WHERE THE KERNEL MEETS THE HARDWARE

LINUX

DEVICE DRIVERS

Presented By,
Sailesh K

Apurva K

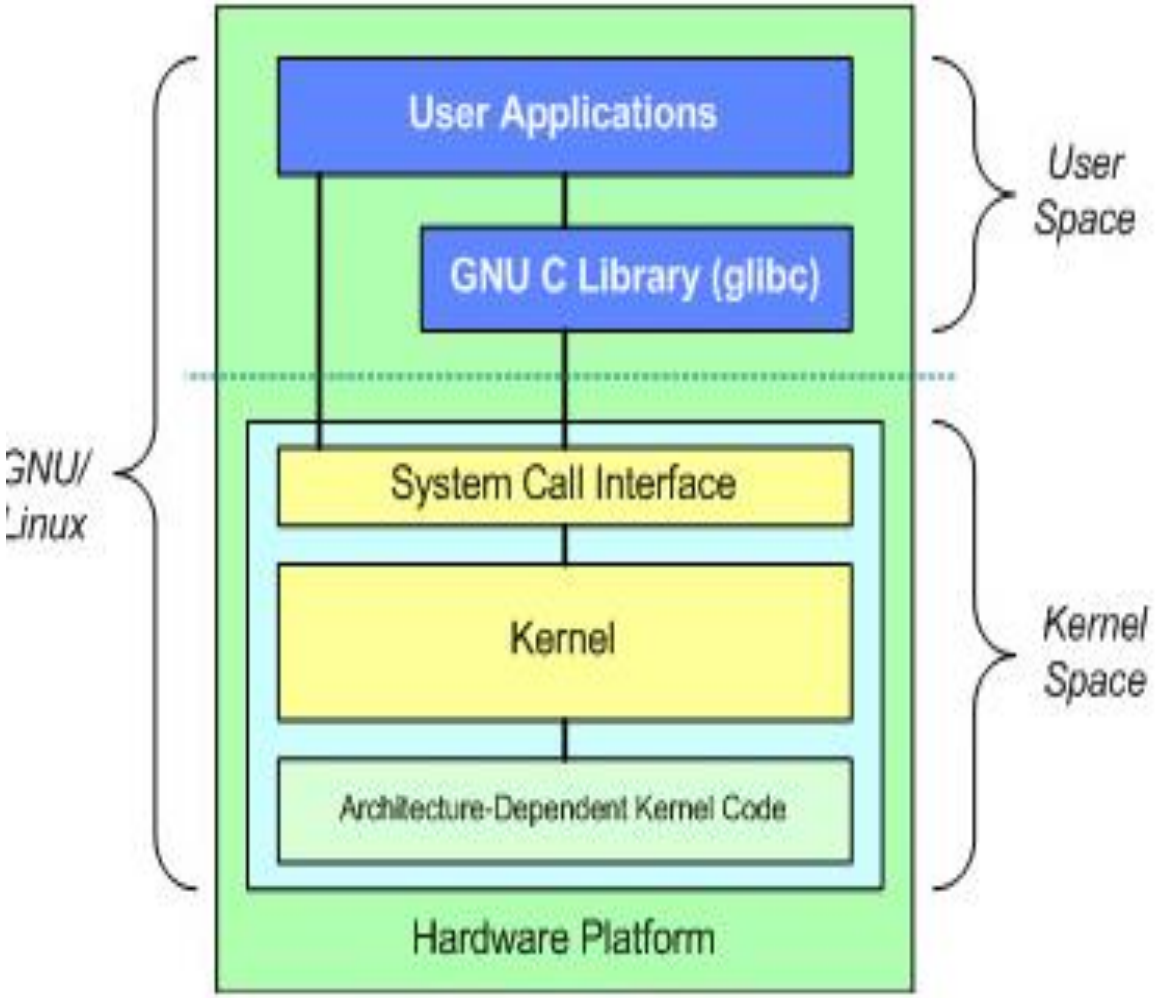
Aakash V

Agenda

- Linux Basic Introduction
  - Linux Architecture
  - Linux Kernel
- Device & Module
- Module Programming
  - Simple “ Hello World ” Module
  - Char Module
  - USB module
- Debugging Techniques
- Git

