LINUX DEVICE DRIVERS

VIVEN
Embedded Academy

Viven Embedded Academy
Hyderabad.

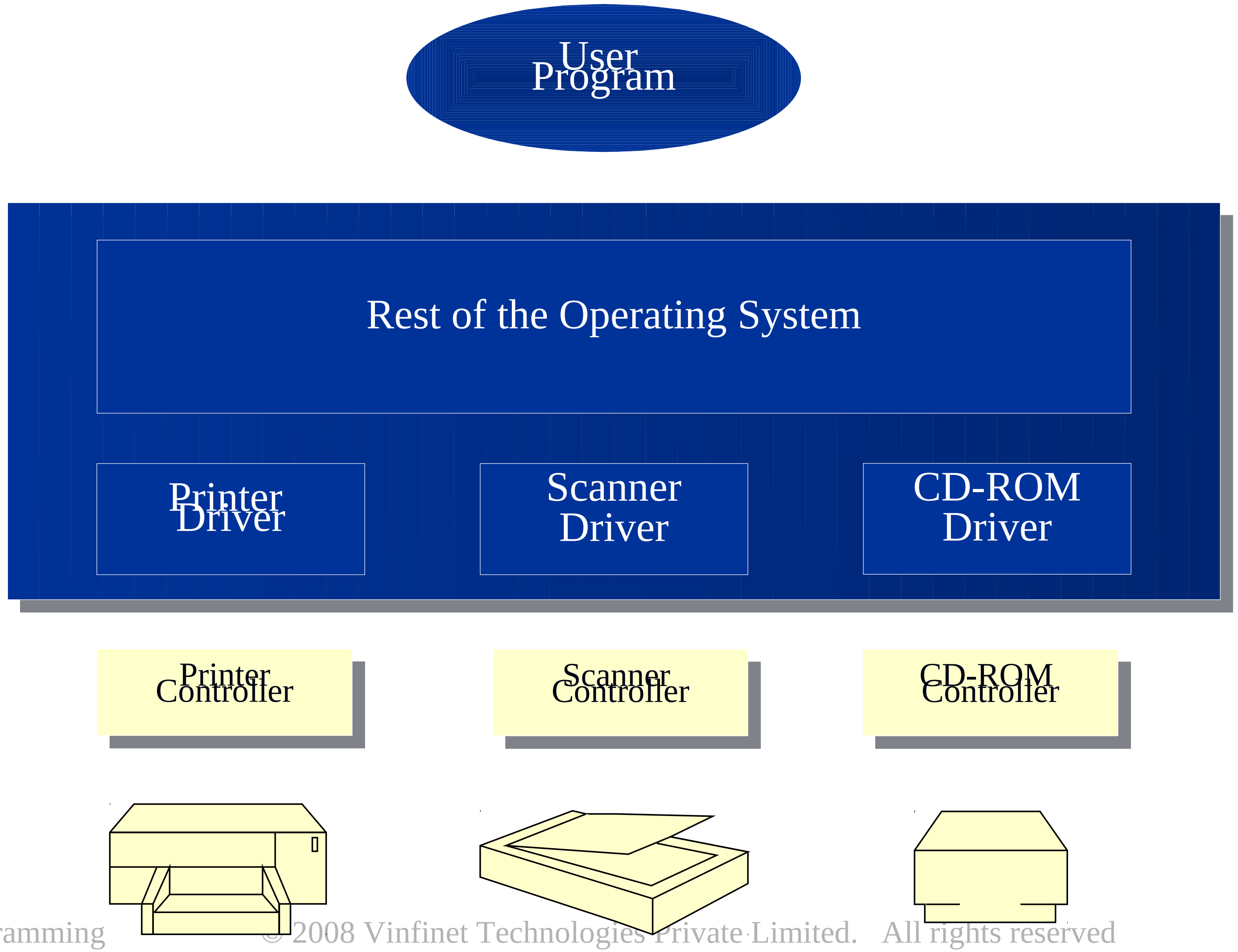
What is a Device

- Any peripheral such as a graphics display , disk driver, terminal, or printer is a device.
- A device is usually considered to be piece of hardware that you can connect to your computer system and that you wish to manipulate by sending command and data
- A device can be a software device such as random number device, virtual consoles, loop back devices etc.,

