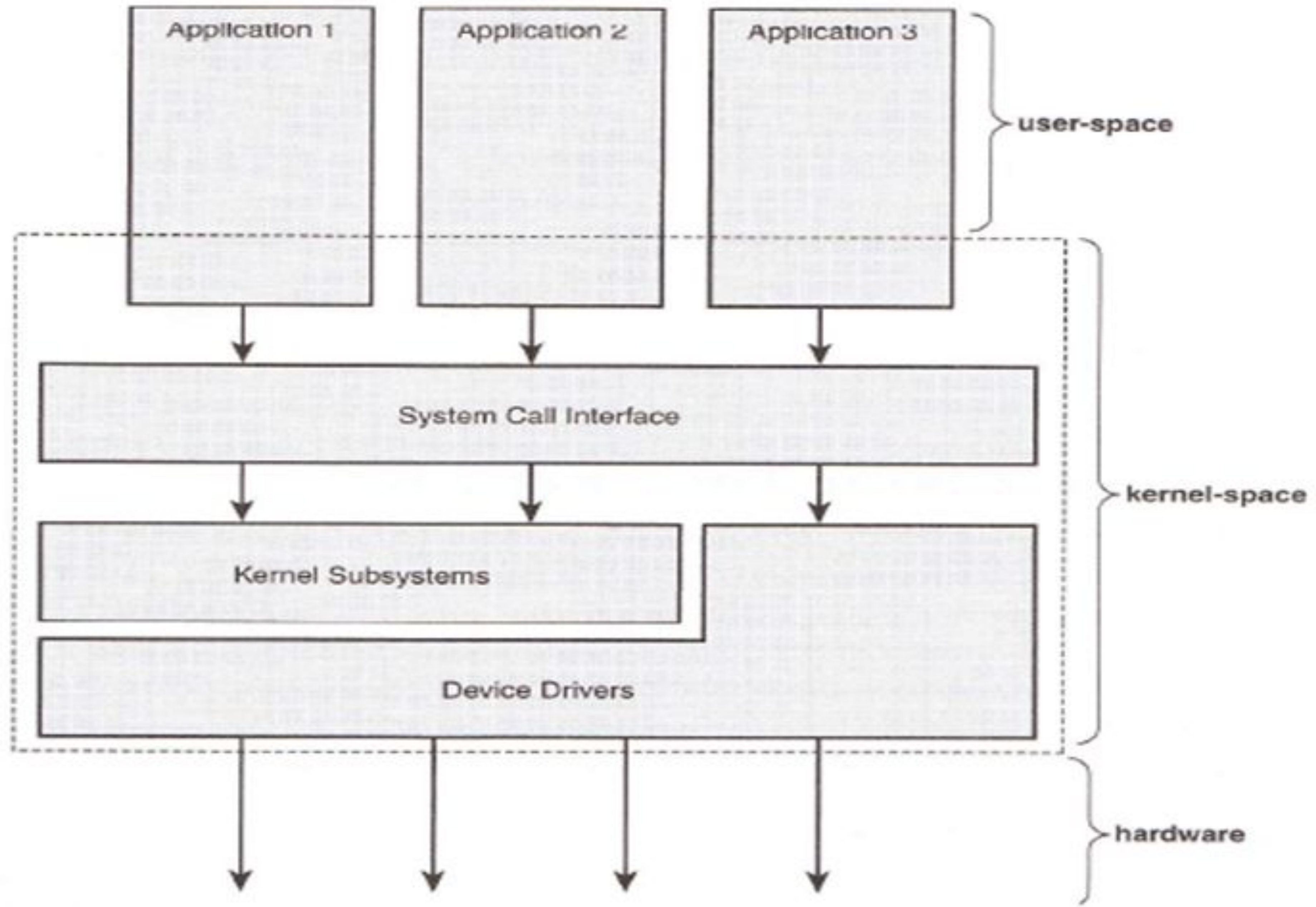
Device Driver

User mode and kernel mode

- A processor in a computer has two different modes: *user mode* and *kernel mode*.
- The processor switches between the two modes depending on what type of code is running on the processor.
- Applications run in user mode, and core operating system components run in kernel mode. Many drivers run in kernel mode, but some drivers run in user mode.
- When you start a user-mode application, Windows creates a *process* for the application.
- The process provides the application with a private *virtual address space* and a private *handle table*. Because an application's virtual address space is private, one application cannot alter data that belongs to another application.
- Each application runs in isolation, and if an application crashes, the crash is limited to that one application. Other applications and the operating system are not affected by the crash.

Kernel Basics



- Types of Drivers, Driver History, Driver Issues, Kernel Level Device drivers, Virtual device drivers(VxD),Writing a Driver, Device Driver Stack Buses and Physical Devices,
- Static Device drivers, Dynamic Device drivers, PnP,
- Device Namespace, and Named Devices.

Device Driver

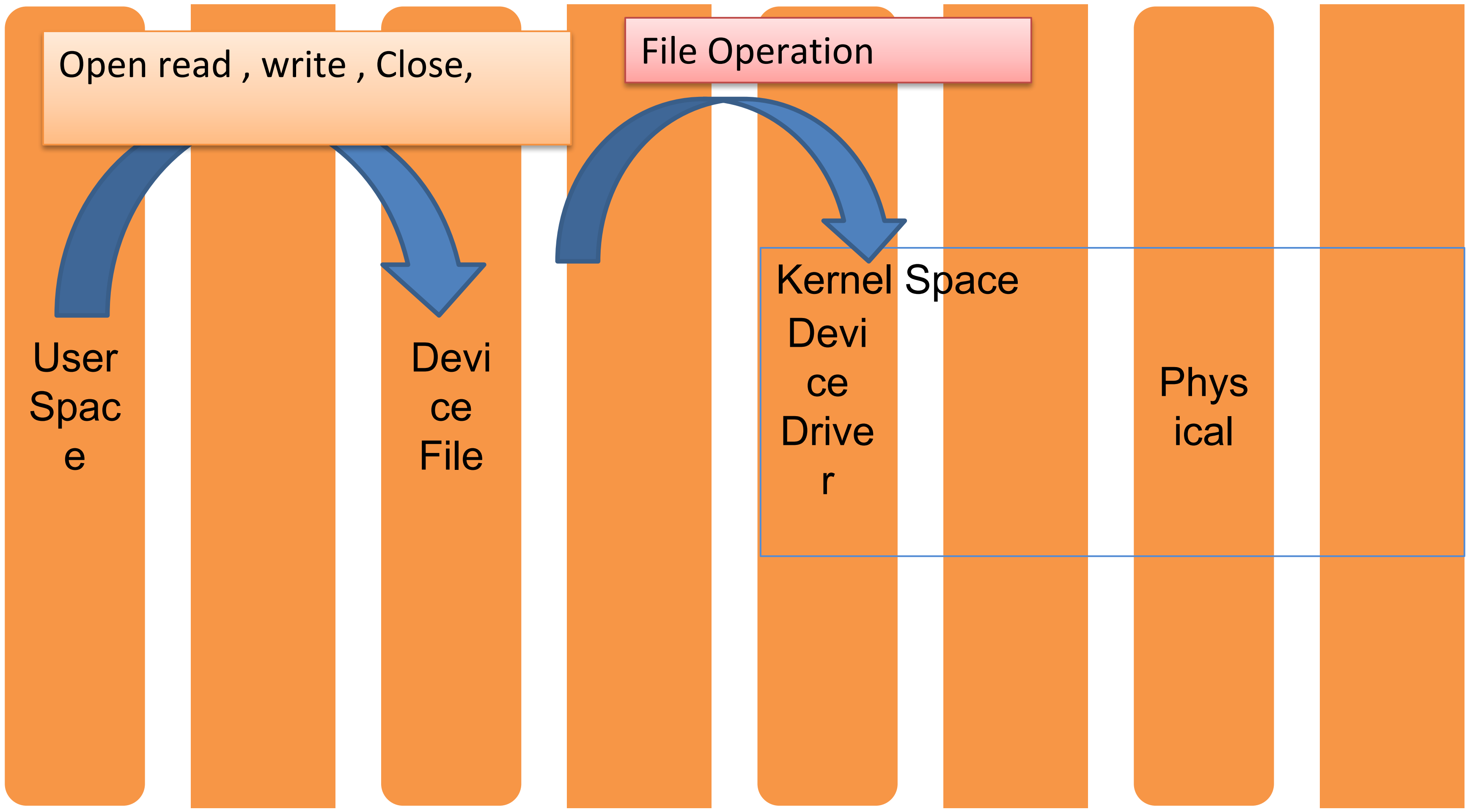
- **Device driver** or **software driver** is a computer program allowing higher-level computer programs to interact with a hardware device.
- Device driver simplifies programming by acting as translator between a hardware device and the applications or operating systems that use it
- Drivers are hardware-dependent and operating-system-specific
- **Hide implementation and hardware-specific details**