Linux Basic Introduction

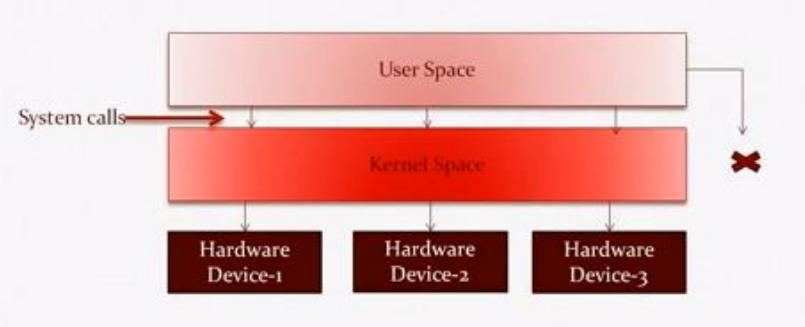
Linux Architecture

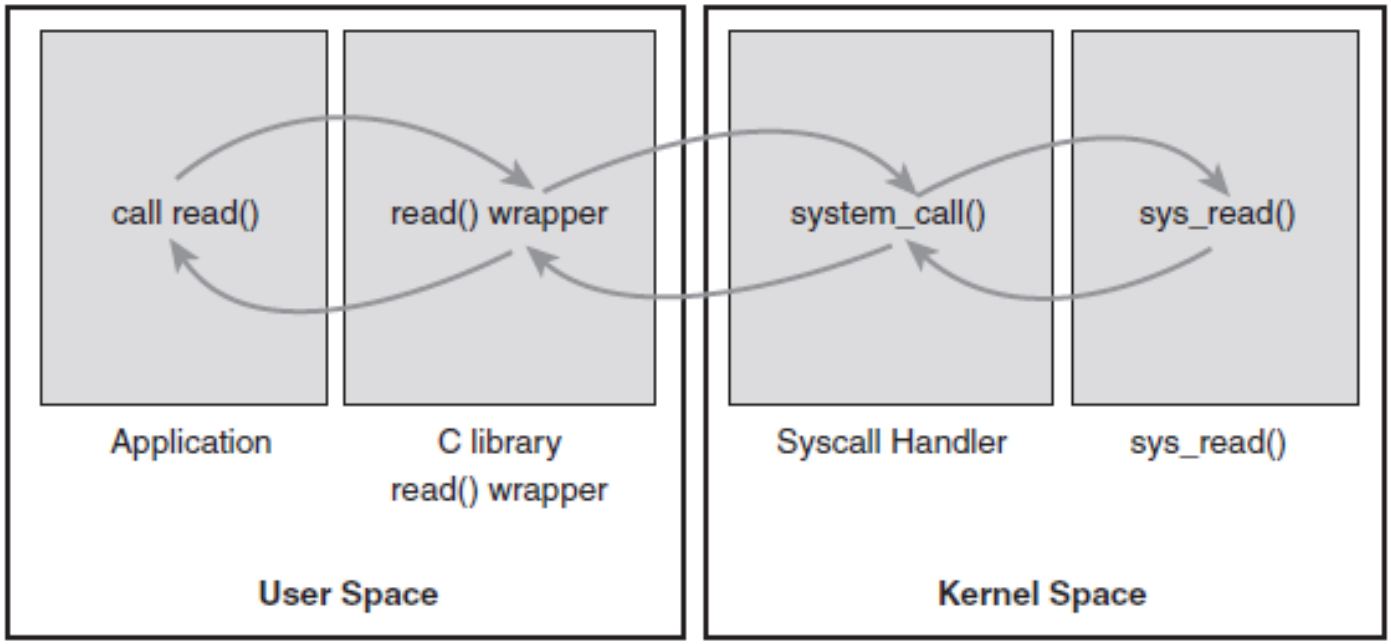


Kernel

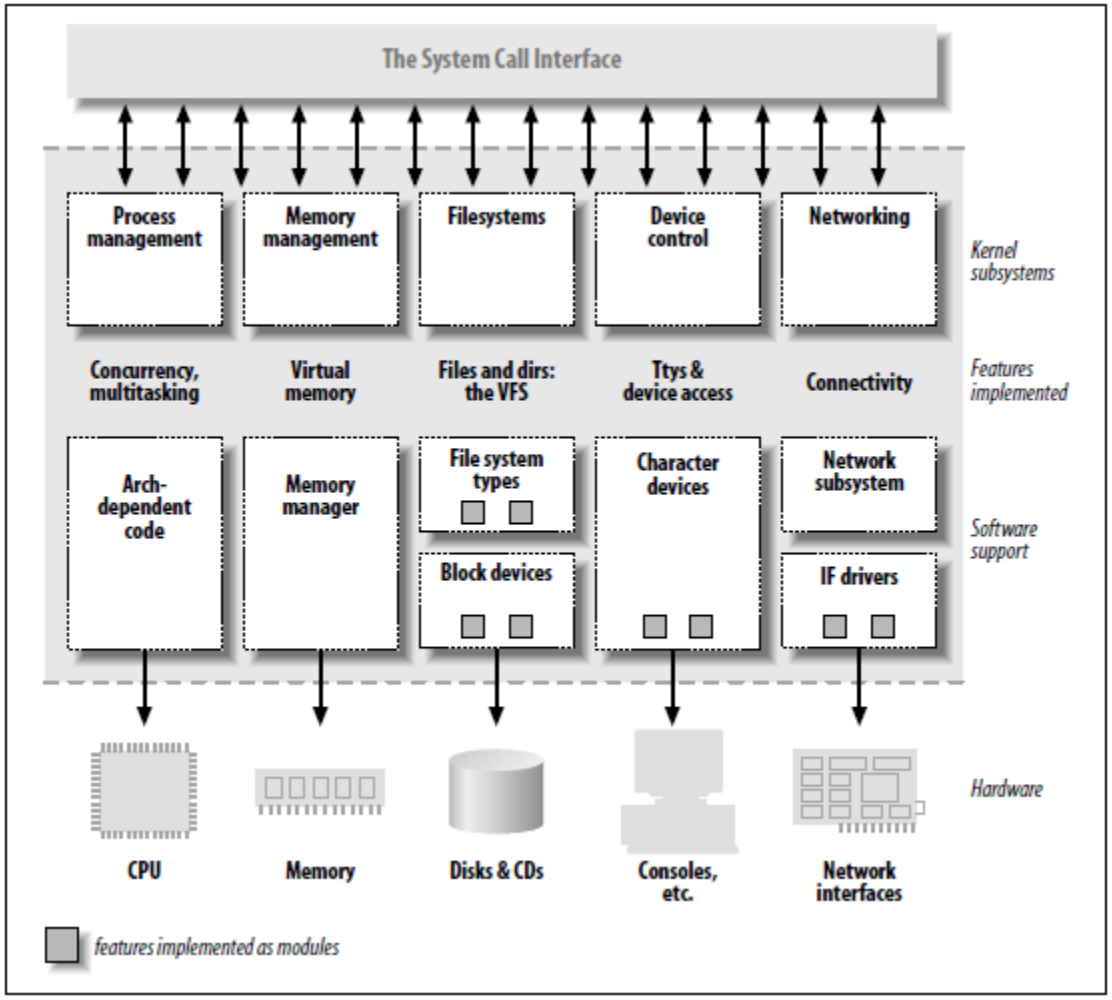
- The core part of Linux.
- Responsible for all major activities of this OS.
- Consists of various modules and interacts directly with the underlying hardware.
- Provides the required abstraction to hide low level hardware details to system or application programs.

Kernel space can be accessed by the processors only through use of system calls.





Split View of the Kernel



Kernel

1) Process Management

- Creating and destroying processes
- Communication between different processes (signals, pipes)
- Sharing of the CPU among different processes

2) Memory Management

- Memory is a very crucial component of a system
- Kernel helps in creation of virtual memory over the already present physical memory
- It builds up virtual address space and maintains the mapping

Kernel

3) File system

- Linux is based on a file system concept. Everything can be considered as a file.
- The Kernel builds a structural file system on top of a unstructured hardware.

