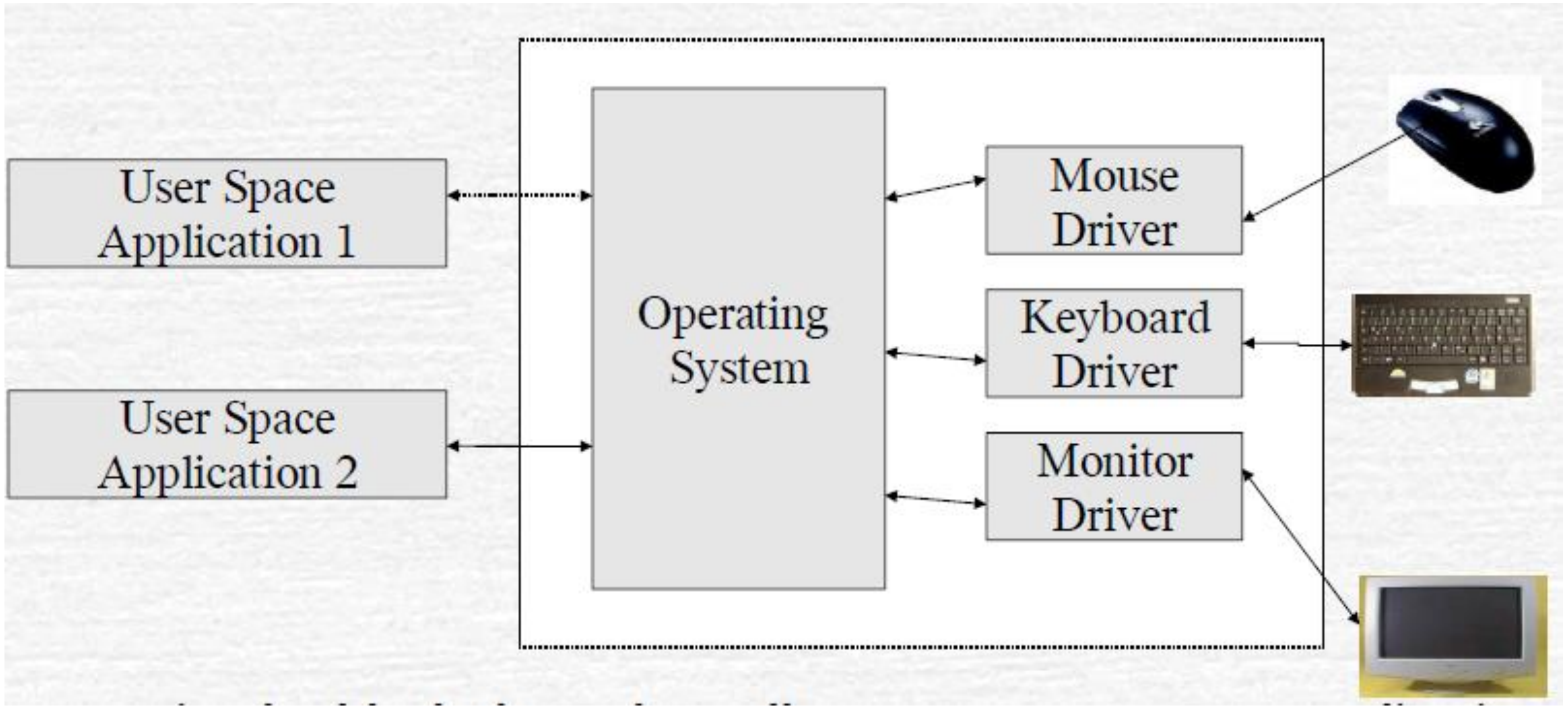
Linux Device Drivers
(Introduction to Device Drivers)

C-DAC Mumbai

AGENDA

- What are Device Drivers???
- Scope of this Module
- The Role of a Device Driver
- Splitting the Kernel
- Classes of Devices and Modules
- Version Numbering
- References