  from a user program

DEVICE DRIVER

- **A device driver is a program that controls a particular type of device that is attached to your computer.**
- **Black boxes to hide details of hardware devices.**
- Use standardized calls
- Main role is to Map standard calls to device-specific operations
- Can be developed separately from the rest of the kernel
- Plugged in at runtime when needed

Device Driver



- Concurrency in the Kernel
  - Reentrant code
  - Data structure – to keep multiple thread execution separate
  - Code access to shared data prevent corruption
  - Concurrency but avoid race condition
  - Nonpreemtive behaviour prevent unwanted concurrency

Device Driver Types

- A character device driver (c)
    - Reading/Writing character by character
    - Most devices are this type (e.g.Modem, lp, USB
    - No buffer.
- A block device driver (b)
    - Reading/Writing block by block
    - Through a system buffer that acts as a data cache.
    - Hard drive controller and HDs
    - Random access devices (buffering) For example a file system can only be mounted on a block device, because it requires random access.
    - For example, disks are commonly implemented as block devices.

Kernel basics

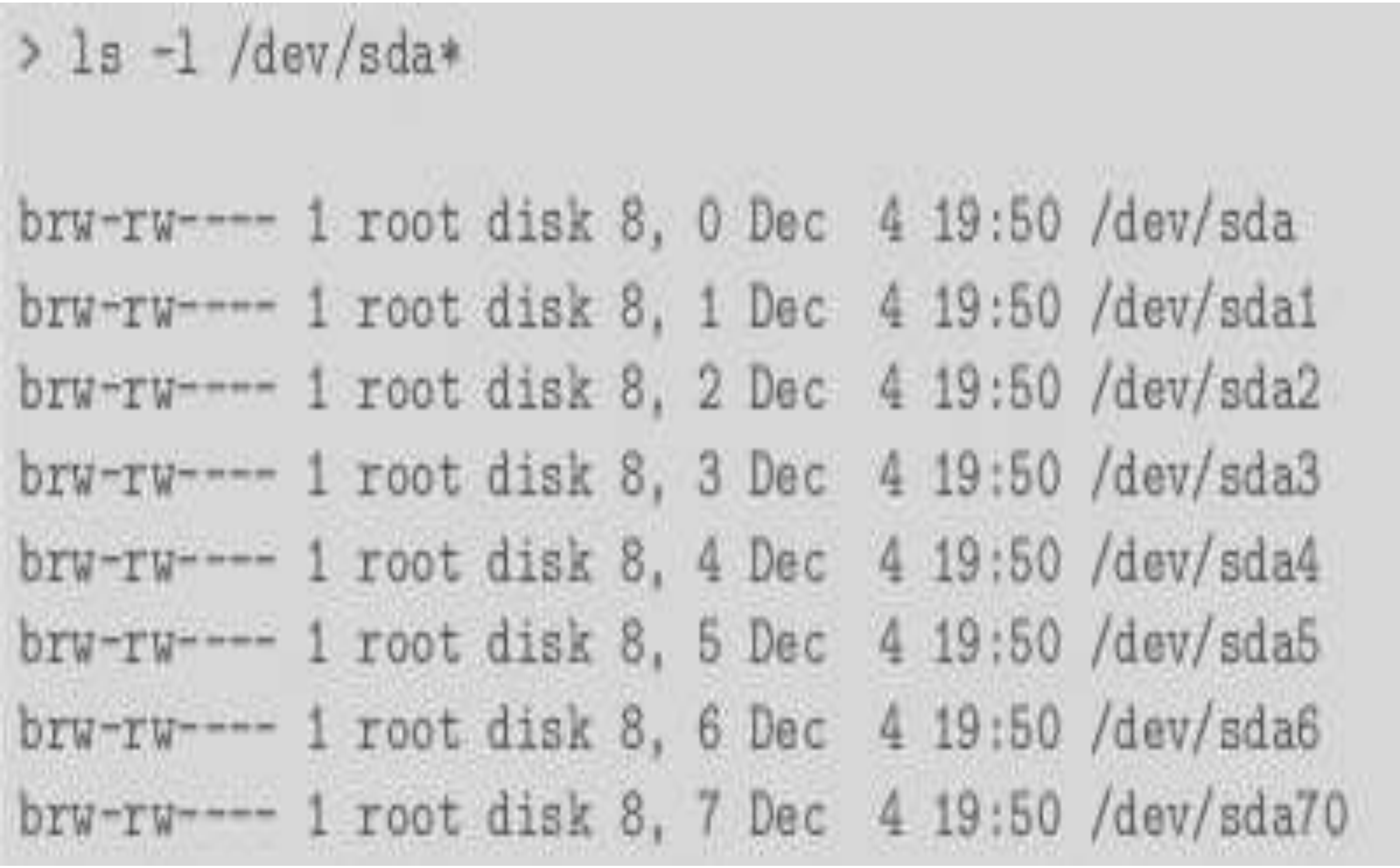
- No libc functions!!!
  - No printf, use printk instead
  - Some common functions implemented inside the kernel
- Small, fixed size stack
- Synchronization and Concurrency issues

Major and Minor Numbers

- Linux identifies each device by a major device number and a minor device number. The *major device number identifies the driver* associated with the device.