4) Device Control

- Has the device drives for every peripheral preset on a system.

5) Networking

- Is responsible for communicating with the other systems in the network.
- Handles packets i.e. Managing the arrival and dispatch of network packets.

Linux Kernel Versions

Every software package used in Linux has a release number

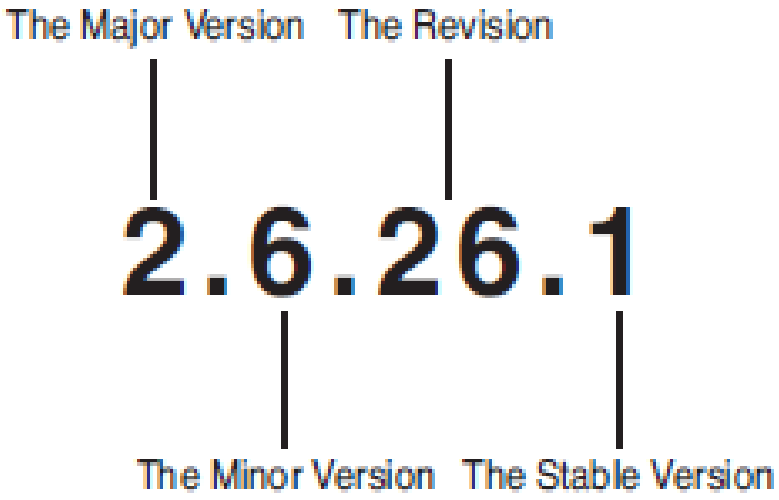
Linux Kernels come in two forms:

- Stable
- Development

Stable kernels - Production releases that is used for widespread deployment

Development Kernels - These are used to try and test out new features. They undergo quick changes.

Kernel version numbering



- Major Version – Major changes to the concept or code of Kernel
- Minor Version- Addition of new features or device drivers.

- Revision Number – Minor changes like addition of new feature or bug fixes.
- Stable version- Crucial Bug Fixes.

The type of the kernel depends on the minor number

- Stable kernels have a even minor number
- Development kernels have a odd minor number

Even numbered minor versions are stable for general use (e.g., 2.6.x)
Odd numbered minor versions are for developers only (e.g., 2.5.x)

Kernel Space programming

- Access to neither the C library nor the standard C headers.

The Kernel is not linked to the Standard C library (saves space and time)
We have to use functions that are defined with the kernel e.g. We use **printk()** instead of printf()

Printk()- works similar to printf(). Copies the formatted string to the kernel log buffer.
Can specify a priority flag also

- Coded in GNU C

The Linux Kernel is not coded strictly in ANSI C but using various language extensions available in GCC

GCC (GNU Compiler Collection) – consists of compilers for various different programming languages such as C, C++, Java etc

How does that affect Programming ?

- Presence of Inline Assembly

Embedding of assembly instructions with the normal C functions.
Can be used in parts of kernel that are unique to a given architecture

- Branch Optimization

Built in directive that optimizes conditional braches
Can use Macros ‘likely’ and ‘unlikely’
Helps in performance boost

- Use of inline functions

Inline functions to be used only for small and time critical codes

```
static inline void wolf(unsigned long tail_size)
```

They are preferred over Macros

- Lacks the memory protection afforded to user-space.

In User Mode, an attempt to write to a illegal memory is caught by the Kernel and the process is killed.
But if the same is performed in Kernel mode. The result can be unpredictable.
Kernel memory is not pageable
Hence we need to be careful while accessing memory in Kernel mode.

- Can't execute floating-point operations.

When the user space application perform Floating point instructions, the kernel initiates a integer to floating point mode transition.
But floating point instructions in kernel mode require manual saving and restoring of registers. Any error in doing so can cause unpredictable results.

- Small per-process fixed-size stack.

User Space has a large stack and can grow dynamically
However Kernel space stacks are small and have a fixed size
Can be approximately 8KB – 32 bit system
16KB – 64 bit system

- Synchronization and Concurrency

User applications are single threaded but kernel allows for concurrent use of shared resources.
Proper synchronization of different threads are necessary.

Linux is preemptive in nature.
- A process scheduler can schedule a process any time
- Two or more processors can access the same resource.

To avoid any kind of a deadlock or race condition, locks and semaphores must to used properly.

Devices

Devices

Linux divides devices into 3 broad categories.

- Character device
- Block device
- Network device

Character Device

- Can be accessed as a stream of bytes. (like a file system)
- Implement the basic calls as open , close , read and write .
- Access data sequentially.
- May not support file seek i.e. cannot jump backwards and forward.
- Examples
  - Text console
  - Serial ports

Block Device

- Permits the transfer of one or more blocks of Data
- The device has to be a random access device
- However in Linux, it is permitted for the Block devices to access data sequentially (one byte at a time) like the character devices, if required.
- Differs from Char devices only in the way data is managed internally by the kernel.
- Filesystems can be mount on a block device but not on a character device.
- Example – a disk

Network Interface

- All network transactions are made are made through a interface (i.e. a device capable of exchanging data with other hosts)
- Responsible for sending and receiving data packets .
- The task is driven by the network subsystem of the kernel.
- No knowledge of the connections. Its only purpose is to handle packets.
- Support protocols and streams related to packet transmission
- Not stream oriented. They do not map to a node in the file system.
- Communication between the kernel and network device driver is different from that of the char or block driver.

