Linux Device Drivers

rahul.batra.it@gmail.com

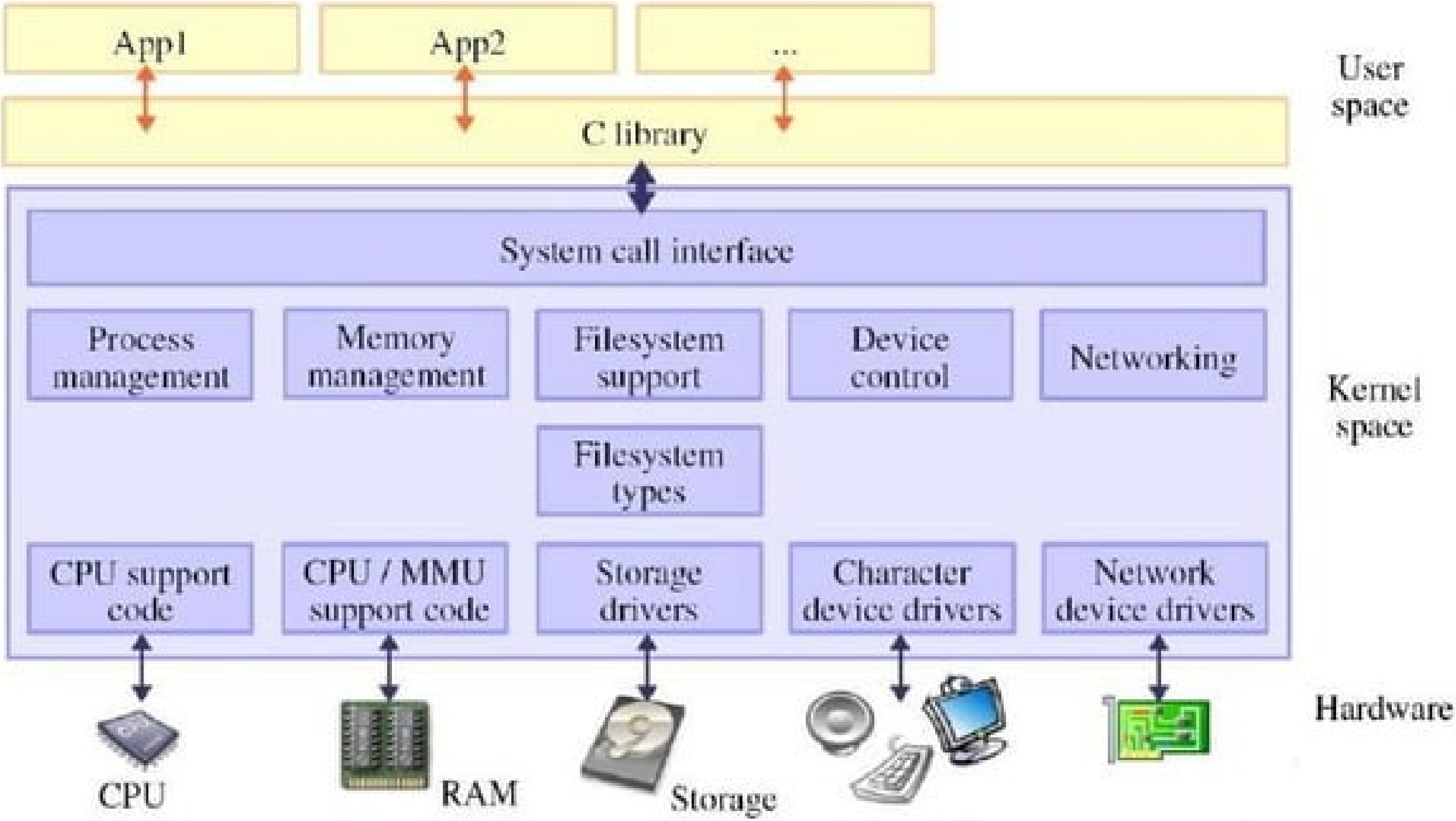
What is a device driver?

- A software component that controls a hardware device
  - interacts with user programs
  - better provide basic hardware controls only
    - leave high level decision to user programs
    - e.g.) floppy driver provides only a view of bytes sequence
- 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, ….