Introduction to Device Drivers

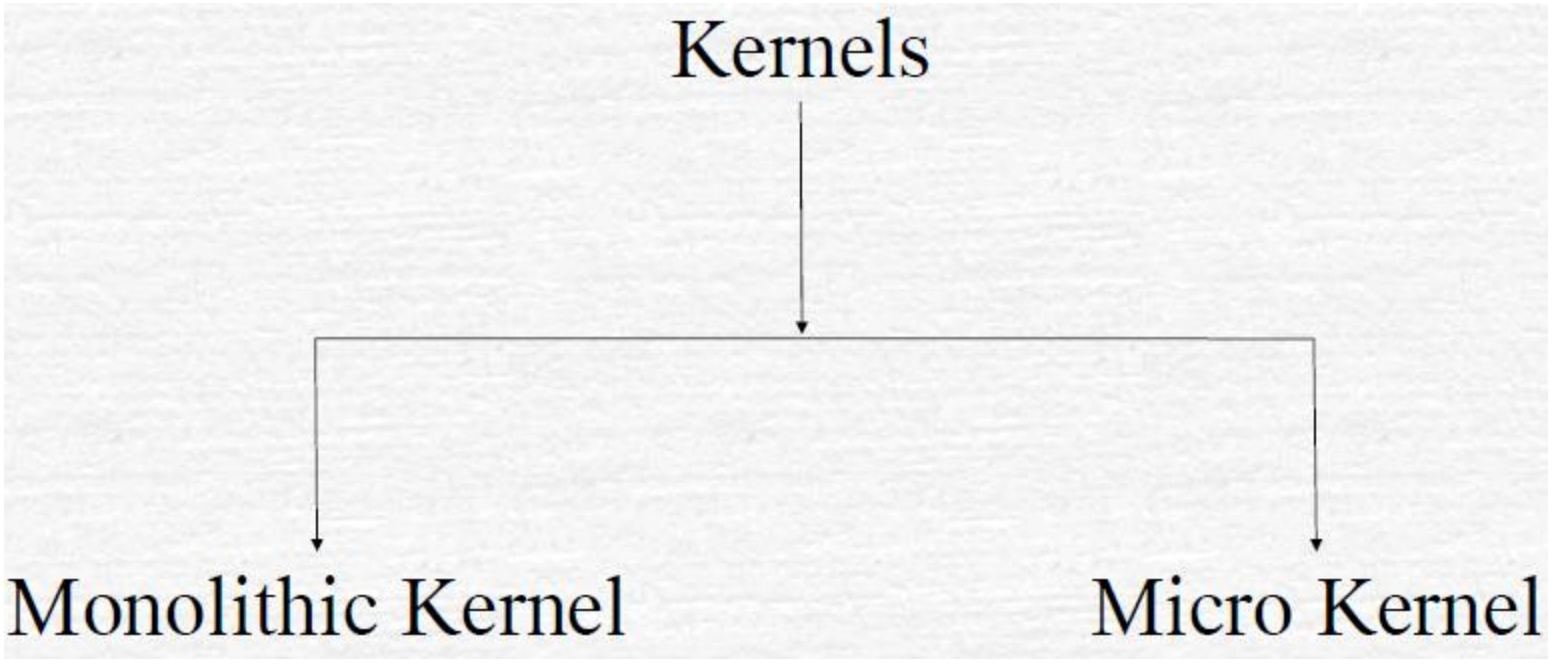


A simple black box that allows a user space application interaction with a hardware, without need for the knowledge of the hardware functioning

WHY THE CRAZE????

- The rate at which new hardware becomes available (and obsolete!) alone guarantees that driver writers will be busy for the foreseeable future
- To gain access to a particular device of interest
- Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential market