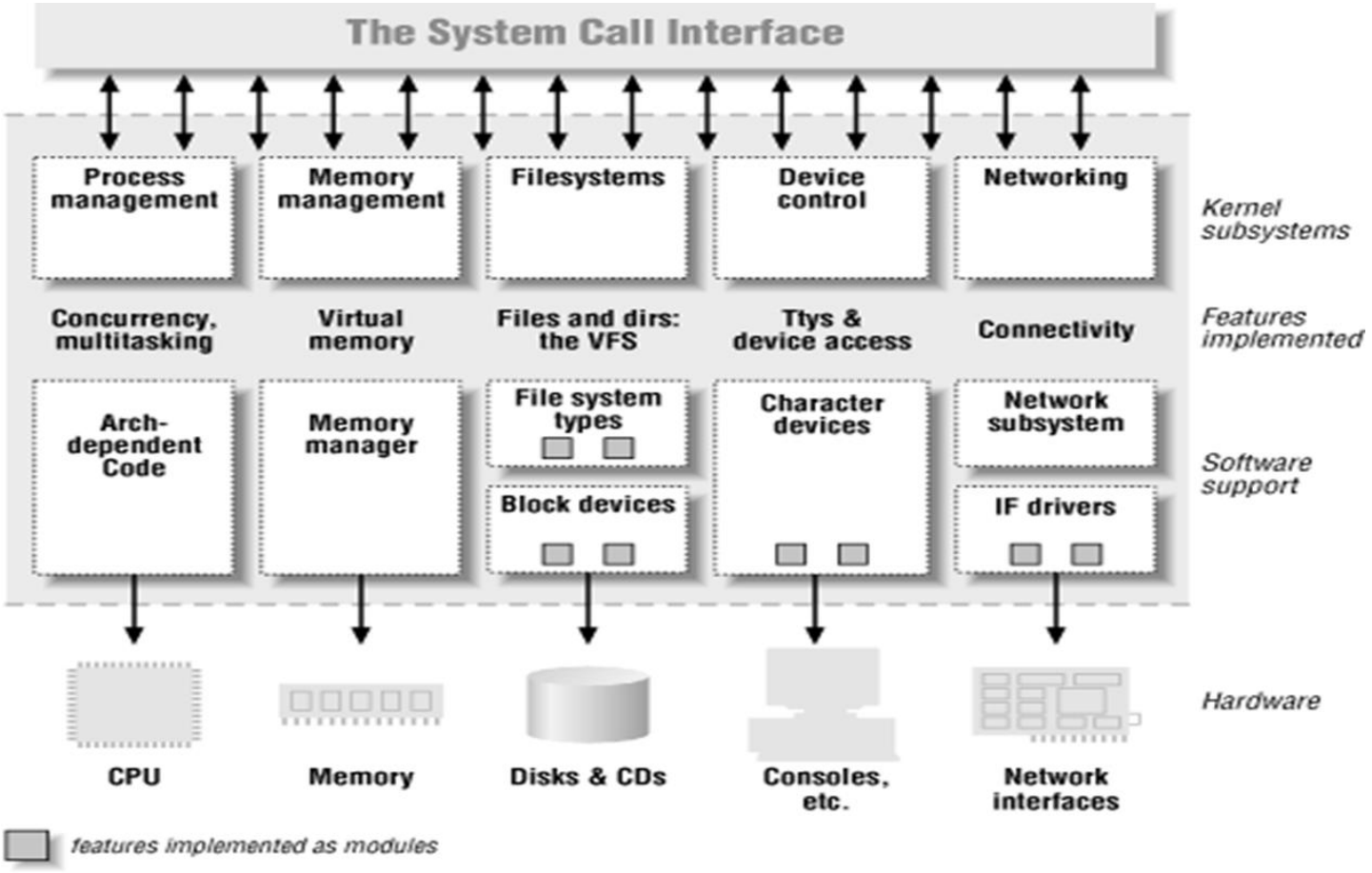
Device Driver

What is a device driver..?

- A software component that controls a hardware device
- A layer between hardware and user programs
  - defines how the device appears to user applications
  - there can be several drivers for a single hardware
  - a single driver can handle several similar devices
- Devices?
  - memory, disk, memory, CPU, ….
- A programming module with interfaces
  - Communication Medium between application/user and hardware

Role of device driver in OS

- The Role of a device driver is providing Mechanism, not Policy.



Module

Kernel Modules

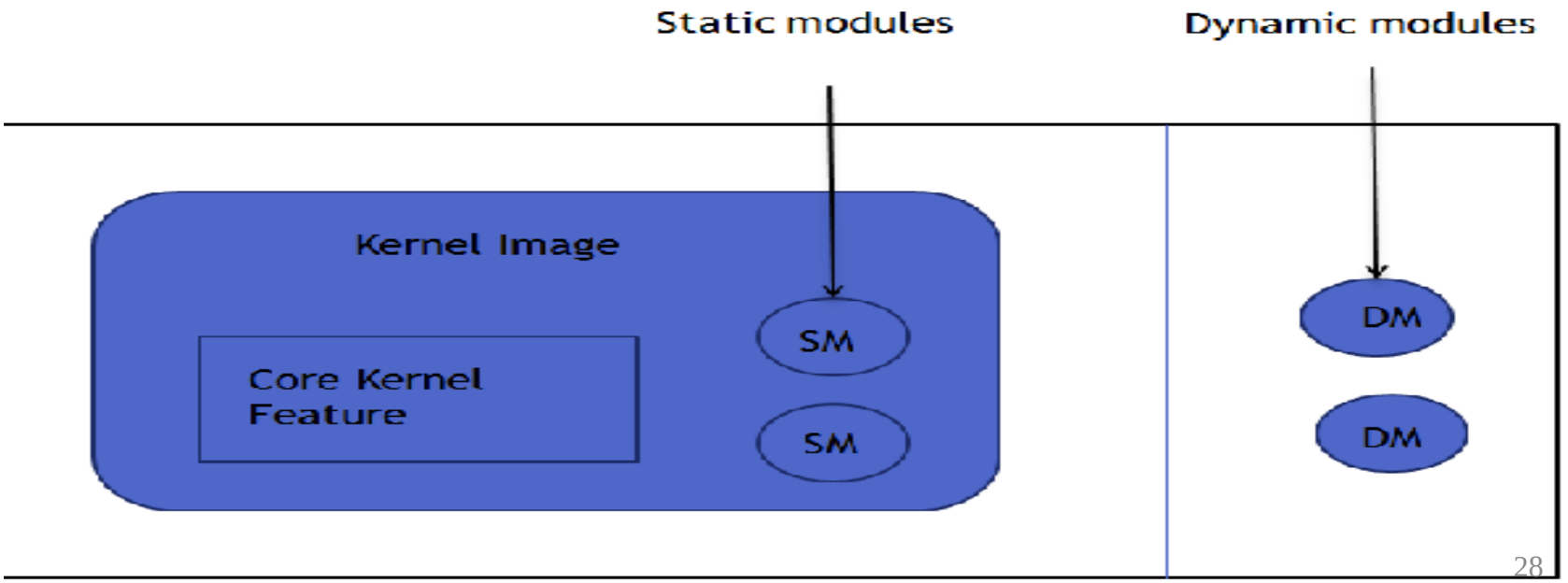
Classified in two types

- Static modules

  compiled as a part of the kernel and available at anytime.