Device Driver

- A programming module with interfaces
  - Communication Medium between application/user and hardware
- In Unix,
  - Kernel module
  - device driver interface = file interface
  - What are normal operations?
  - Block vs. character

Kernel View



App1
App2
...
User space
C library
System call interface
Process management
Memory management
Filesystem support
Device control
Networking
Kernel space
Filesystem types
CPU support code
CPU / MMU support code
Storage drivers
Character device drivers
Network device drivers
Hardware
CPU
RAM
Storage

Kernel Components

Process management

– Creating and destroying processes

– Handling their connection to the outside world (input and output).

– Communication among different processes (through signals, pipes, or Inter Process communication primitives)

– Scheduling

Memory management

– Provides virtual addressing space for any and all processes

– The different parts of the kernel interact with the memory-management subsystem

Kernel Components

File systems

– Unix is heavily based on the file system concept; almost everything in Unix can be treated as a file.

– The kernel builds a structured file system on top of unstructured hardware.

– The resulting file abstraction is heavily used throughout the whole system.

– In addition, Linux supports multiple file system types, that is, different ways of organizing data on the physical medium.

Kernel Components

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 a *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.

Kernel Components

Networking

– Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events.

– The packets must be collected, identified, and dispatched before a process takes care of them.

– The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity.

– Routing and address resolution are implemented in the Kernel

What are device drivers in Linux?

- From the User's point of view: files
- From the Kernel's point of view:
  a set of VFS functions (read, write, open)
  plus some register functions
- Part of the Kernel -> run in kernel mode
- Either loadable or statically build in the kernel

Two different kinds of access:

- Sequential and random
- char and block devices

Linux Device Drivers

- A set of API subroutines (typically system calls) interface to hardware
- Hide implementation and hardware-specific details from a user program
- Typically use a file interface metaphor
- Device is a special file
- Manage data flow between a user program and devices
- A self-contained component (add/remove from kernel)
- A user can access the device via file name in /dev , e.g. /dev/lp0

Kernel Modules Versus Applications

- ***Modules are event-driven***
  - Every kernel module registers itself in order to serve future requests
  - It's initialization function terminates immediately
  - Exit function of a module must carefully undo everything the init function built up

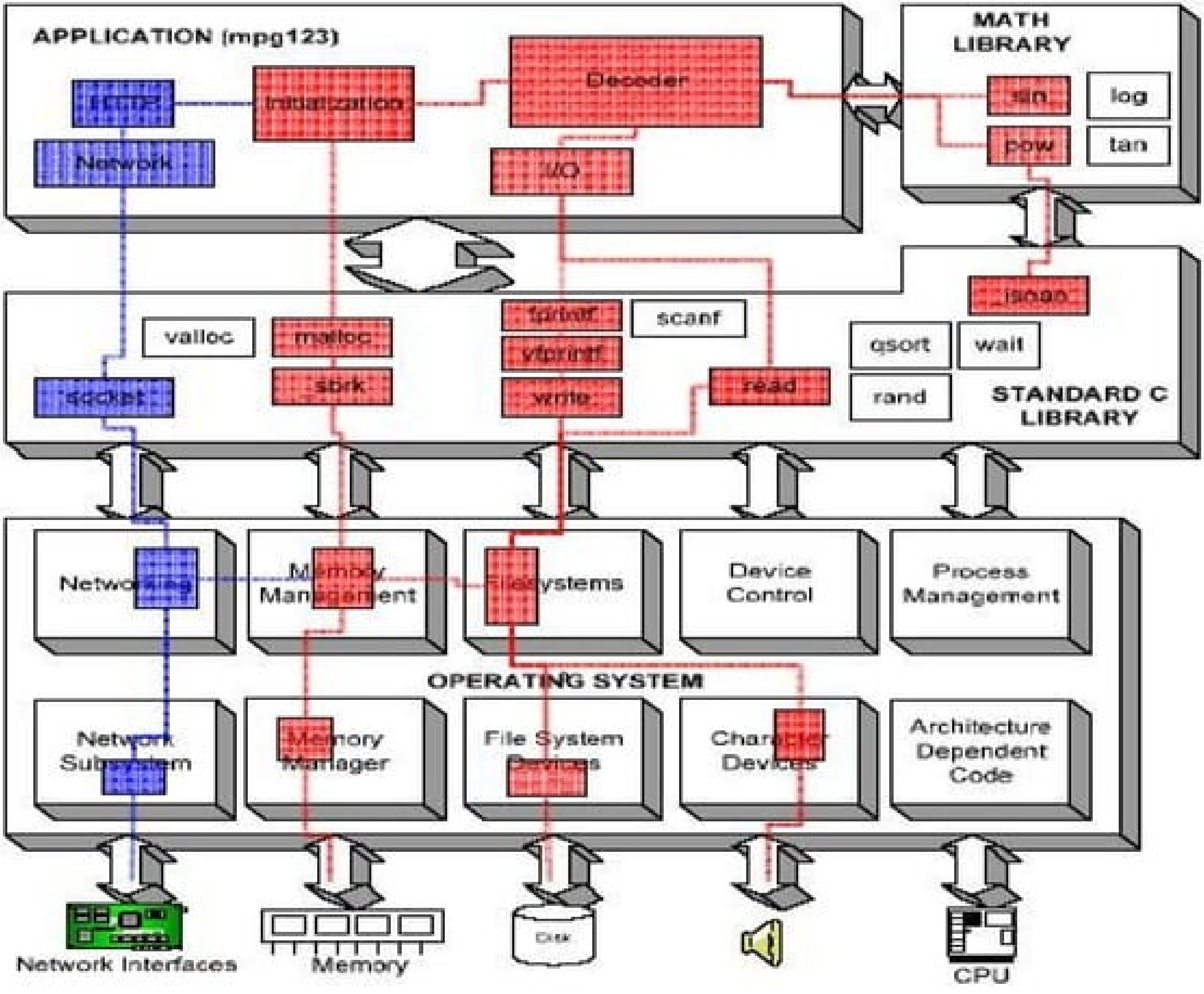
- ***User-level applications can call functions they don't define***
  - Linking stage resolves external references using libraries
- ***Module is linked only to the kernel,***