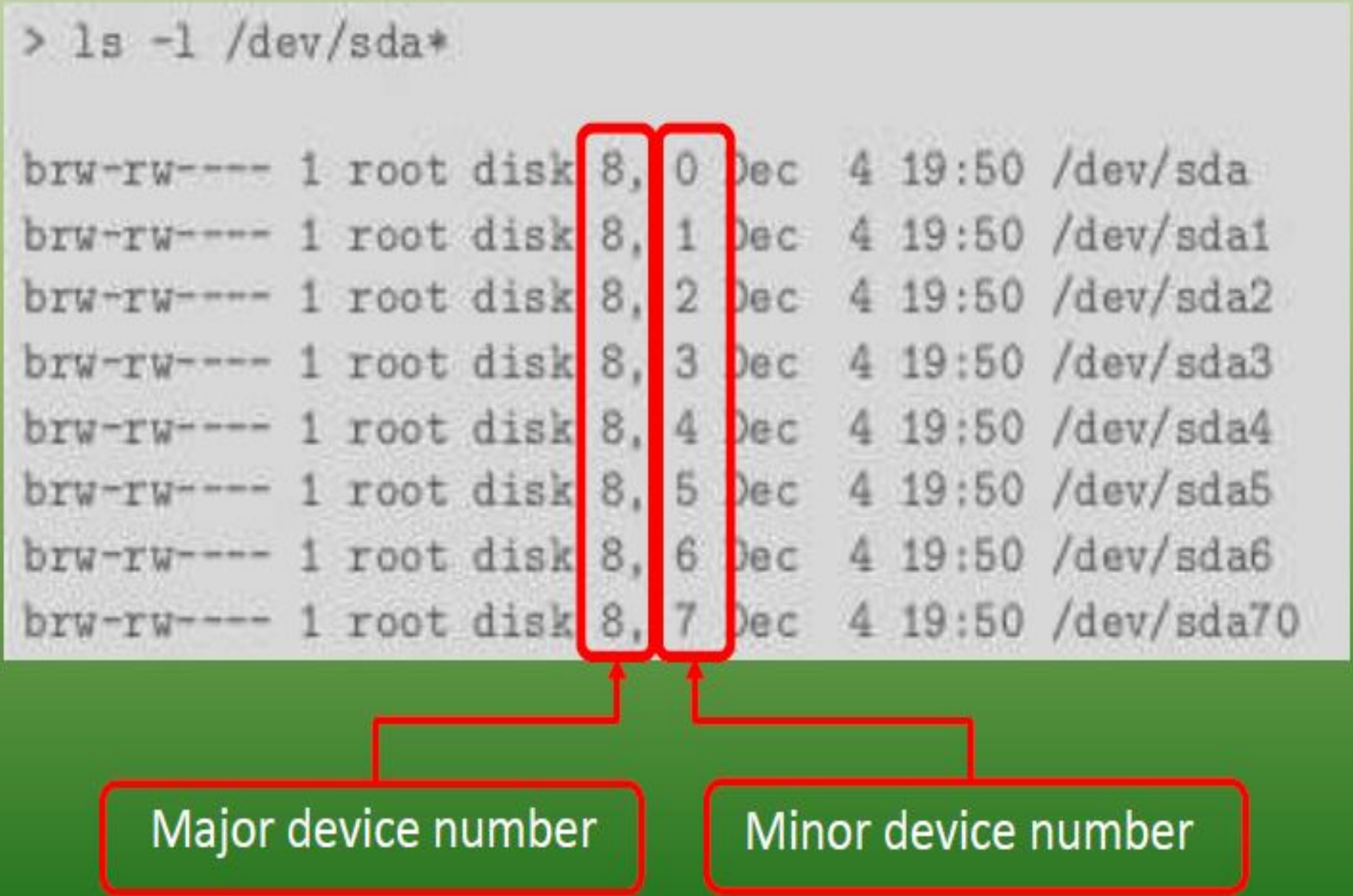
- **Major number**
  - Each device driver is identified by a unique major number.
  - This number is assigned by the Linux Device Registrar.
  - This number is the index into an array that contains the information about the driver (the device_struct)
  - The *major device number identifies the driver* associated with the device.
- **Minor number**
  - This uniquely identifies a particular instance of a device. For example, a system may have multiple IDE hard disks each would have a major number of 3, but a different *minor number*.
  - the minor device number is used by the kernel to determine exactly which device is being referred to.

Major and Minor Numbers



```
> ls -l /dev/sda*


brw-rw---- 1 root disk 8, 0 Dec  4 19:50 /dev/sda
brw-rw---- 1 root disk 8, 1 Dec  4 19:50 /dev/sda1
brw-rw---- 1 root disk 8, 2 Dec  4 19:50 /dev/sda2
brw-rw---- 1 root disk 8, 3 Dec  4 19:50 /dev/sda3
brw-rw---- 1 root disk 8, 4 Dec  4 19:50 /dev/sda4
brw-rw---- 1 root disk 8, 5 Dec  4 19:50 /dev/sda5
brw-rw---- 1 root disk 8, 6 Dec  4 19:50 /dev/sda6
brw-rw---- 1 root disk 8, 7 Dec  4 19:50 /dev/sda70
```

Major and Minor Numbers



```
> ls -l /dev/sda*

brw-rw---- 1 root disk 8, 0 Dec  4 19:50 /dev/sda
brw-rw---- 1 root disk 8, 1 Dec  4 19:50 /dev/sda1
brw-rw---- 1 root disk 8, 2 Dec  4 19:50 /dev/sda2
brw-rw---- 1 root disk 8, 3 Dec  4 19:50 /dev/sda3
brw-rw---- 1 root disk 8, 4 Dec  4 19:50 /dev/sda4
brw-rw---- 1 root disk 8, 5 Dec  4 19:50 /dev/sda5
brw-rw---- 1 root disk 8, 6 Dec  4 19:50 /dev/sda6
brw-rw---- 1 root disk 8, 7 Dec  4 19:50 /dev/sda70
```

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()

Driver program

```
#include <linux/module.h>     /* Needed by all modules */
#include <linux/kernel.h>      /* Needed for KERN_INFO */

int init_module(void)
{
        printk(KERN_INFO "Hello world 1.\n");

/*  A non 0 return means init_module failed; module can't be loaded.          */
        return 0;
}
void cleanup_module(void)
{
        printk(KERN_INFO "Goodbye world 1.\n");
}
```

- **printk()**

  - not meant to communicate information to the user.

  - logging mechanism for the kernel, and is used to log information or give warnings.

  - printk() statement comes with a priority

  -If the priority is less than (int)console_loglevel, the message is printed on your current terminal. If both **syslogd** and klogd are running, then the message will also get appended to /var/log/messages

Makefile

- Makefile is a script for appropriate compilation of different type of sources to the appropriate object code.
- To use a makefile just type make or gmake in a console window opened in the same dir as the makefile

Makefile

```
obj-m += hello-1.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
```

Commands to run module

$ dmesg (*display message* or *driver message*) is a command on most Unix-like operating systems that prints the message buffer of the kernel.)

$ make

$ dmesg

$ sudo insmod hello.ko (insmod command is used to insert modules to Linux kernel.)

$ lsmod | hello (to verify that the module has been inserted successfully)

$ rmmod hello

$ dmesg

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.

Virtual device drivers

- used to emulate a hardware device, particularly in virtualization environments
- Instead of enabling the guest operating system to dialog with hardware, virtual device drivers take the opposite role and emulate a piece of hardware, so that the guest operating system and its drivers running inside a virtual machine can have the illusion of accessing real hardware

VxD(virtual xxx driver)

- A virtual device driver (VxD) is a software device driver that emulates hardware and other devices so that multiple applications running in protected mode can access hardware interrupt channels, hardware resources and memory without causing conflicts.
- To share arbitrary physical resources amongst these virtual machines, Microsoft introduced dynamically-loadable virtual device drivers.

