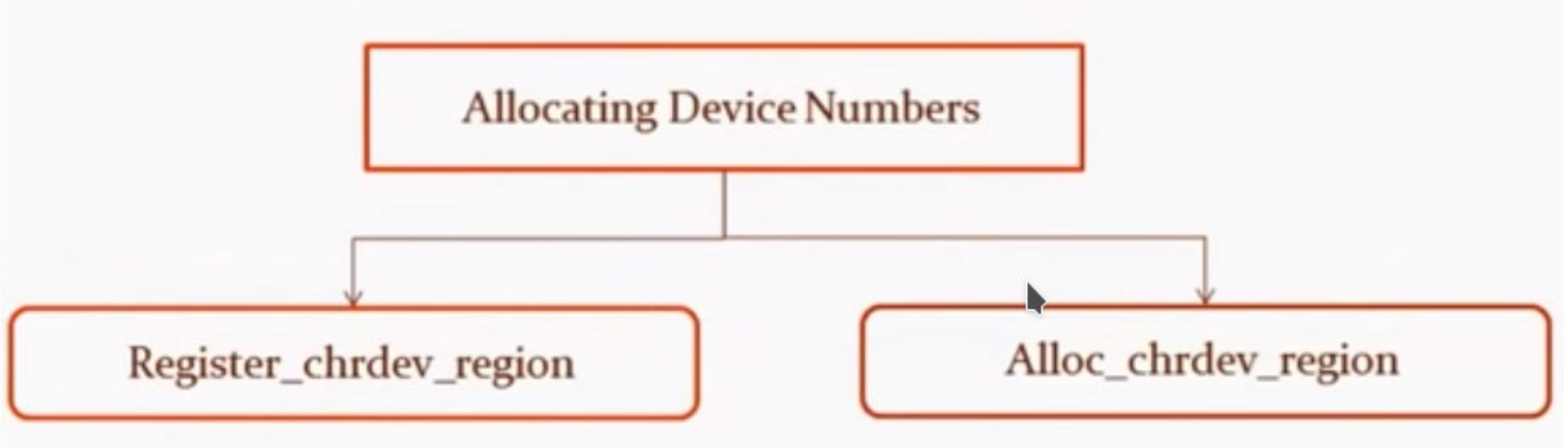
Allocating device numbers

- While setting up a char device, first thing our driver need to do is to obtain one or more device numbers to work with
- To do so, Linux kernel provides following functions
  - int register_chrdev_region(dev_t first, unsigned int count, char *name);
  - int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

register_chrdev_region & alloc_chrdev_region

- int register_chrdev_region(dev_t first, unsigned int count, char *name);
  - first – is the beginning device number of the range you would like to allocate
  - count – is the total number of contiguous device numbers you are requesting
  - Name – is the name of the device that should be associated with this number range; it will appear in /*procs/*devices and sysfs
  - Return value of this function will be 0 on success. In case of error, a negative error code will be returned and obviously you will not have access to the requested region
- int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);
  - 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 of parameters work like those given to request_chrdev_region
- Note: The usual place to call these functions would be in your module's init function

Recap


Allocating Device Numbers
Register_chrdev_region
Alloc_chrdev_region

Freeing device numbers

- Now in order to free this region when its no longer required, kernel provides following function
  - Void unregister_chrdev_region(dev_t first, unsigned int count);
  - The usual place to call unregister_chrdev_region would be in your module's clean-up function

Best way to allocate device numbers

- As a device driver programmer, we have two choices to pick major number for a device
  - Pick a random number that appears to be unused
  - Allocate major numbers in a dynamic manner
- If we randomly pick a number that is unused, the driver works until it's used by only you. If your driver is widely used, a randomly picked major number will lead to conflict and cause more trouble
- So, it's always recommended to dynamically allocate major numbers for a new driver by using alloc_chrdev_region rather than register_chrdev_region

How application communicate with the hardware device

- First application will open the device file which is created by device driver which we can create in two ways:
  - Manual: mknod -m 666 /*dev*/char_device c 246 0
  - Automatically:
    - Include the header file linux/device.h and linux/kdev_t.h
    - Create the struct class
      - Struct class **class_create(struct module** *owner, const char *name);
    - Create device with the class
      - Struct device *device_create(struct *class, struct device *parent, dev_t dev, const char *fmt,…
  - You can destroy the device using device_destroy()
    - Void device_destroy(struct class *class, dev_t devt);