the only functions it can call are the ones exported by the kernel

– *No libraries to link to*

– Example: *printk* is the version of *printf* defined within the kernel and exported to the modules

- ***Don't include typical user-level header files*** (like <stdio.h>, etc.)
  - Only functions that are actually part of the kernel may be used in kernel modules
  - Anything related to the kernel is declared in headers found in the kernel source tree



APPLICATION (mpg123)
Initialization
Decoder
Network
I/O
MATH LIBRARY
sin
log
pow
tan
valloc
malloc
sbrk
fprintf
vfprintf
write
scanf
read
socket
qsort
wait
rand
STANDARD C LIBRARY
Networking
Memory Management
Filesystems
Device Control
Process Management
OPERATING SYSTEM
Network Subsystem
Memory Manager
File System Devices
Character Devices
Architecture Dependent Code
Network Interfaces
Memory
CPU

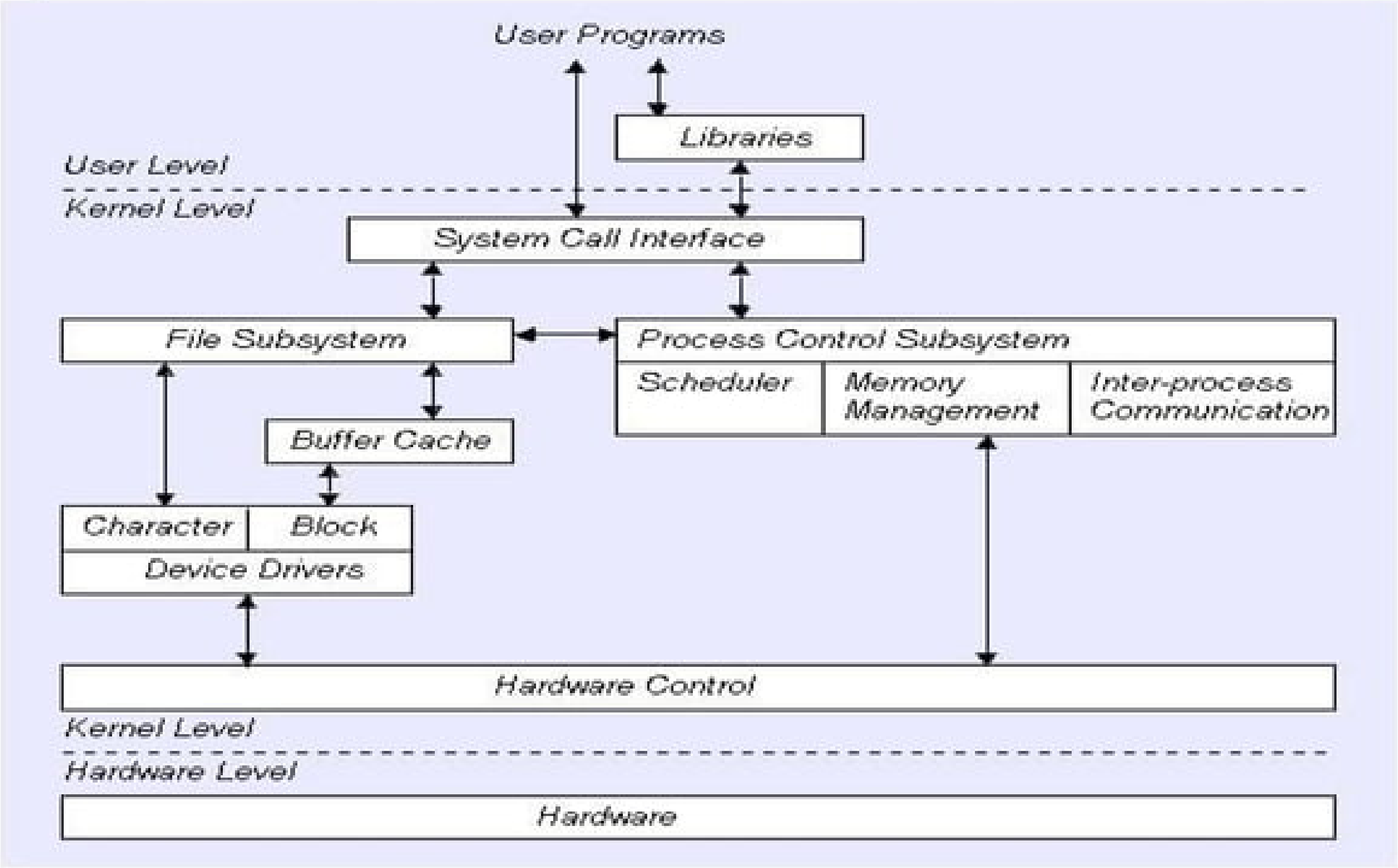
User Space Versus Kernel Space

• ***Module*** runs in **kernel space**,
whereas ***Applications*** run in ***user space***

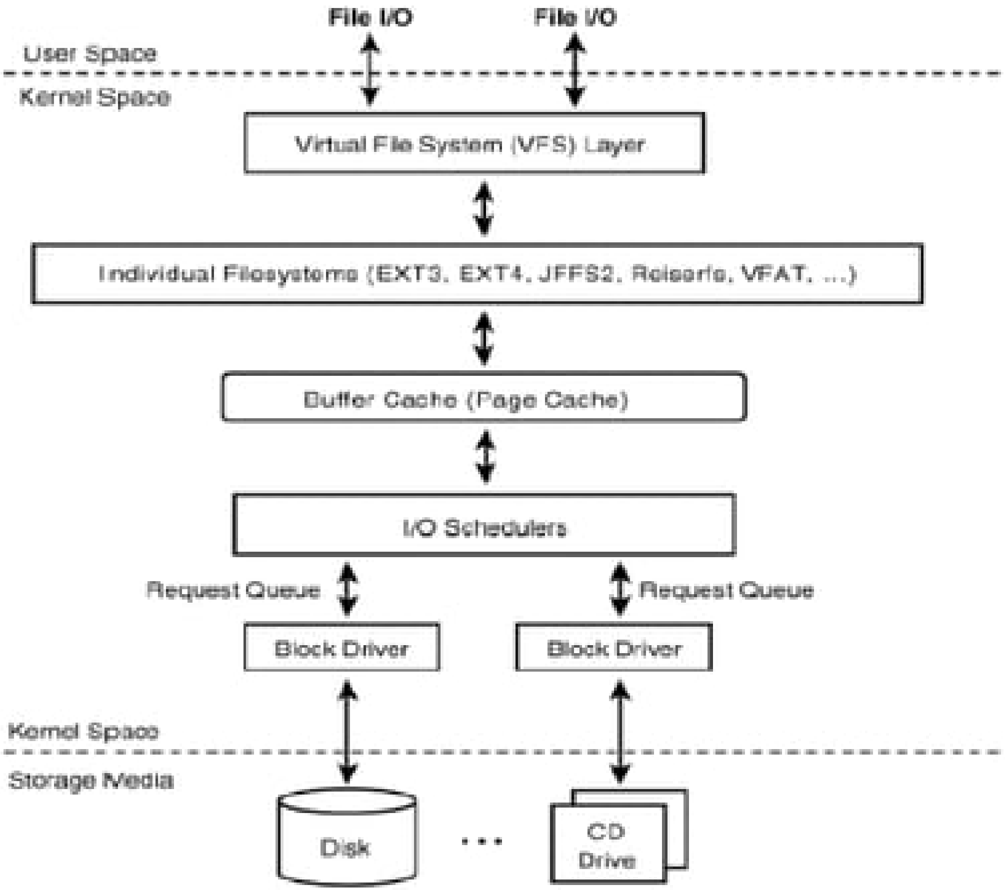
• Unix transfers execution from user space to kernel space whenever an application issues a system call or is suspended by a hardware interrupt