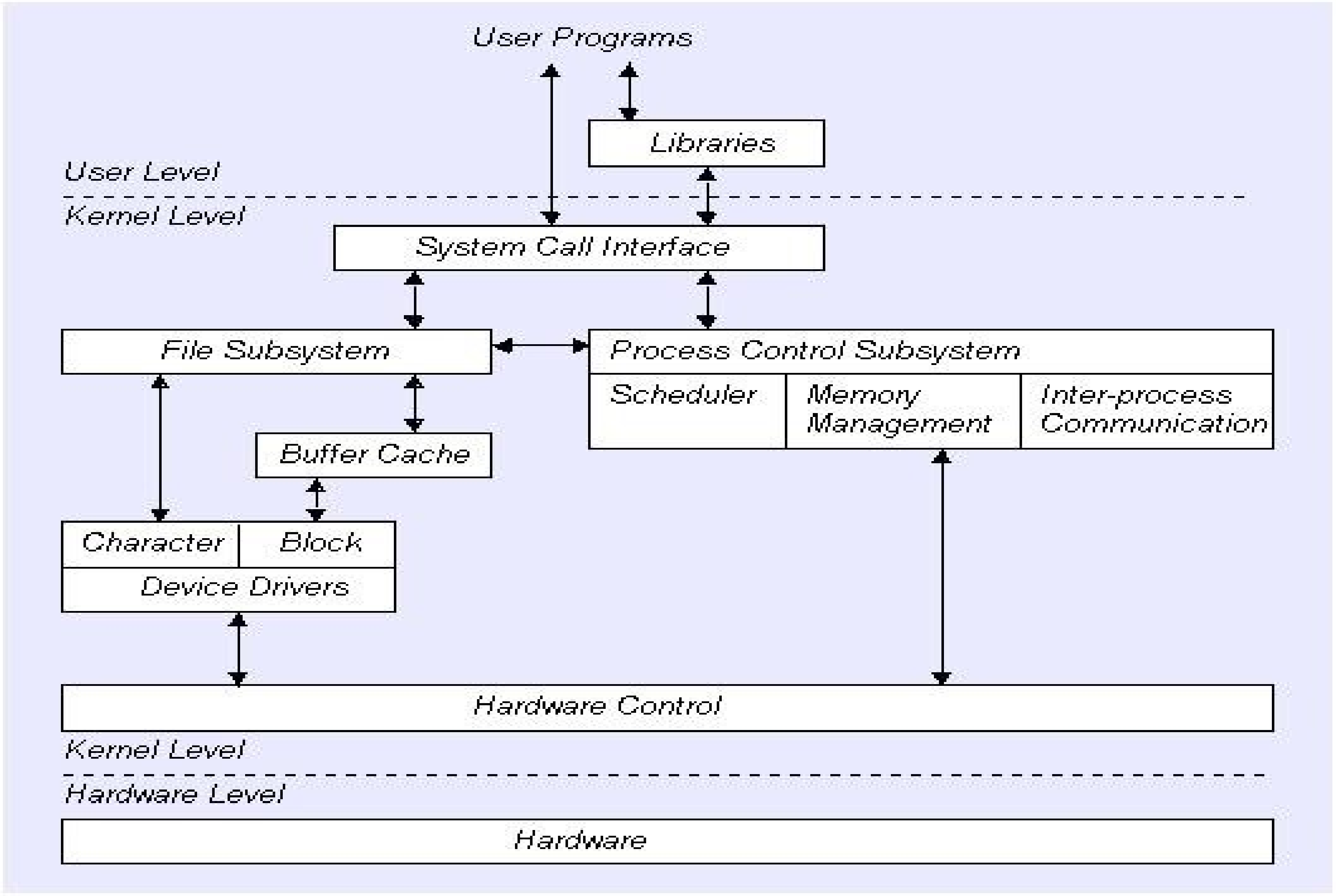
What is a Device Driver

- What's Device Driver?
  - “Black Boxes” respond to well defined internal API.
  - Hide completely the details of device works.
- The role
  - Mapping API calls to device specific hardware operations.
- Mechanism vs policy
  - What capabilities are to be provided(Mechanism)
  - How those capabilities are to be used(Policy)
  - Policy free drivers
  - Eg X Server vs Window Manager(KDE/GNOME).