- Dynamic modules

  can be loaded and unload into the kernel upon demand.



LKM Utilities cmd

- insmod

  Insert an LKM into the kernel.

- rmmod

  Remove an LKM from the kernel.

- depmod

  Determine interdependencies between LKMs.

- kerneld

  Kerneld daemon program

- ksyms

  Display symbols that are exported by the kernel for use by new LKMs.

LKM Utilities cmd

- lsmod

  List currently loaded LKMs.

- modinfo

  Display contents of .modinfo section in an LKM object file.

- modprobe
  - Insert or remove an LKM or set of LKMs intelligently. For example, if you must load A before loading B, Modprobe will automatically load A when you tell it to load B.

Hello World Module

```
#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;
}
```

Hello World Module (Cont.)


```
static void hello_exit(void){
printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);

module_exit(hello_exit);
```

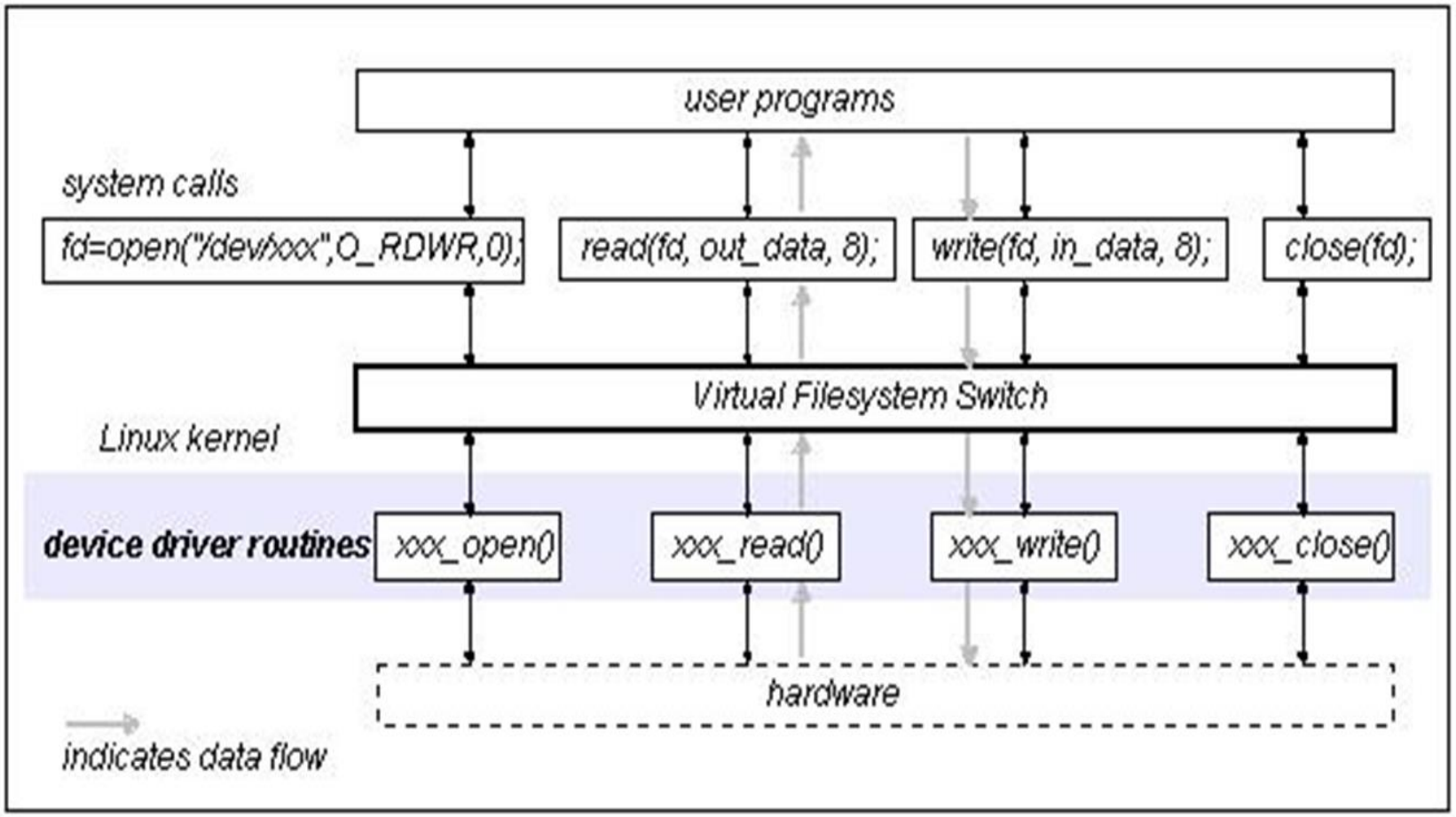
Makefile..

```
obj-m +=test.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
```

Device Driver Interface



Major and Minor Number

Major Number

- it's a number which attach a device with its driver.
- Kernel identifies each driver by major number.
- When we open the device or device file, kernel find the driver for that device by using major number.
- After that subsequent read and write etc. calls are handled by same driver.

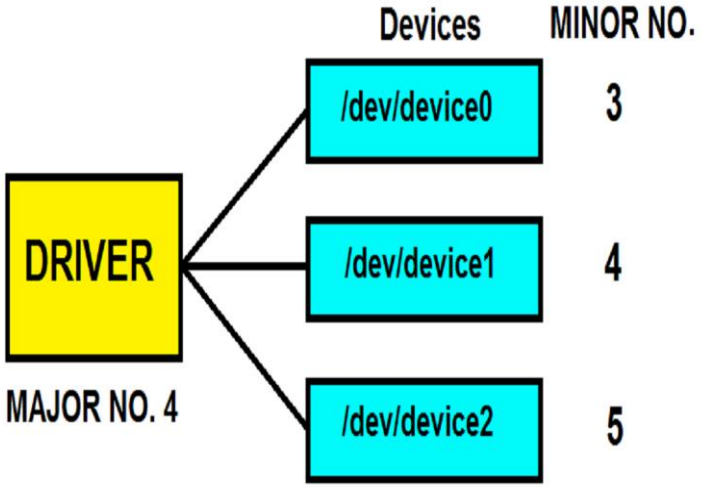
Minor Number

- Each device in system identified by a unique number. That number is called Minor number.