• Role of a module is to extend kernel functionality
– Some functions in the module are executed as part of system calls
– Some are in charge of interrupt handling
• Linux driver code must be reentrant
– Must be capable of running in more than one context at the same time
– Must avoid race conditions
=> Linux 2.6 is a preemptive kernel!

User program & Kernel interface



Block I/O on Linux



Device Driver Types

Character Devices

– Accessed as a stream of bytes (like a file)
– Typically just data channels, which allow only sequential access

Some char devices look like data areas and allow moving back and forth in them (example: frame grabbers)
– A char driver is in charge of implementing this behavior
– Char devices are accessed by means of file system nodes

Example: */dev/tty1* and */dev/lp0*
– Driver needs to implement at least the *open*, *close*, *read*, and *write* system calls

Examples:
– Text console (*/dev/console*)
– Serial ports (*/dev/ttyS0*)

Device Driver Types

Block Devices

– In some Unix systems, block devices can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of 2).

Linux, instead, allows applications to read and write a block device like a char device

– Like char devices accessed through file system nodes in the /dev directory

– Char and block devices differ in the kernel/driver interface

– Difference between char and block devices is transparent to users

Device Driver Types

Network Interfaces

– Network transactions made through an interface
  - Hardware device
  - Pure software device (loop back)

– Network interfaces usually designed around the transmission and receipt of packets
  - Network driver knows nothing about individual connections; it only handles packets

– Char device? Block device?
  - not easily mapped to file system nodes
  - Network interfaces don't have entries in the file system
  - Communication between the kernel and network device driver completely different from that used with char and block drivers

Block Versus Character devices

```
# cat /proc/devices
Character devices:              Block devices:
  1 mem                           2 fd
  2 pty                           3 ide0
  3 ttyp                          8 sd
  4 ttyS                         22 ide1
  5 cua                          65 sd
  6 lp                           66 sd
  7 vcs
 10 misc
 14 sound
128 ptm
136 pts
162 raw
180 usb
```

Loadable Kernel Module (LKM)

- A new kernel module can be added on the fly (while OS is still running)
- LKMs are often called “kernel modules”
- They are not user program

Types of LKM

- Device drivers
- File system driver (one for ext2, MSDOS FAT16, 32, NFS)
- System calls
- Network Drivers
- TTY line disciplines. special terminal devices.
- Executable interpreters.

Basic LKM (program)

- Every LKM consist of two basic functions (minimum) :

```
int init_module(void)                /*used for all initialition stuff*/
{
...
}
void cleanup_module(void)            /*used for a clean shutdown*/
{
...
}
```

- Loading a module - normally restricted to root - is managed by issuing the following command: **# insmod module.o**

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.
- **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.