Kernel Classifications and Linux



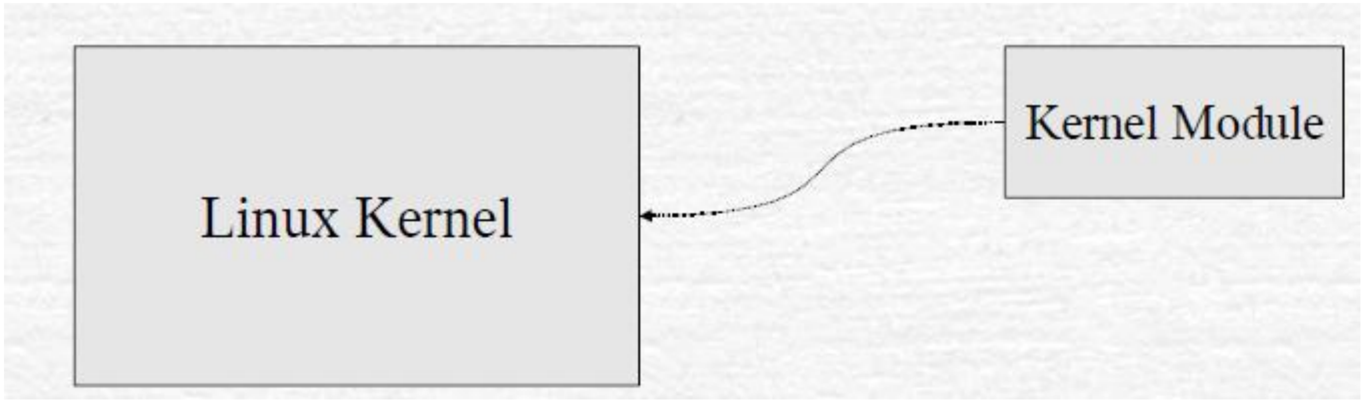
Monolithic Kernels

- **All the parts of a kernel like the device drivers, scheduler, file system, memory handling, networking stacks, etc., are maintained in one unit within the kernel**
- **Advantages**
  - Faster processing as message passing is not required
- **Disadvantages**
  - If any one module crashes, the entire system may go down
- **Examples**
  - Windows, Linux, etc…

Micro Kernels

- **Only the very important parts like IPC, basic scheduler, basic memory handling, basic I/O primitives etc., are put into the kernel.**
- **The other parts like the complete scheduler, memory handling, file systems and networking stacks are maintained as user space applications.**
- **Advantages**
  - The parts running in the user space may crash without causing major damage to the functioning of the entire system
- **Disadvantages**
  - Message passing mechanisms between the kernel and user space render slower functioning of the system
- **Examples:** QNX, Mach (CMU), Minix 3, EROS, KeyKOS

The Linux Kernel



- Linux is a monolithic kernel but allows a functionality of modules.
- Kernel modules may be added dynamically to the kernel whenever required and unloaded whenever it is not in use.
- This may be done either at kernel configuration time or later, once the system has booted

The Role of a Device Driver

- **The role of a driver is to provide mechanisms and not policies.**
- **Mechanisms????**
  - “what capabilities are to be provided”
- **Policies?????**
  - “how these capabilities can be used”

The Role of a Device Driver

- **Most software problems are split into the above discussed ideologies. If the two issues are addressed by separate parts of a program or by two separate programs altogether, the software package is easier to develop**
- **Examples of this modularization:**
  - Unix management of the graphic display being split into the X-server and the windows & sessions manager
- **Advantages:**
  - Users can use the same window manager on different hardware
  - Multiple window configurations can run on the same system

The Role of a Device Driver

- **Another example is the layered structure of TCP/IP networking:**
- **The OS offers a socket abstraction, which implements no policy regarding the data being transferred, while servers (like the ftp server) are in charge of their services (their associated policies)**

Characteristics of Policy free drivers

- Include support for synchronous and asynchronous operation
- The ability to be opened multiple times
- The ability to exploit full capabilities of the hardware
- Lack of software layers to “simplify things” or provide policy-related operations

User Space Applications and Kernel Space Modules

Comparison of User and Kernel Spaces

- Event Driven Execution: User space applications run to completion and once executed, they die out.
- Kernel modules on the other hand just register themselves with the kernel and wait until they are called to be executed. In short, kernel modules may also be called *event driven modules*
- Libraries: There are no libraries present in the kernel, unlike user space applications