Note : Two or more device can have same major number but not minor number.

Internal Representation

- ls -l /dev
- Major is to Driver; Minor is to Device
- <linux/types.h> (>= 2.6.0)
- dev_t: 12 & 20 bits for major & m
- <linux/kdev_t.h>
- MAJOR(dev_t dev)
- MINOR(dev_t dev)
- MKDEV(int major, int minor)



Module Programming

Register and Unregister Device


```
/*used for all initialition stuff*/
int init_module(void){
        /* Register the character device */
        Major = register_chrdev(0,DEVICE_NAME,
                                &Fops);
}

/*used for a clean shutdown*/
void cleanup_module(void) {
    ret = unregister_chrdev(Major, DEVICE_NAME);
}
```

Loading Module..

- Compile
  - Makefile and Make
- Install the module
  - insmod
- List the module
  - lsmod
- If you let the system pick Major number, you can find the major number (for special creation) by
  - /proc/devices
- Make a special file
  - mknod /dev/device_name c major minor

Driver Generic Structure

1. Init and Exit Calls

```
static int hello_init(void){
 printk(KERN_ALERT "Hello world"); return 0;
}

static void hello_exit(void) {
 printk(KERN_ALERT "Goodby");
}

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

Driver Generic Structure

2. Fops Structure

```
//structure containing device operation
static struct file_operations fops=
{
      .read=dev_read,  //pointer to device read funtion
      .write=dev_write, //pointer to device write function
      .open=dev_open,   //pointer to device open function
      .release=dev_release, //pointer to device realese function
};
```

Driver Generic Structure

- Interrupt Routine