Common LKM util cmd

- Create a special device file

  % mknode /dev/driver c 40 0

- Insert a new module

  % insmod modname

- Remove a module

  %rmmod modname

- List module

  % lsmod or

  % more /proc/modules

audio	37840	0
cmpci	24544	0
soundcore	4208	4 [audio cmpci]
nfsd	70464	8 (autoclean)

General implementation steps

1. Understand the device characteristic and supported commands.
2. Map device specific operations to UNIX file operation
3. Select the device name (user interface)
   - Namespace (2-3 characters, /dev/lp0)
4. (optional) select a major number and minor (a device special file creation) for VFS interface
   - Mapping the number to right device sub-routines
5. Implement file interface subroutines
6. Compile the device driver
7. Install the device driver module with loadable kernel module (LKM)
8. or Rebuild (compile) the kernel

Read/write (I/O)

- IO-Operations have unpredictable termination time
  -waiting for positioning the head of a hard disk
  -waiting for keyboard input
- Two strategies
  - **-polling mode**
  - **-interrupt mode**

Polling mode

- To poll (befragen)
- Read the status register repeatedly until it changes

    -> spin locks (busy waits)
- Inefficient, if duration in the order of milliseconds

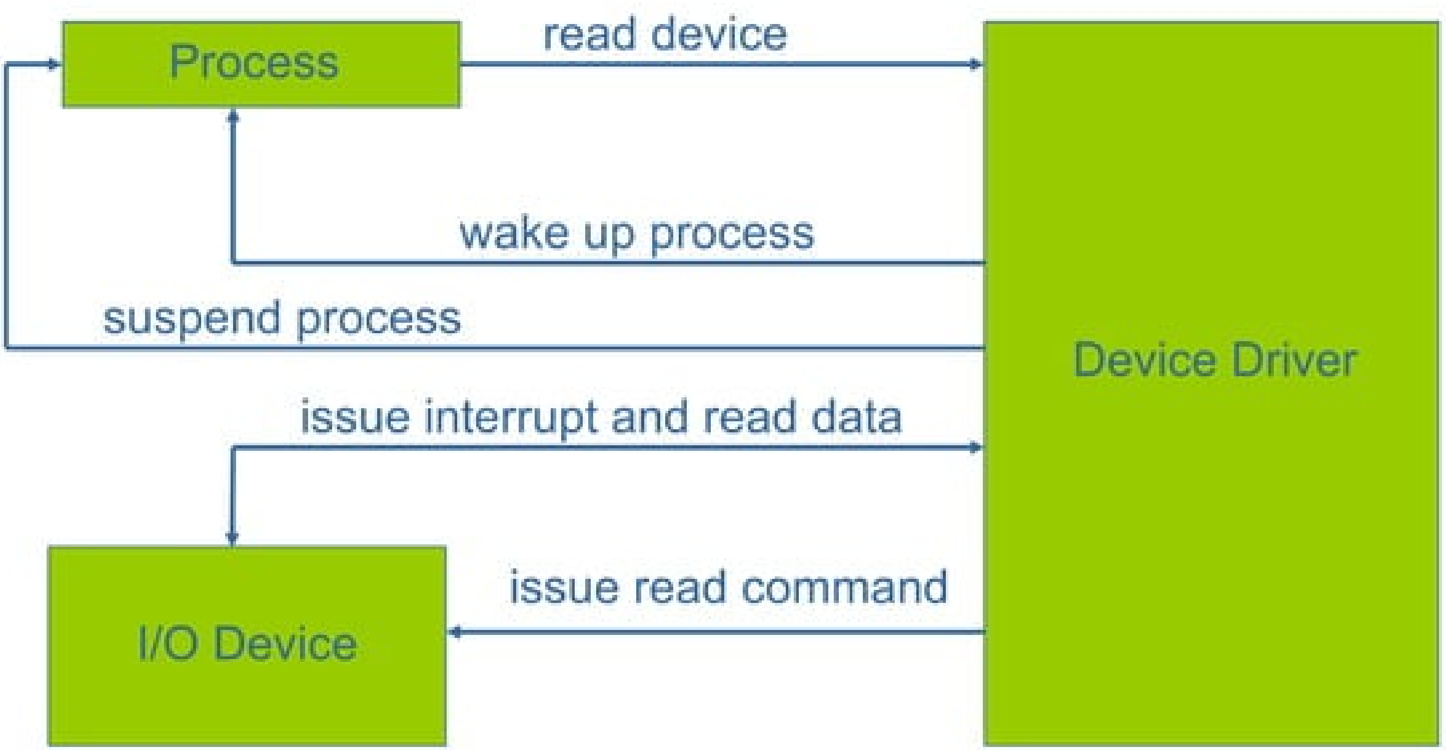
    schedule inside the loop

    interrupt mode
- I/O-Controller not capable of signaling
- Fastest way to communicate with hardware