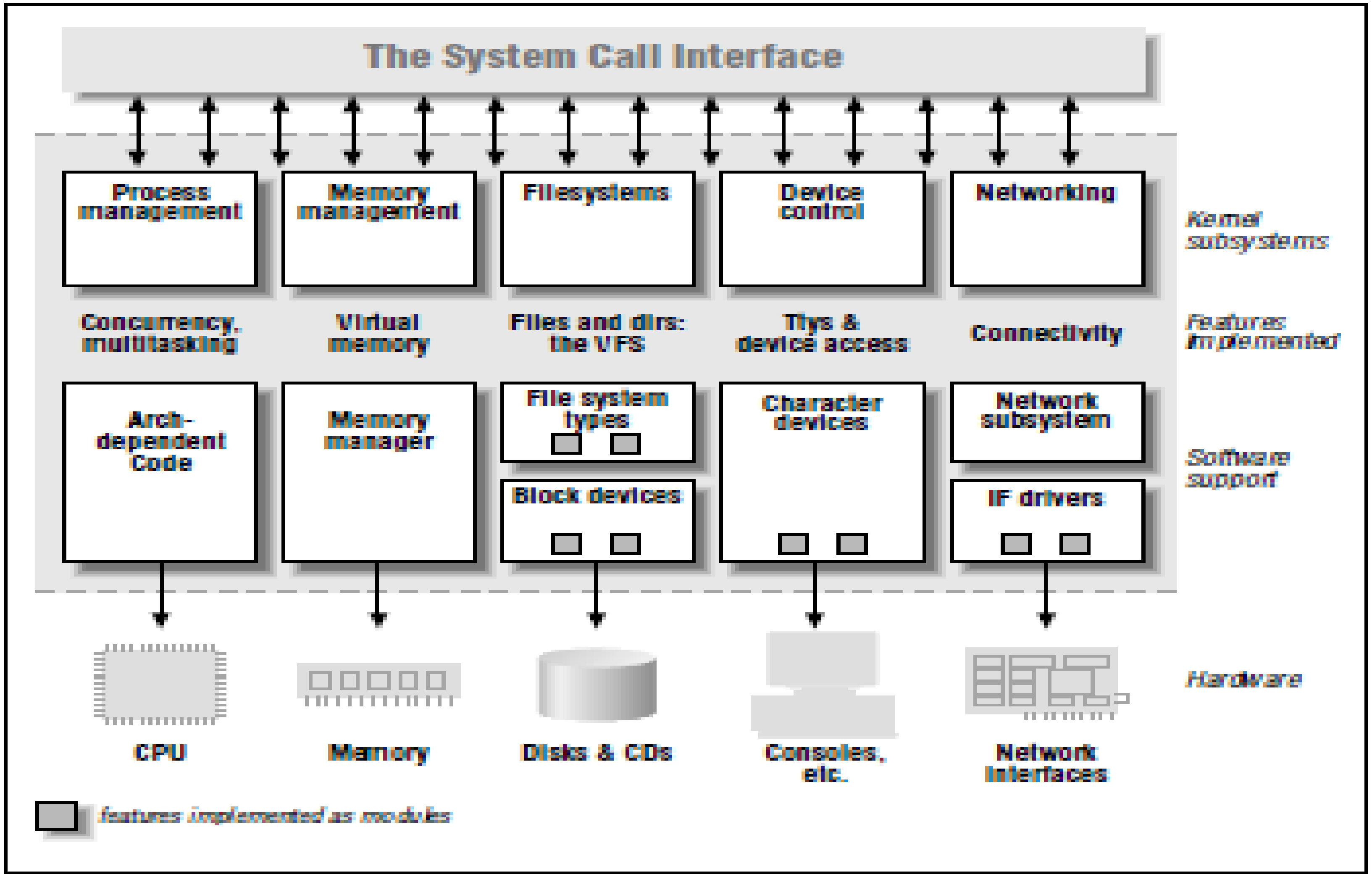
Comparison of User and Kernel Spaces

- Floating Point: There is no floating point support in the kernel.
- Segmentation Faults: Segmentation faults in the kernel may in the least kill the current process running or may even kill the whole system
- Kernel Debugging: No specialized mechanisms of debugging in the kernel

Comparison of User and Kernel Spaces

- **Kernel Stack: The stack in the kernel is limited to about 4k bytes. Therefore the stack is to be used very sparingly within the kernel**
- **Infinite Looping: Infinite looping should not be done in the kernel**

Splitting the Linux Kernel



Splitting the Kernel

- **Process Management**
  - creating and destroying processes, handling connection to the outside world, communication among different processes.... scheduler controls how processes share the CPU
- **Memory Management**
  - memory usage and virtual address space
- **Filesystems**
  - managing files, supporting multiple file systems

Splitting the Kernel

- **Device Control**
  - device drivers
- **Networking**
  - packet reception, handling, transmission, routing, address resolution

Classes of Devices and Modules

- **The Linux way of looking at devices distinguishes between three fundamental device types**
- **All modules usually implement one of these types and thus are classifiable as**
  - character module
  - block module
  - network module

Character Devices

- Device that can be accessed as a stream of bytes. A char driver is in charge of implementing this behaviour Implements the basic functionalities such as *open, read, write, close system calls*
  - Examples: text console, serial ports..etc..,

- Char devices are accessed by user space applications through device nodes implemented by the file system such as */dev/ttyS0, /dev/tty1*

- The relevant difference between a char device and a usual file of the system is the backward seeking not being possible in a char device.