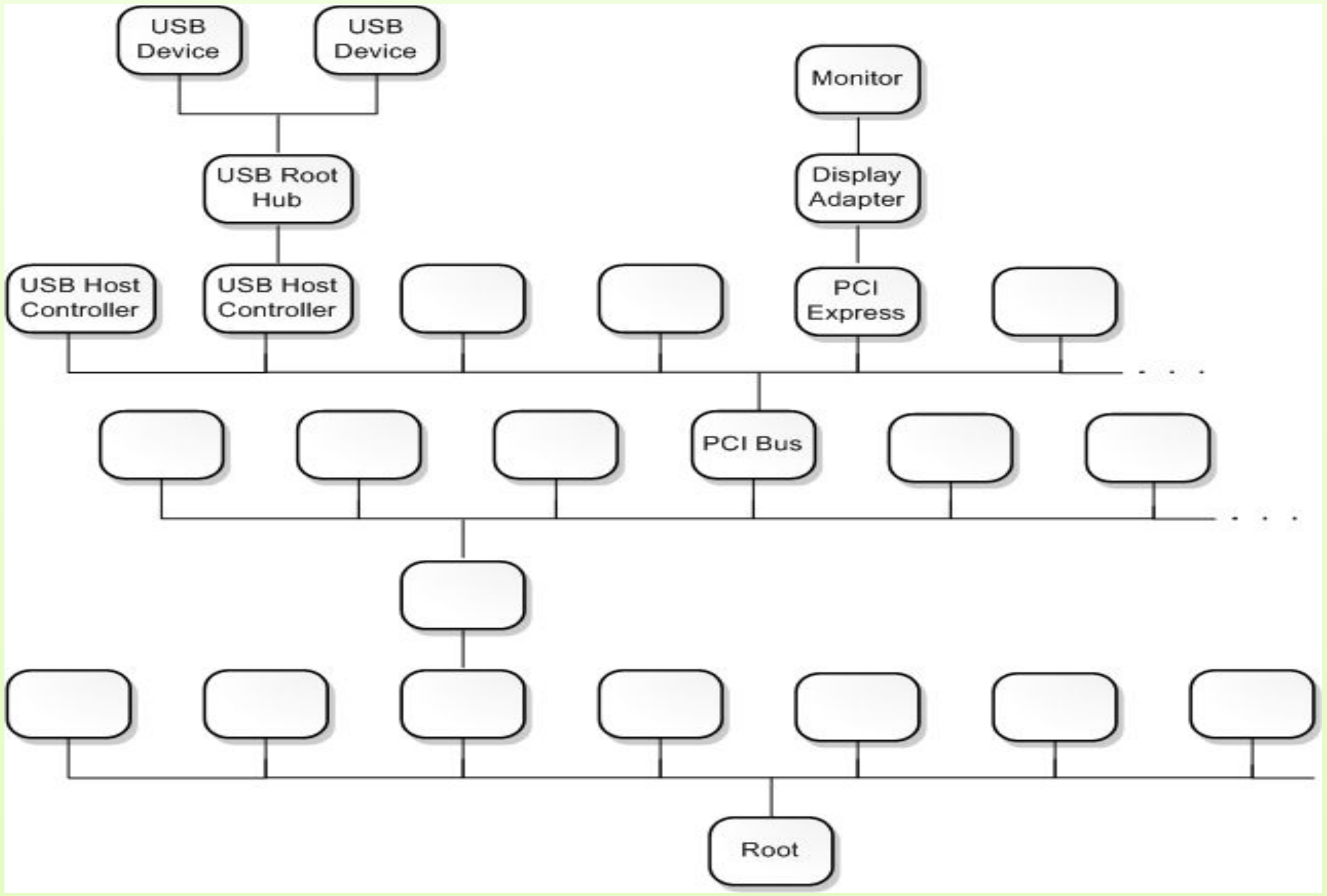
- In Windows, a device driver is controlled by the operating system's virtual device driver manager (VDDM) and is shared by the applications running within that kernel
- There was no hardware device sharing allowed in most standard DOS applications, so the virtual device driver (VxD) was introduced to prevent device access conflicts. The VxD passed interrupt and memory requests through to the kernel, which in turn allocated the resources as required, always ensuring only a single request thread could access a single interrupt channel of any device at any one time

User mode and kernel mode

- A processor in a computer running Windows has two different modes: *user mode* and *kernel mode*.
- The processor switches between the two modes depending on what type of code is running on the processor.
- Applications run in user mode, and core operating system components run in kernel mode. Many drivers run in kernel mode, but some drivers run in user mode.
- When you start a user-mode application, Windows creates a *process* for the application.
- The process provides the application with a private *virtual address space* and a private *handle table*. Because an application's virtual address space is private, one application cannot alter data that belongs to another application.
- Each application runs in isolation, and if an application crashes, the crash is limited to that one application. Other applications and the operating system are not affected by the crash.

Device nodes and device stacks

- Windows organizes devices in a tree structure called the *device tree*
- node in the device tree is called a *device node*



Device objects and device stacks

- A *device object* is an instance of a **DEVICE_OBJECT** structure.
- Each device node has an ordered list of device objects, and each of these device objects is associated with a driver.
- The ordered list of device objects, along with their associated drivers, is called the *device stack* for the device node.
- first device object to be created in the device stack is at the bottom, and the last device object to be created and attached to the device stack is at the top

Driver stacks

- In the Windows operating system, WDM drivers are layered in a vertical calling sequence that is called a *driver stack*.
- The topmost driver in the stack typically receives I/O requests from user applications, after the requests have passed through the operating system's I/O manager. The lower driver layers typically communicate with computer hardware.

Device stacks

- Each driver stack supports one or more *device stacks*. A device stack is a set of *device objects* that are created from WDM-defined DEVICE_OBJECT structures. Each device stack represents one device. Each driver creates a device object for each of its devices and attaches each device object to a device stack. Device stacks are created and removed as devices are plugged in and unplugged, and each time the system is rebooted.
- When a bus driver detects that child devices have been plugged in or unplugged, it informs the Plug and Play (PnP) manager. In response, the PnP manager asks the bus driver to create a physical device object (PDO) for each child device that is connected to the parent device (that is, the bus). The PDO becomes the bottom of a device stack.

Device stacks

- **Next, the PnP manager loads function and filter drivers to support each device (if they are not already loaded), and then the PnP manager calls these drivers so that each can create a device object and add it to the top of the device stack. Function drivers create functional device objects (FDOs), and filter drivers create filter device objects (filter DOs).**
- **When the I/O manager sends an I/O request to a device's drivers, it passes the request to the driver that created the topmost device object in the device stack. If that driver asks the I/O manager to pass the request to the next-lower driver, the I/O manager uses the device stack to determine the next-lower driver. (The next-lower driver is the driver that created the next-lower device**

Function drivers, filter drivers, and bus drivers

- Each driver in a device stack plays the role of either function driver, filter driver, or bus driver.
- main driver for the device stack is called the *function driver for the device stack* or the *function driver for the device node*
- function driver is responsible for handling read requests, write requests, and device control requests
- device object that is associated with a function driver is called a *functional device object* (FDO).

- driver at the bottom of the device stack is called the *bus driver for the device stack* or the *bus driver for the device node*.
- A device object that is associated with the bus driver for a device stack is called a *physical device object* (PDO).
- A driver that is above the function driver or in between the bus and function drivers is called a *filter driver*.
- A device object that is associated with a filter driver is called a *filter device object* (Filter DO).

.inf, PnP

- When the drivers for a device are installed, the installer uses information in an information (INF) file to determine which driver is the function driver and which drivers are filters.
- Typically the INF file is provided either by Microsoft or by the hardware vendor.
- After the drivers for a device are installed, the Plug and Play (PnP) manager can determine the function and filter drivers for the device by looking in the registry.

Buses and bus drivers

1. A *bus* is any parent node in the device tree. In other words, any device node that has one or more child nodes is a bus.

The *bus driver* for a device node is the driver associated with the PDO at the bottom of the node's device stack.

2. A *bus* is a physical device that can have other physical devices connected to it. For example, the PCI bus is a set of conductors and circuitry on the motherboard, and devices are connected to the PCI bus by being built into the motherboard or by being plugged into expansion slots. Physical USB devices can be connected to a USB host controller.