What is a Device Driver



Linux I/O Subsystem



Classes of devices and modules

- Character devices
  - Can accesses as stream of bytes
  - /dev/console, /dev/ttyS0
  - Accessed by means of filesystem nodes tty1, lp0
- Block devices
  - Something that can host a filesystem (e.g. disk)
  - Can accesses ONLY as multiples of a block (e.g. 1K)
  - Different with char device is transparent to user under Linux.
- Network interfaces (eth0, eth1) – stream of packets
- USB, Firewire, SCSI, I20 ….

Character devices

Accessed as a stream of bytes (like a file)

- Such drivers at least implement ***open, close, read and write*** system calls
- Examples:
  - /dev/console (Text Console)
  - /dev/ttyS0 (Serial Ports)
  - /dev/lp0 (Line Printers)

•Character devices are accesed by means of file system nodes., such as /dev/tty1 and /dev/lp0. The only difference between difference between a char device and a regular file is that you can always move back and forth in the reuglar file, whereas most char devices are just data channels, which you can only access sequentially.

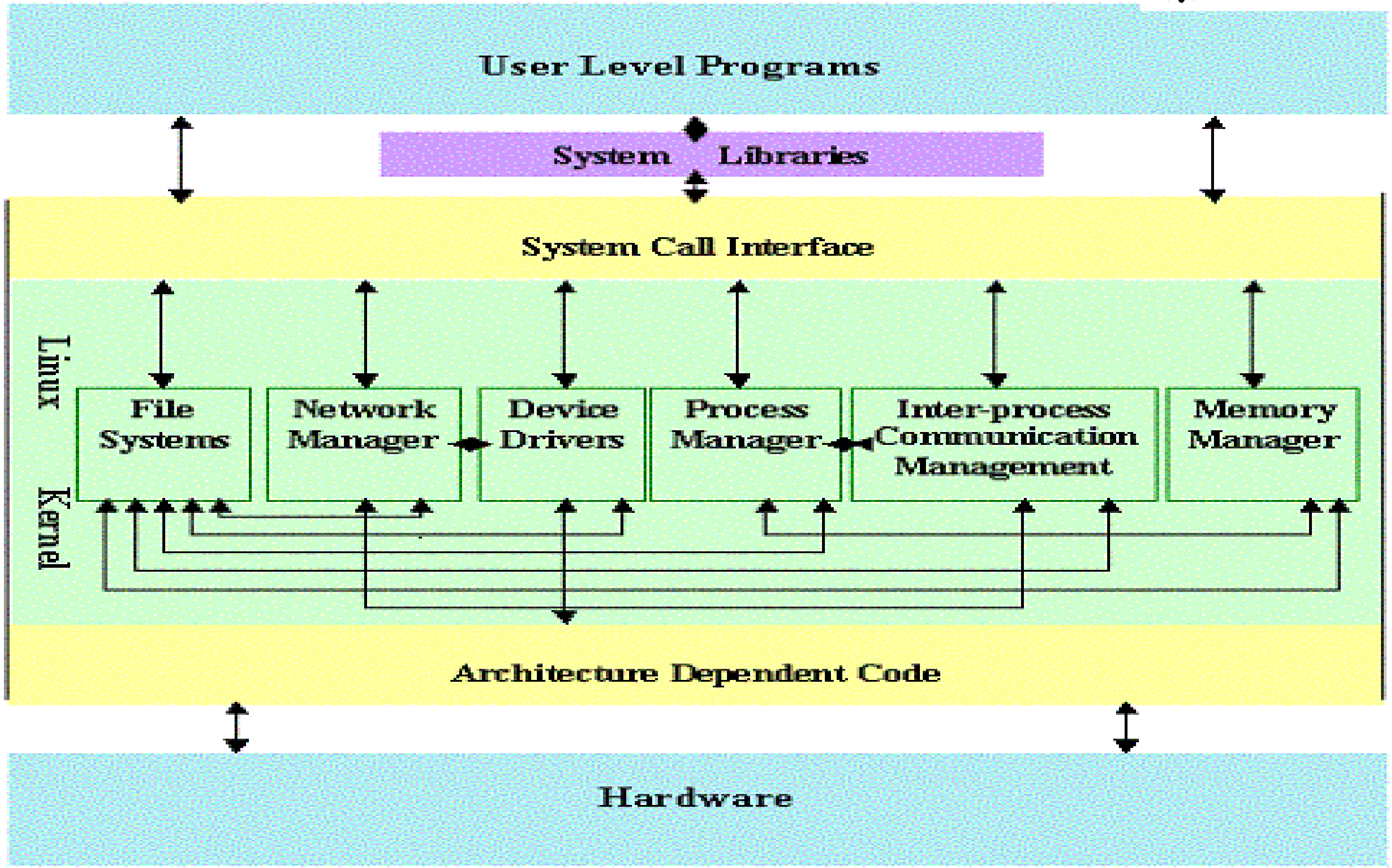
Block devices

- Block Devices also accessed by filesystem nodes in the /dev directory.
- Block Device can host a filesystem such as disk.
- Block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes(or a large power of 2) bytes in in length.
- 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.
- Block and Char devices differ only in the way data is managed internally by the kernel,.