Block Devices

- A block device is one that can host a filesystem. A block device can handle I/O operations that transfer one or more whole blocks
- Like char devices, block devices are accessed through File system nodes in the */dev directory*
- 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
- To a user, both the char device and block device look similar but differ in interface to the kernel

Network Interfaces

- Any network interactions are made through an interface, that is, a device that can exchange data with other hosts.
- A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted.
- A network driver does not know anything about connections. It only handles packets.

Network Interfaces

- A network interface isn't easily mapped to a node in the filesystem. A unique name is assigned to them (such as eth0) that does not have an entry in the filesystem.
- Communications between the kernel and a network device is completely different as compared to a char or block device.
- Functions related to packet transmission are issued in place of read and write in network drivers

Other Driver Classifications

- **There are other classifications to driver modules that are orthogonal to the previously discussed. In general, some types of drivers work with additional layers of kernel support functions for a given type of device**
  - **Example:**
  - Every USB device is driven by a USB module that works with the USB subsystem, but the device itself shows up in the system as a char device (USB serial port), a block device (a USB memory card reader) or, a network device (a USB Ethernet Interface)

Software Drivers

- In addition to device drivers, there are other modules in the kernel that do not relate to any specific hardware. They simply map the organization of a device to higher level structures.
- A common example is a filesystem implementation which maps lower level structures to higher level structures of the filesystem. Such a module is called a Software Driver.

Version Numbering

- Every software package used in a Linux system has its own release number, and there are often interdependencies across them
- Even numbered kernel numbers are stable versions and are intended for general distributions. eg., linux 2.6, 2.4
- Odd numbered kernel versions are development snapshots and represent the current status of development

Module Utilities

- When a new module is loaded, related information is available in the kernel log.
- The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)
- Kernel log messages are available through the dmesg command (diagnostic message)
- Kernel log messages are also displayed in the system console (console messages can be filtered by level using the loglevel kernel parameter, or completely disabled with the quiet parameter).
- Note that you can write to the kernel log from userspace too:

```
echo "Debug info" > /dev/kmsg
```

Module Utilities

```
modinfo <module_name>
modinfo <module_path>.ko
```

- Gets information about a module: parameters, license, description and dependencies.

- Very useful before deciding to load a module or not.

```
sudo insmod <module_path>.ko
```

- Tries to load the given module. The full path to the module object file must be given.

UNDERSTANDING MODULE LOADING ISSUES

- When loading a module fails, insmod often doesn't give you enough details!
- Details are often available in the kernel log.

- Example:

```
$ sudo insmod ./intr_monitor.ko
```

- insmod: error inserting './intr_monitor.ko': -1 Device or resource busy

```
$ dmesg
```

- [17549774.552000] Failed to register handler for irq channel 2

Module Utilities

```
sudo modprobe <module_name>
```

- Most common usage of modprobe: tries to load all the modules the given module depends on, and then this module.
- Lots of other options are available. modprobe automatically looks in /lib/modules/<version>/ for the object file corresponding to the given module name.

```
$lsmod
```

- Displays the list of loaded modules
- Compare its output with the contents of /proc/modules!

Module Utilities

```
sudo rmmod <module_name>
```

- Tries to remove the given module.
- Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)

```
sudo modprobe -r <module_name>
```

- Tries to remove the given module and all dependent modules (which are no longer needed after removing the module)

Hello World (First Driver program)

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

static int __init hello_init(void)
{
		printk(KERN_INFO"Good morning");
		printk(KERN_INFO"How are you  World.\n");
		return 0;
}

static void __exit hello_exit(void)
{
		printk(KERN_INFO"Bye Bye.......");
		printk(KERN_INFO"Have a nice day..\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("MY_FIRST_MODULE");
MODULE_AUTHOR("CDAC_MUMBAI");
```

Hello World (First Driver program)

- Headers specific to the Linux kernel: linux/xxx.h
- No access to the usual C library, we're doing kernel programming
- An initialization function

  Called when the module is loaded, returns an error code (0 on success, negative value on failure)
- Declared by the module_init() macro:

  The name of the function doesn't matter, even though <modulename>_init() is a convention.
- A cleanup function

  Called when the module is unloaded
- Declared by the module_exit() macro.
- Metadata information declared using MODULE_LICENSE(), MODULE_DESCRIPTION() and MODULE_AUTHOR()

Thank You….