A *bus driver* is the function driver associated with a physical bus

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()

Driver program

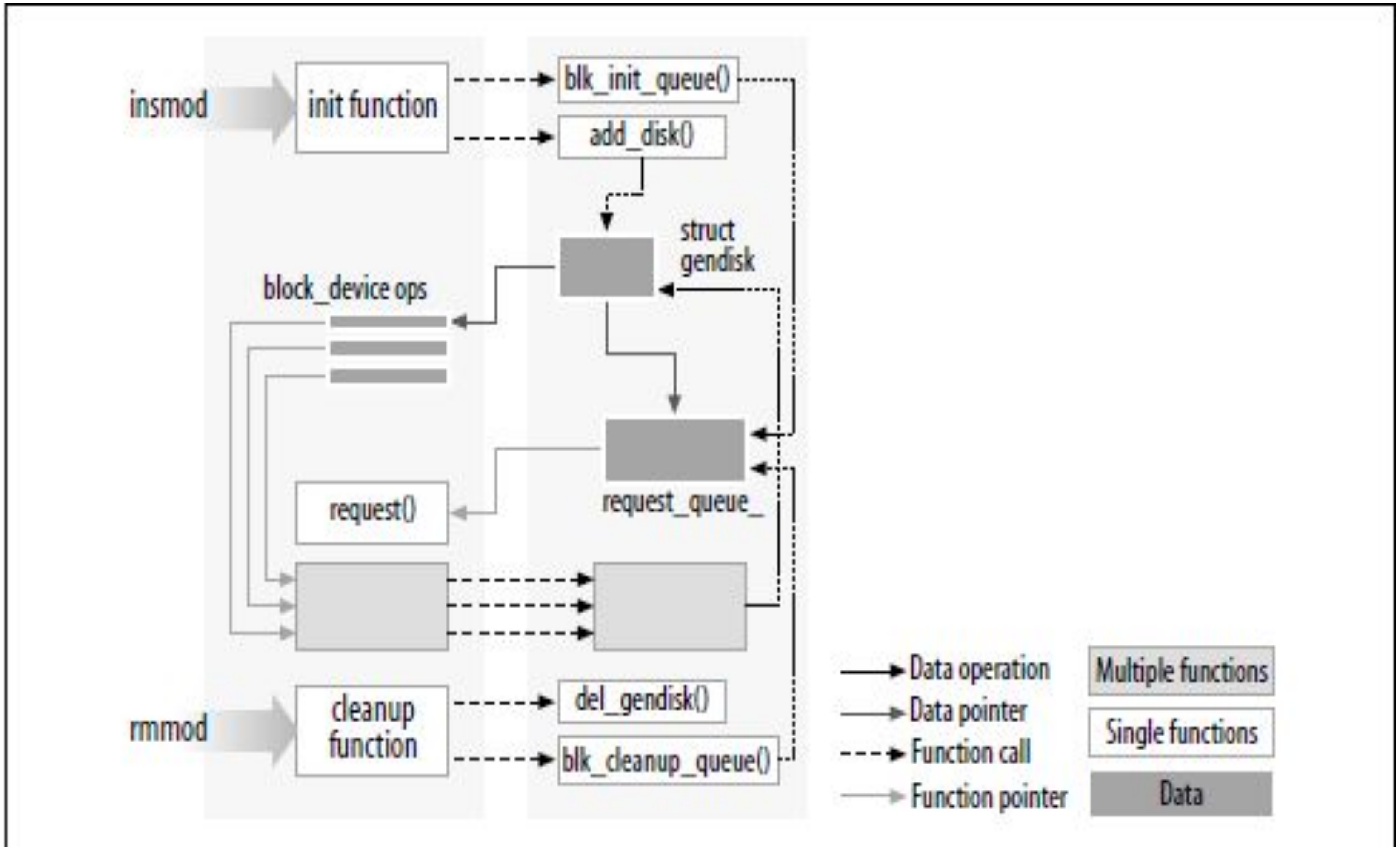
```
#include <linux/module.h>     /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */

int init_module(void)
{
        printk(KERN_INFO "Hello world \n");

/*  A non 0 return means init_module failed; module can't be loaded.         */
        return 0;
}
void cleanup_module(void)
{
        printk(KERN_INFO "Goodbye world \n");
}
```

- $ gcc –c hello.c
- $ insmod ./hello.o

Hello world

- $ rmmod hello

Goodbye world



insmod
init function
blk_init_queue()
add_disk()
struct gendisk
block_device ops
request_queue_
request()
rmmod
cleanup function
del_gendisk()
blk_cleanup_queue()
Data operation
Data pointer
Function call
Function pointer
Multiple functions
Single functions
Data

- **printk()**

  - not meant to communicate information to the user.

  - logging mechanism for the kernel, and is used to log information or give warnings.

  - printk() statement comes with a priority

  -If the priority is less than (int)console_loglevel, the message is printed on your current terminal. If both **syslogd** and klogd are running, then the message will also get appended to /var/log/messages

Kernel Module

- Modules are pieces of code that can be loaded and unloaded into the kernel upon demand.
- They extend the functionality of the kernel without the need to reboot the system.
- (For Ex- device driver, which allows the kernel to access hardware connected to the system.)
- Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality
- kernel headers are more than sufficient to compile kernel modules / drivers

- Lsmod- Command to see which modules are already loaded into the kernel
- It reads information by reading the file /proc/modules.
- When the kernel needs a feature that is not resident in the kernel, the kernel module daemon *kmod* execs *modprobe* to load the module in.

- insmod-modprobe uses insmod to first load any prerequisite modules into the kernel, and then the requested module.
- modprobe directs insmod to/lib/modules/version/, the standard directory for modules
- modprobe is aware of the default location of modules, knows how to figure out the dependancies and load the modules in the right order.

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.

Kernel Functions

- copy_to_user()
- copy_from_user()
- vmalloc()
- alloc_pages(), alloc_page()
- free_pages(), free_page()
- kmalloc(), kfree()
- vmalloc(), vfree()

Makefile

- Makefiles are a simple way to organize code compilation
- Makefile is a script for appropriate compilation of different type of sources to the appropriate object code.
- To use a makefile just type make or gmake in a console window opened in the same dir as the makefile

hellomake.c

```c
#include <hellomake.h>
int main() {
  // call a function in
another file
  myPrintHelloMake();
  return(0);
}
```

hellofunc.c

```c
#include <stdio.h>
#include <hellomake.h>
void
myPrintHelloMake(void) {
  printf("Hello
makefiles!\n");
  return;
}
```

hellomake.h

```c
/*
example include file
*/
void
myPrintHelloMake(void);
```

```
gcc -o hellomake hellomake.c hellofunc.c -I
```

- Makefile1

```
hellomake: hellomake.c hellofunc.c
        gcc -o hellomake hellomake.c hellofunc.c -I.
```

- Makefile2

```
CC=gcc
CFLAGS=-I.
 hellomake: hellomake.o hellofunc.o
$(CC) -o hellomake hellomake.o hellofunc.o -I.
```

- create a C test file "test.c" and then create makefile as shown below
- **Makefile**

```
all:
	cc -o test test.c
clean:
	rm -f test
```

- Save this file as "Makefile"
- Run this makefile by make command. It then creates test executable file at present directory
- Run the file using ./test
- You can use make clean to delete the executable file(test).

Makefile

```
obj-m += hello-1.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
```

From a technical point of view just the first line is really necessary, the "all" and "clean" targets were added for pure convenience.