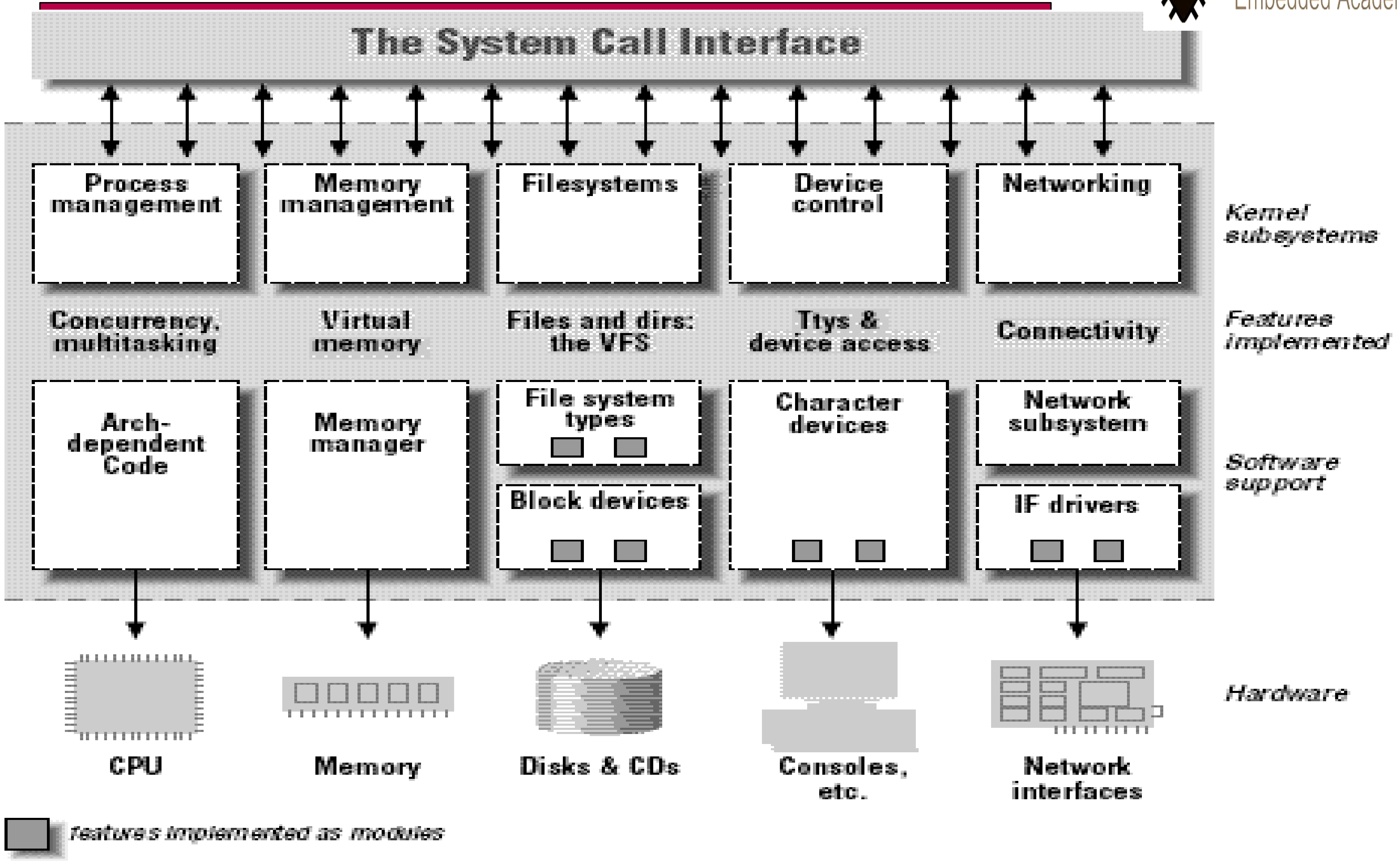
Network devices

- Any network interconnection is made through the interface(device), that is able to exchange data with other hosts.
  - Transaction made using an interface.
  - Interface can be hardware or software.
  - Interface is incharge of sending and receiving data packets, driven by the network subsystem of the kernel.
  - Communcation between kernel and network driver is completely different. Instead of read and write, the kernel calls functions related to packet transmission.
  - Though UNIX will map it as eth0, it doesn't have any corresponding entry in the file system.
  - A network driver knows nothing about individual connections. It only handles packets.