```
irqreturn_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
        printk("hi i am handling interrupt");
        return IRQ_HANDLED;
}
```

Navigation Tools

- CSCOPE
- CTAGS

Character Driver

- Write a character driver….

  Major Number=90

  Device name= mydev

  use the calls –

  copy_to_user (read section)

  copy_from_user (write section)

Create device node – mknod /dev/mydev c 90 1

1st sem Assignment

- Discussion

How to Debug at Kernel level?

Debugging Techniques

- By printing
  - Log levels
  - Syslog
  - dmesg
- Kernel debugger
  - gdb
  - kgdb

Loglevel Description

• KERN_EMERG	An emergency condition; the system is probably dead.
• KERN_ALERT	A problem that requires immediate attention.
• KERN_CRIT	A critical condition.
• KERN_ERR	An error.
• KERN_WARNING	A warning.
• KERN_NOTICE	A normal, but perhaps noteworthy, condition.
• KERN_INFO	An informational message.
• KERN_DEBUG	A debug message—typically superfluous.

Syslog

Kernel debugger

- gdb
- kgdb

Universal Serial Bus

Universal Serial Bus

- Technology that allows a person to connect an electronic device to a computer
- Fast serial bus
- The standard was made to improve plug and play devices

USB transfer speed

- LowSpeed: up to 1.5 Mbps

  Since USB 1.0
- FullSpeed: up to 12 Mbps

  Since USB 1.1
- HiSpeed:

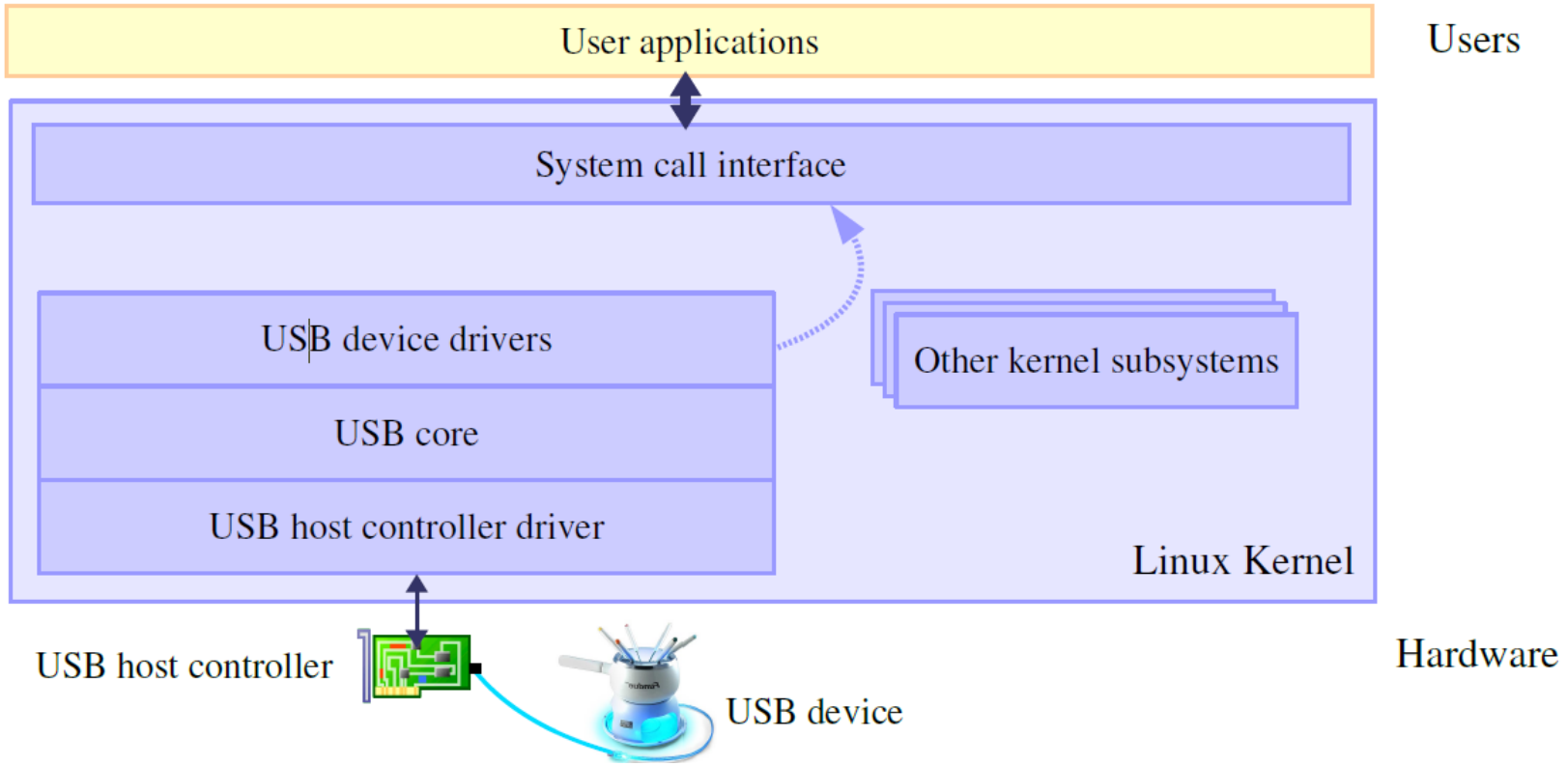
  USB 2.0 - up to 480 Mbps

  USB 3.0 - up to 5 Gbps

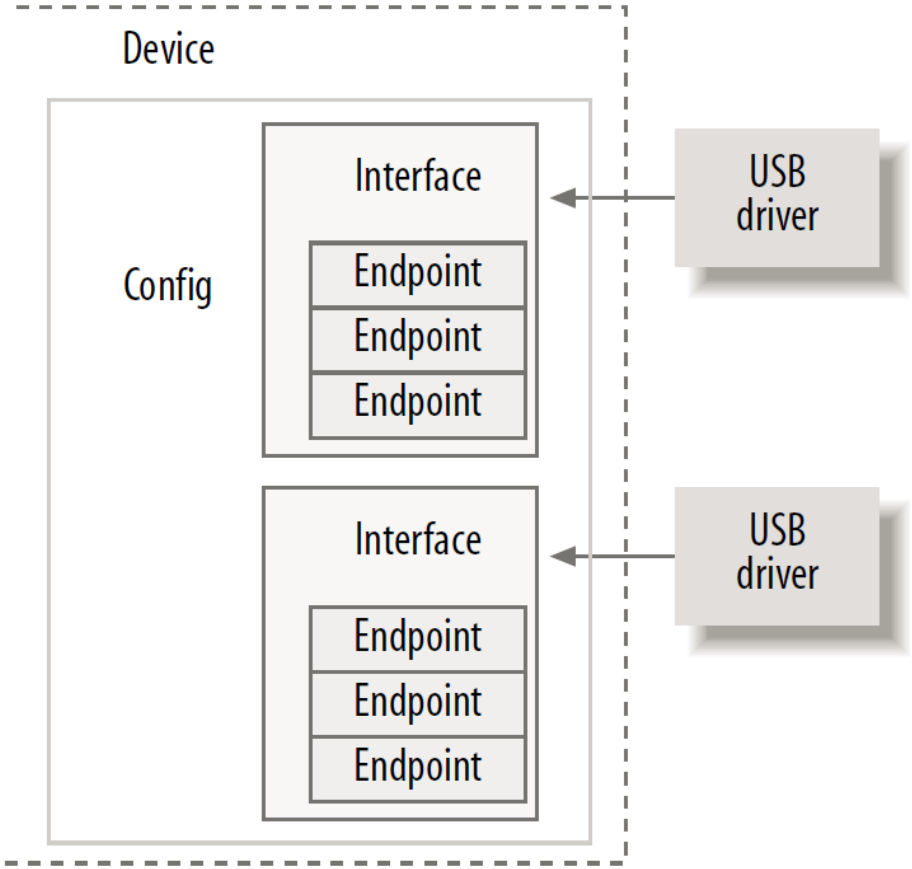
USB Device Classes

Device Class	Example Device
Display	Monitor
Communication	Modem
Audio	Speakers
Mass storage	Hard drive
Human interface	Data glove

USB driver overview



USB module overview



USB device descriptors

Operating system independent. Described in the USB specification

- Device - Represent the devices connected to the USB bus.

  Ex : USB speaker with volume control buttons.
- Configurations - Represent the state of the device.

  Ex : Active, Standby, Initialization
- Interfaces - Logical devices.

  Ex : speaker, volume control buttons.
- Endpoints - Unidirectional communication pipes.

  Either IN (device to computer) or OUT (computer to device).

Control endpoints

- Used to configure the device, get information about it, send commands to it, retrieve status information.
- Simple, small data transfers.
- Every device has a control endpoint (endpoint 0), used to configure the device at insertion time.
- The USB protocol guarantees that the corresponding data transfers will always have enough (reserved) bandwidth.

Interrupt endpoints

- Transfer small amounts of data at a fixed rate each time the hosts asks the device for data.
- Guaranteed, reserved bandwidth.
- For devices requiring guaranteed response time, such as USB mice and keyboards.
- Note: different than hardware interrupts. Require constant polling from the host.

Isochronous endpoints

- Also for large amounts of data.
- Guaranteed speed (often but not necessarily as fast as possible).
- No guarantee that all data makes it through.
- Used by realtime data transfers (typically audio and video).

Bulk endpoints

- Large sporadic data transfers using all remaining available bandwidth.
- No guarantee on bandwidth or latency.
- Guarantee that no data is lost.
- Typically used for printers, storage or network devices.

Interfaces

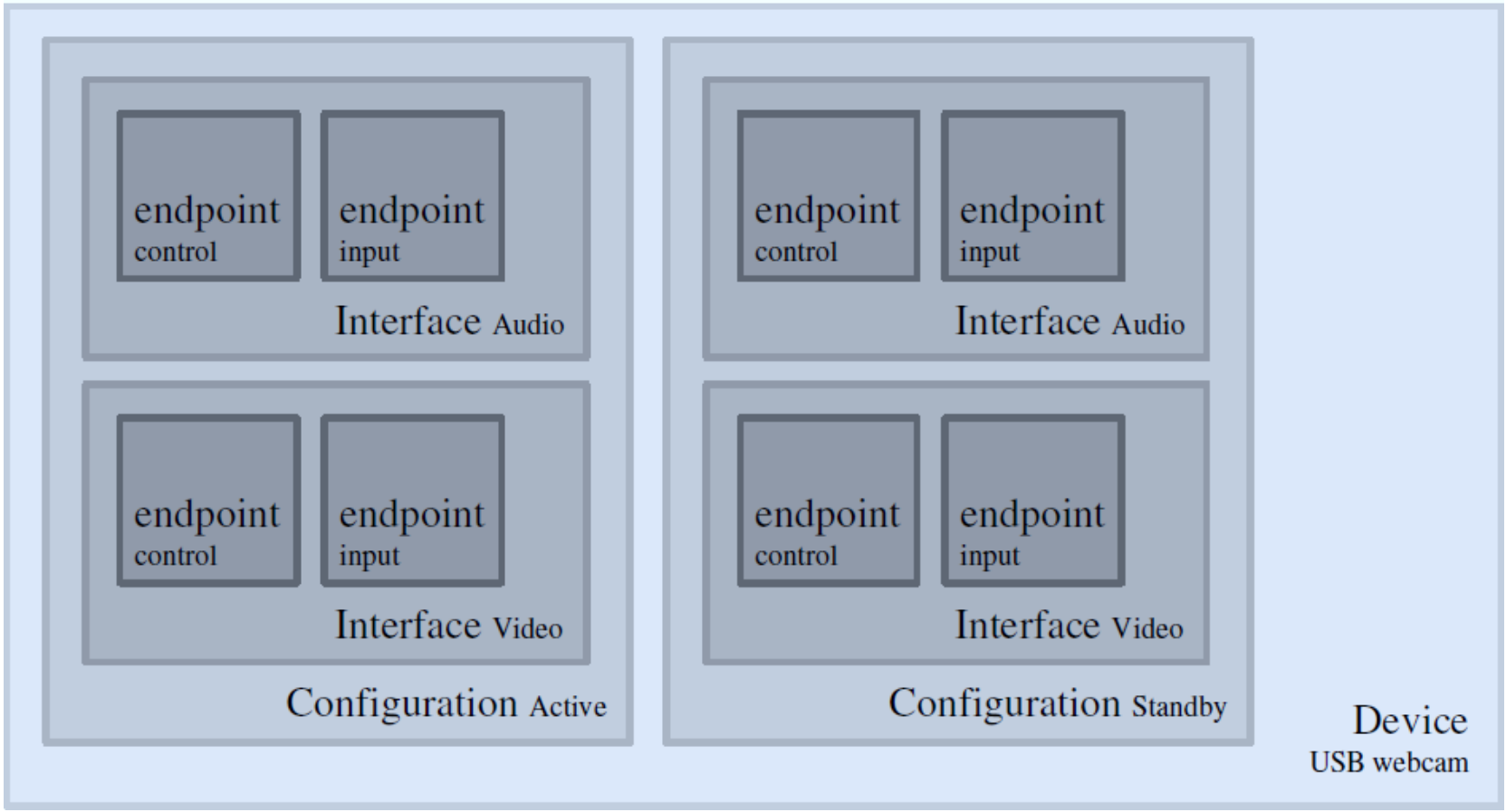
- Each interface encapsulates a single highlevel function (USB logical connection). Example (USB webcam): video stream, audio stream, keyboard (control buttons).
- One driver is needed for each interface!
- 1 or more interfaces per device

Configurations

- Configurations represent the state of the device.

  Examples: Active, Standby, Initialization

USB device overview



USB devices - Summary

Hierarchy: device -> configurations -> interfaces -> endpoints

4 different types of endpoints

- Control : device control, accessing information, small transfers. Guaranteed bandwidth.
- Interrupt (keyboards, mice...): data transfer at a fixed rate. Guaranteed bandwidth.
- Bulk (storage, network, printers...): use all remaining bandwidth. No bandwidth or latency guarantee.
- Isochronous (audio, video...): Guaranteed speed. Possible data loss.

usbtree

- cat /proc/bus/pci/devices

Driver registration