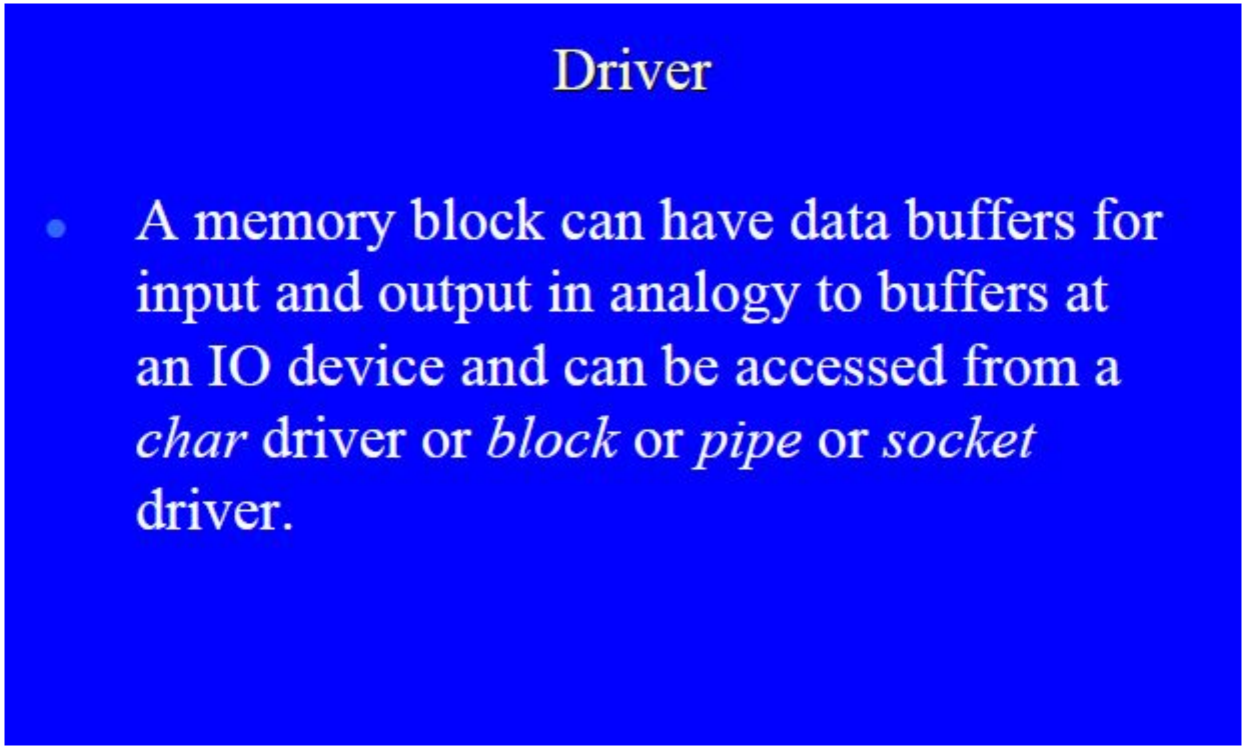
- (obj-m) specifies object files which are built as loadable kernel modules.
  - CUR = $(shell uname -r)
  - DIR = /lib/modules/$(CUR)/build
  - PWD = $(shell pwd)
  - obj-m := m1.o m2.o
  - default: $(MAKE) -C $(DIR) SUBDIRS=$(PWD) modules

if we delete the obj-m variable initialization then this make doesn't compile m1 and m2 kernel modules

- $(obj-m) specifies object files which are built as loadable kernel modules.
- A module may be built from one source file or several source files.
- In the case of one source file, the kbuild or makefile simply adds the file to $(obj-m).

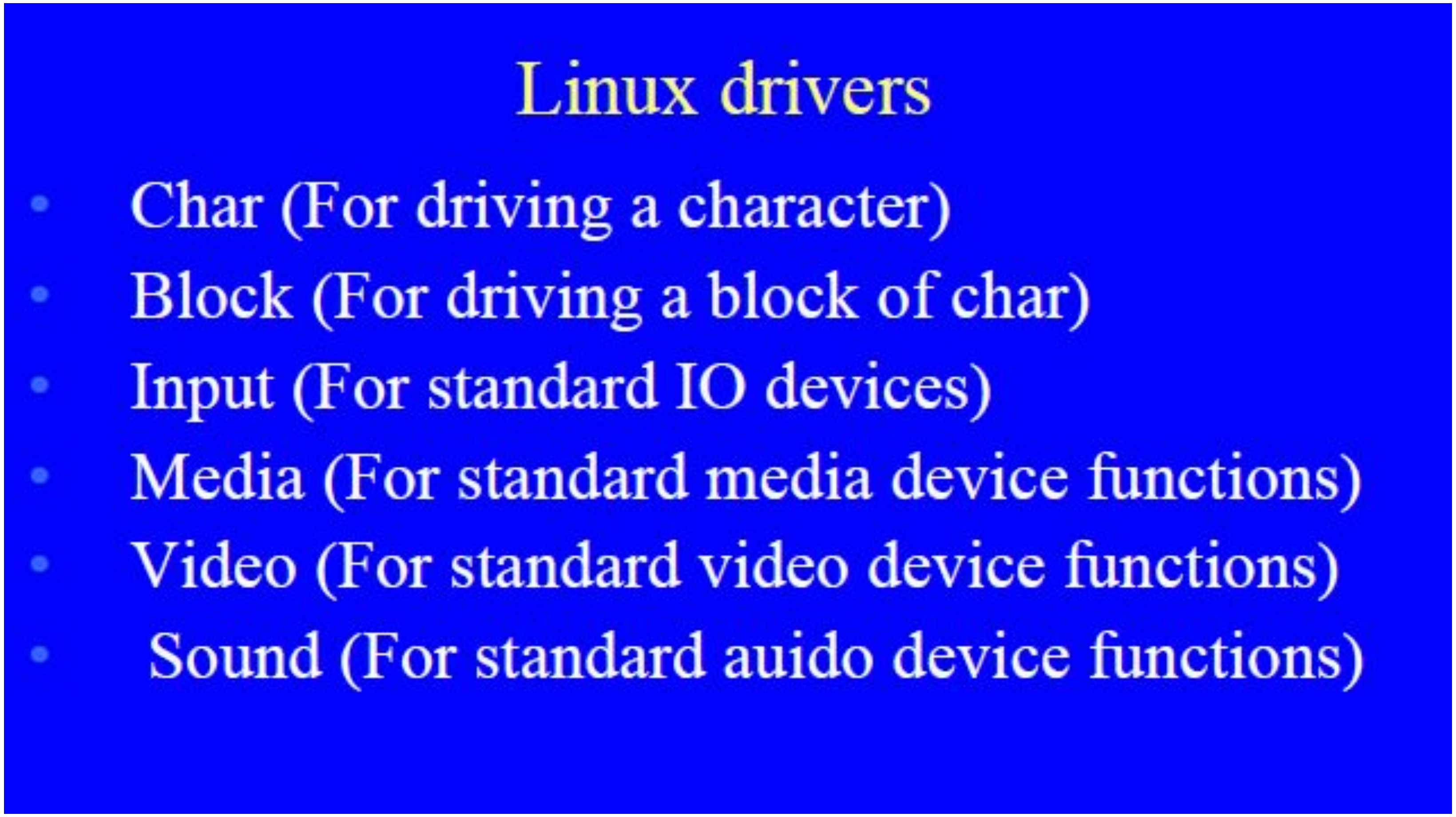
Commands to run module

```
$ make
$ insmod hello.ko
$ lsmod | less
$ tail –f /var/log/messages
$modinfo hello.ko
$ rmmod hello
$ tail –f /var/log/messages
```



Driver

- A memory block can have data buffers for input and output in analogy to buffers at an IO device and can be accessed from a *char* driver or *block* or *pipe* or *socket* driver.