Linux Kernel Internals



Split view of kernel



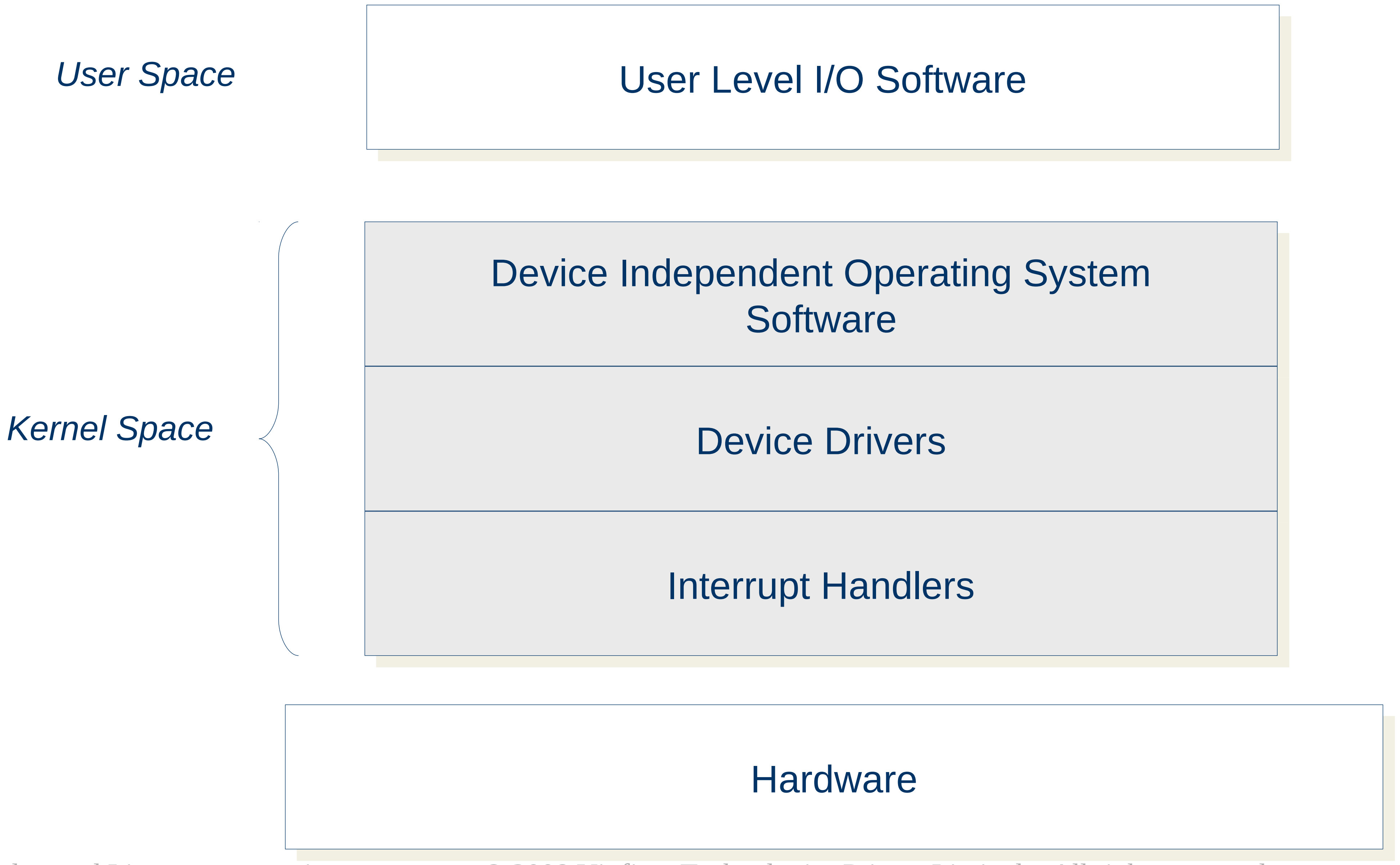
Linux Device Drivers

- Distinct Black Boxes
- And they make a particular piece of H/W, respond to a well-defined internal programming interface
- They hide completely the details of how the device works
- A 'C' Program that controls a device.
- They stand between kernel and peripherals, translating requests for work into activities by hardware.

Device Drivers

- A device controller has:
  - Control registers
  - Status registers
  - Data buffers
- This structure changes from device to device
  - A mouse driver should know how far the mouse has moved and which buttons are pressed.
  - A disk driver should know about sectors, tracks, cylinders, heads, arm motion, motor drives, head settling times, …
- Each I/O device need some device specific code for controlling it.
  - This code is called device driver.

I/O Software Layers



Applications Vs Device Drivers

- Applications performs a single task from beginning to end, A module registers itself in order to serve future requests.
- Application call functions resolves external references using appropriate lib functions during link stage, whereas modules are linked only to kernel, and only functions it can call are exported by kernel.

Locating Driver Entry Points – Switch Table

- The kernel maintains a set of data structures known as :
    - -block device switch table
    - -character device switch table
- These data structures are used to locate and invoke the entry points of a device driver.
- Each switch table is an array of structures.Each structure contains a number of function pointers.
- To decide which element of the switch table should be used, the system uses the “major device number” associated with the device special file that represents the device.