```
static struct usb_driver pen_driver ={
    .name = "pen_driver",
    .probe = pen_probe,
    .disconnect = pen_disconnect,
    .id_table = pen_table,
};
```

Driver registration

```
static int __init pen_init(void)
{
    return usb_register(&pen_driver);
}module_init(pen_init);
```

Driver unregistration

```
static void __exit pen_exit(void)
{
    usb_deregister(&pen_driver);
}module_exit(pen_exit);
```

Table of supported USB devices


```
static struct usb_device_id pen_table[] =
{
    { USB_DEVICE(0x054c, 0x05ba) },
    {} /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, pen_table);
```

Makefile

```
obj-m := my_usb.o

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

all:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
	rm -rf *.o *.ko *.mod.* *.symwers *.order *-
```

USB module driver

- Demo

Git

- Open source distributed version control system that facilitates GitHub activities on your laptop or desktop.

- Quick view of commands and its working
  - Demo

Create Project


```
$ git init [project-name]
```

Creates a new local repository with the specified name

```
$ git clone [url]
```

Downloads a project and its entire version history

Make changes


```
$ git status
```

Lists all new or modified files to be committed

```
$ git diff
```

Shows file differences not yet staged

```
$ git add [file]
```

Snapshots the file in preparation for versioning

```
$ git diff --staged
```

Shows file differences between staging and the last file version

```
$ git reset [file]
```

Unstages the file, but preserve its contents

```
$ git commit -m "[descriptive message]"
```

Group changes

$ git branch

Lists all local branches in the current repository

$ git branch [branch-name]

Creates a new branch

$ git checkout [branch-name]

Switches to the specified branch and updates the working directory

$ git merge [branch]

Combines the specified branch's history into the current branch

$ git branch -d [branch-name]

Review History


```
$ git log
```

Lists version history for the current branch

```
$ git log --follow [file]
```

Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```

Shows content differences between two branches

```
$ git show [commit]
```

Outputs metadata and content changes of the specified commit

Any Question ?

THANK YOU