Linux drivers

- Char (For driving a character)
- Block (For driving a block of char)
- Input (For standard IO devices)
- Media (For standard media device functions)
- Video (For standard video device functions)
- Sound (For standard auido device functions)

Linux Internals and Device Drivers and Linux Network Functions

- Linux has internal functions called *Internals*. Internals exist for the device-drivers and network-management functions.
- Useful *Linux drivers* for the embedded system and gives the uses of each.

Linux drivers in the *net* directory

The Linux internal functions exist for

- Sockets,
- Handling of Socket buffers,
- firewalls,
- network Protocols (for examples, NFS, IP, IPv6 and Ethernet) and
- bridges.

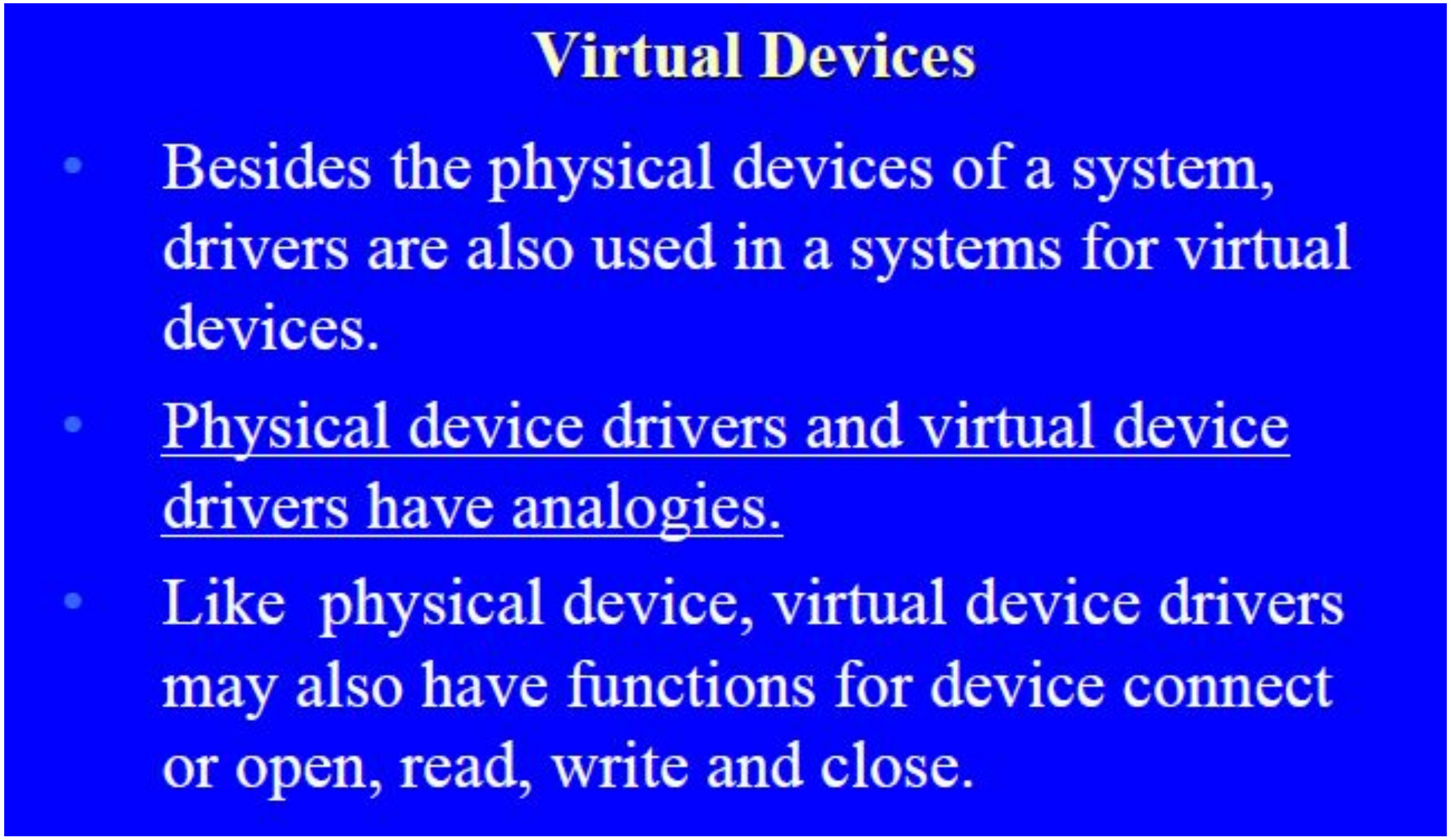
- They work separately as drivers and also form a part of the network management function of the operating system.

Virtual device driver

- Definition : A virtual-device driver is the component of a device driver that communicates directly between an application and memory or a physical device.
- Virtual device driver controls the flow of data
- Allows more than one application to access the same memory or physical device without conflict.

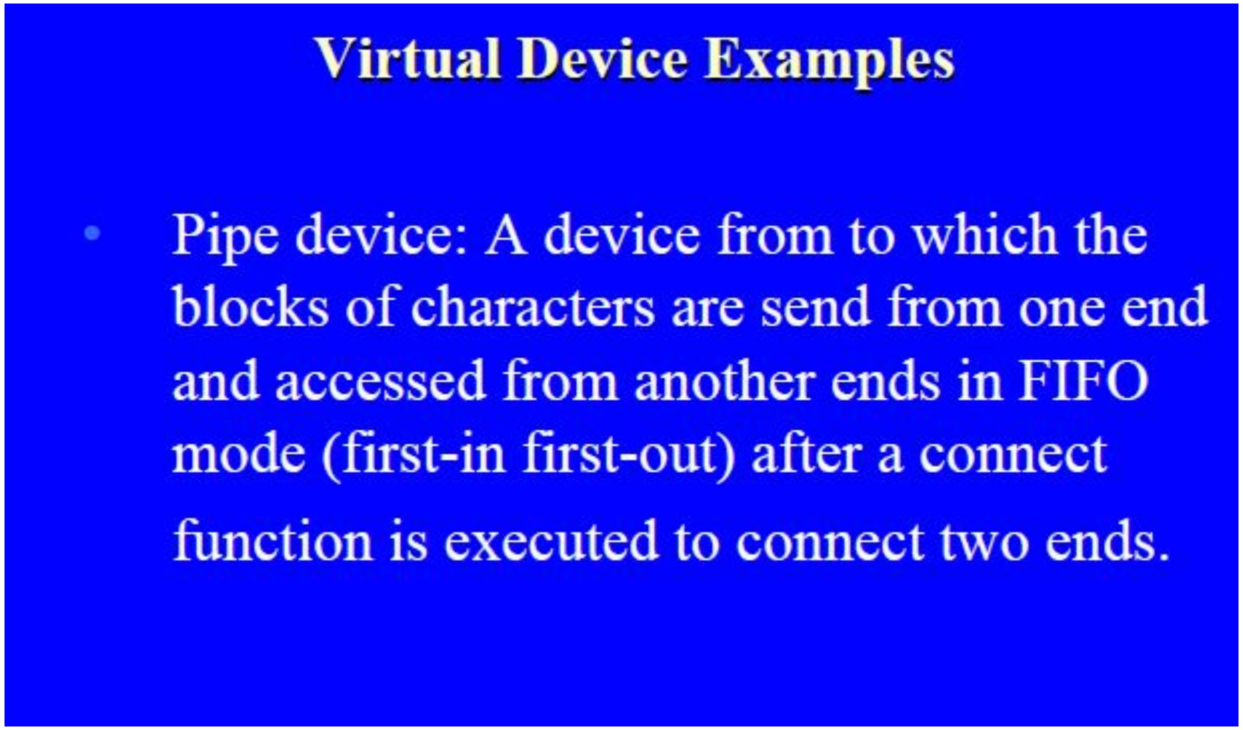
Virtual device drivers

- used to emulate a hardware device, particularly in virtualization environments
- Instead of enabling the guest operating system to dialog with hardware, virtual device drivers take the opposite role and emulate a piece of hardware, so that the guest operating system and its drivers running inside a virtual machine can have the illusion of accessing real hardware



Virtual Devices

- Besides the physical devices of a system, drivers are also used in a systems for virtual devices.
- Physical device drivers and virtual device drivers have analogies.
- Like physical device, virtual device drivers may also have functions for device connect or open, read, write and close.