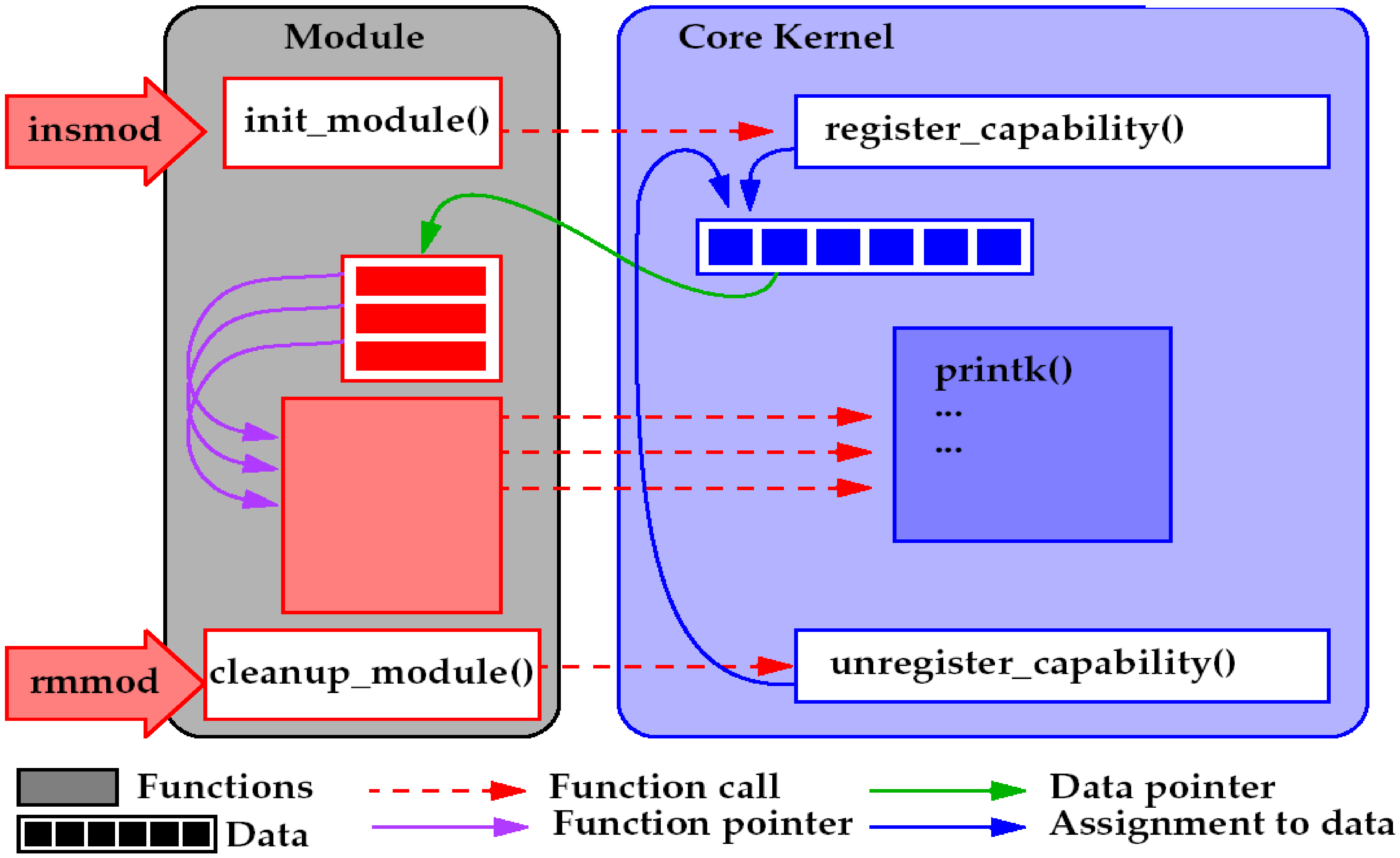
Locating Driver Entry Points – Switch Table

- Having decided which element of the switch table to use, the system decides which entry-point of the element should be used.
- This depends upon the system call used by the process. If the process used “open” system call, then the “open” entry point is used and so on……

Hello World Module

- This module defines two functions, one to be invoked when the module is loaded into the kernel(hello_init). And one for when the module is removed (hello_exit).
- **MODULE_LICENSE** is used to tell the kernel that this module bears a free license. Without such a declaration the kernel complains when when the module is loaded.
- The **printk** function is defined in the linux kernel and made available to modules, It behaves similar to the standard C library function printf. The string KERN_ALERT is the priority of the message.
- The kernel runs without the help of the C library.
- /linux/module.h contains many definitions of symbols and functions needed by loadable modules.
- /linux/init.h to specify your initialization and cleanup functions.

Linking a module to the kernel



Kernel Module Vs Applications

- Applications perform a single task from begining to end whereas kernel modules register itself in order to serve future requests and its initialization function terminates immediately.(Event driven programming).
- The exit function of a module must carefully undo everything the init function built up, or the peices remain in the system till it is booted.
- Ability to unload a module is a one of the good feature. Otherwise during development you need to go through the lengthy shutdown/reboot cycle each time.
- Applications can call functions it doesnt define; the linking stage resolves external references using the appropirate library of functions. A module is linked only to the kernel, and the only functions it can call are the ones exported by the kernel. There are no libraries to link.