Interrupt mode

- An interrupt handling routine is registered with the kernel
- After triggering the operation, process is suspended
- when finished, an interrupt is issued
- process is awaken

Interrupt mode



Interrupt mode

```
# cat /proc/interrupts
            CPU0
   0:     150146        XT-PIC  timer
   1:       3451        XT-PIC  keyboard
   2:          0        XT-PIC  cascade
   5:          0        XT-PIC  via82cxxx
   9:          0        XT-PIC  acpi
  10:       1806        XT-PIC  usb-uhci, usb-uhci, usb-uhci, eth0
  12:      19080        XT-PIC  PS/2 Mouse
  14:      89841        XT-PIC  ide0
  15:          6        XT-PIC  ide1
 NMI:          0
 LOC:     145850
 ERR:         84
 MIS:          0
```

Device special file

- Device number
  - Major (used to VFS mapping to right functions)
  - Minor (sub-devices)
- mknod /dev/stk c 38 0
- ls –l /dev/tty
  - crw-rw-rw-   1 root   root   5,   0 Apr 21 18:33 /dev/tty

Register and unregister device

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

}
void cleanup_module(void)                    /*used for a clean shutdown*/

    {ret = unregister_chrdev(Major, DEVICE_NAME);

...
}
```

Register and unregister device

- compile

  -Wall -DMODULE -D__KERNEL__ -DLINUX –DDEBUG -I /usr/include/linux/version.h -I/lib/modules/`uname -r`/build/include
- Install the module

  %insmod module.o
- List the module

  %lsmod
- If you let the system pick Major number, you can find the major number (for special creation) by

  % more /proc/devices
- Make a special file

  % mknod /dev/device_name c major minor

Implementation

- Assuming that your device name is Xxx
- Xxx_init() initialize the device when OS is booted
- Xxx_open() open a device
- Xxx_read() read from kernel memory
- Xxx_write() write
- Xxx_release() clean-up (close)
- init_module()
- cleanup_module()

kernel functions

- **add_timer()**
  - Causes a function to be executed when a given amount of time has passed
- **cli()**
  - Prevents interrupts from being acknowledged
- **end_request()**
  - Called when a request has been satisfied or aborted
- **free_irq()**
  - Frees an IRQ previously acquired with request_irq() or irqaction()
- **get_user*()**
  - Allows a driver to access data in user space, a memory area distinct from the kernel
- **inb(), inb_p()**
  - Reads a byte from a port. Here, inb() goes as fast as it can, while inb_p() pauses before returning.
- **irqaction()**
  - Registers an interrupt like a signal.
- **IS_*(inode)**
  - Tests if inode is on a file system mounted with the corresponding flag.
- **kfree*()**
  - Frees memory previously allocated with kmalloc()
- **kmalloc()**
  - Allocates a chu nk of memory no larger than 4096 bytes.
- **MAJOR()**
  - Reports the major device number for a device.
- **MINOR()**
  - Reports the minor device number for a device.

kernel functions

- **memcpy_*fs()**
  - Copies chunks of memory between user space and kernel space
- **outb(), outb_p()**
  - Writes a byte to a port. Here, outb() goes as fast as it can, while outb_p() pauses before returning.
- **printk()**
  - A version of printf() for the kernel.
- **put_user*()**
  - Allows a driver to write data in user space.
- **register_*dev()**
  - Registers a device with the kernel.
- **request_irq()**
  - Requests an IRQ from the kernel, and, if successful, installs an IRQ interrupt handler.
- **select_wait()**
  - Adds a process to the proper select_wait queue.
- ***sleep_on()**
  - Sleeps on an event, puts a wait_queue entry in the list so that the process can be awakened on that event.
- **sti()**
  - Allows interrupts to be acknowledged.
- **sys_get*()**
  - System calls used to get information regarding the process, user, or group.
- **wake_up*()**
  - Wakes up a process that has been put to sleep by the matching *sleep_on() function.

Pitfalls

– **Using standard libraries**: can only use kernel functions, which are the functions you can see in /proc/ksyms.

- **Disabling interrupts** You might need to do this for a short time and that is OK, but if you don't enable them afterwards, your system will be stuck
- Changes from version to version