Compiling Modules

- The build process for modules differs significantly from that used for user-space applications.
- The first thing is to ensure you have sufficiently current versions of the compiler, module utilities, and other necessary tools.
- Trying to build a kernel(and its modules) with the wrong tool versions can lead to difficult problems.
- You cannot build a loadable modules for a 2.6 kernel without this source tree on your filesystem.
- Statements in makefile
  - Obj-m := hello.o (States that there is one module to be built from the object file hello.o. The resulting module is named hello.ko after being built from the object file.)
  - Obj-m := module.o

module-objs := file1.o file2.o (for two files)

Compiling Modules

- Make command required to build your module is
  - $(MAKE) -C <kernel source directory path> M='pwd' modules
- This command starts by changing its directory to the one provided with the -C option which is kernel source directory. There it finds the kernel top level make file.
- The $(MAKE) – invokes the kernel build system and the kernel makefiles take care of actually building the modules.
- The M= option causes that makefile to move back into your module source directory before tyring to build the modules target. The modules are build in the path mention by the M= option.

Loading and Unloading Modules

- **insmod** loads modules into the kernel.
  - insmod hello.ko
- It 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 values to parameters in your module before linking it to kernel. If a module is correctly designed, it can be configured at load time. (Explained in Module Parameters)
- Insmod relies on a system call **sys_init_module** defined in kernel/module.c. This function allocates krenel memory to hold a module, it then copies the module text int that memory region, resolves the kernel references in the module via the kernel sysmbol table, and calls the module's initialization function.

Loading and Unloading Modules

- **Modprobe** like insmod loads a module into the kenrel, it will look at eh 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 relavent symbols. When modprobe finds those modules it loads them into kernel as well.
- If you use insmod in this situation the command fails with 'unresolved symbol' message symbol message left in the system logfile.

Loading and Unloading Modules

- **rmmod** utilty is used to remove the module.
  - rmmod hello.ko
- Rmmod removal fails if the kernel believes that the module is till in use or the kernel has been configured to disallow module removal.
- The **lsmod** module lists the modules currently loaded in the kernel. Lsmod works by reading the /proc/modules virtual file. Information on currently loaded module can also be found in the sysfs virutal filesystem under /sys/module.
- A look into the system log file (/var/log/messages wil reveal the specific problem that caused the module to fail to load.

Loading and Unloading Modules

- **rmmod** utilty is used to remove the module.
  - rmmod hello.ko
- Rmmod removal fails if the kernel believes that the module is till in use or the kernel has been configured to disallow module removal.
- The **lsmod** module lists the modules currently loaded in the kernel. Lsmod works by reading the /proc/modules virtual file. Information on currently loaded module can also be found in the sysfs virutal filesystem under /sys/module.
- A look into the system log file (/var/log/messages wil reveal the specific problem that caused the module to fail to load.

Initialization & Shutdown

- Module initialization registers any functionality offered by the module.
- The functionality can be a new driver or a software abstraction that can be accessed by an application.
- Initialization function should be declared static, since they are not meant to be visible outside the specific file.ldd
- Modules can register many different types of functionalities, including different kinds of devices, filesystems an0d etc. For each functionality there is a specific kernel function that accomplishes the registration.
- The arguments passed to the kernel registration function are usually pointers to data structures which usually contains pointers to module functions, which is how the functions in the module body get callled.

Major and Minor Numbers

- Char devices are accessed through names in the filesytem. Those names are called special files or device files or simply nodes of the filesystem tree. They are located in the /dev directory.
- Special files for char drivers are identified by a “c” in the first column of the output of ls -l.

```
crw-rw----  1 root root   4,  64 2011-01-10 12:15 ttyS0
crw-rw----  1 root tty   4,  65 2011-01-10 12:15 ttyS1
crw-rw----  1 root root   4,  66 2011-01-10 12:15 ttyS2
crw-rw----  1 root root   4,  67 2011-01-10 12:15 ttyS3
```

- If you issue ls -l command, you'll see two numbers(separated by comma) in the device file entries before the data of the last modification. These numbers are the major and minor device numbers.

Major and Minor Numbers

- 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 4, because they're all controlled by the same driver. **The kernel uses the major number at open time to dispatch execution to the appropriate driver**.
- The minor number is used by the driver to distinguish between the various hardware it controls. Returning to the example above, although all three devices are handled by the same driver they have unique minor numbers because the driver sees them as being different pieces of hardware.
- 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/skull c 12 2**. You don't have to put your device into /dev directory.