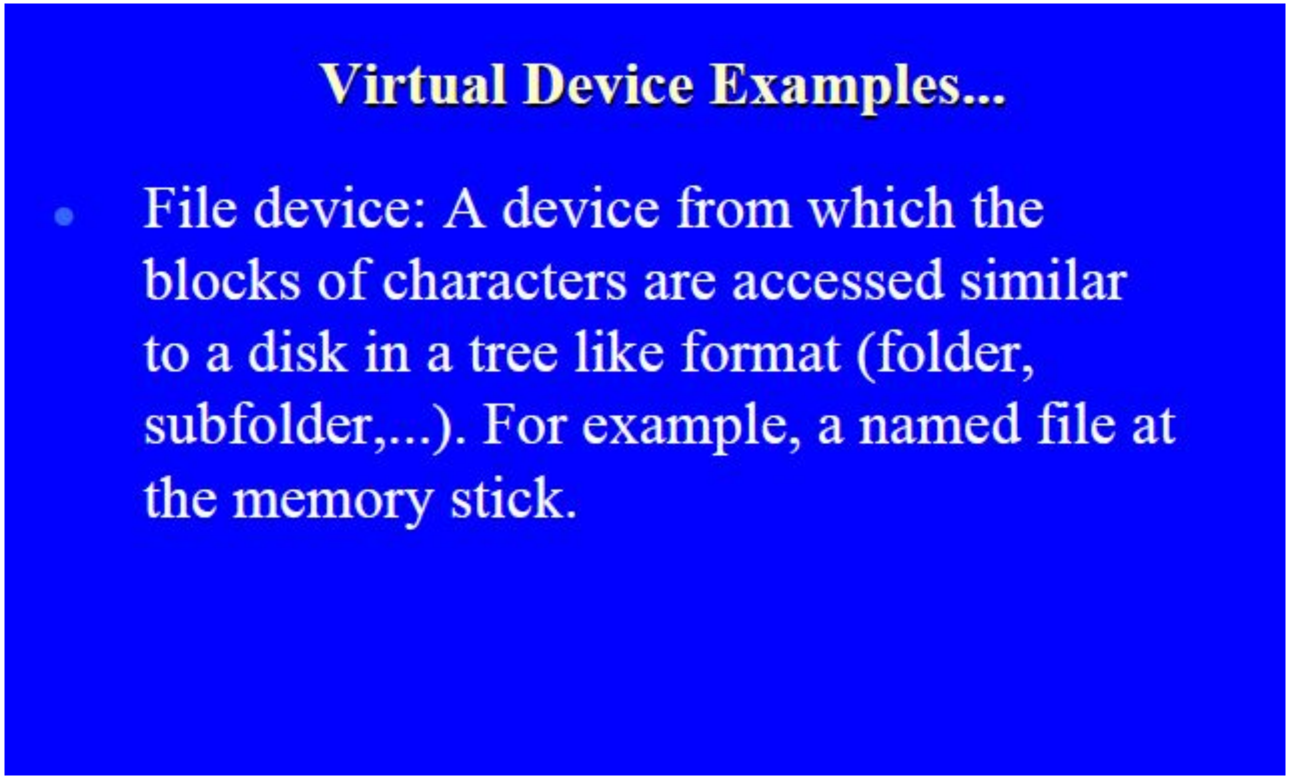


Virtual Device Examples

- Pipe device: A device from to which the blocks of characters are send from one end and accessed from another ends in FIFO mode (first-in first-out) after a connect function is executed to connect two ends.

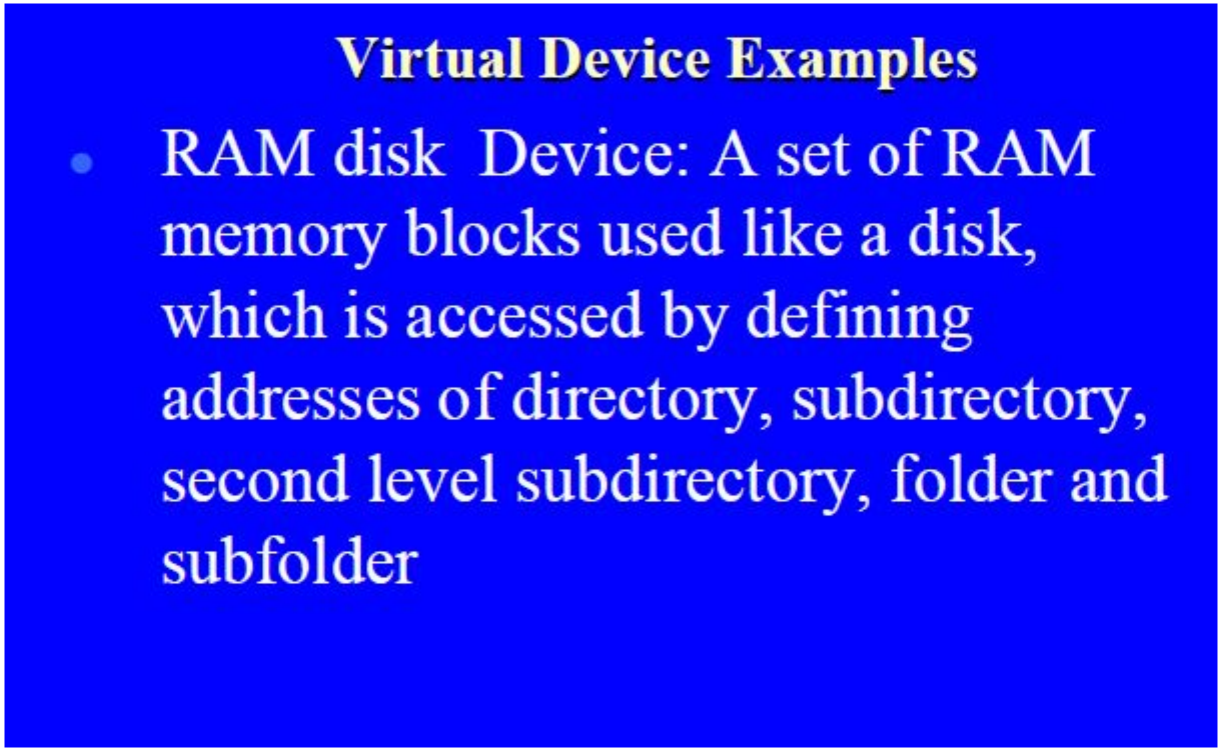
Virtual Device Examples ...

- Socket device: A device from to which (a) the blocks of characters are send from one end with a set of the port (application) and sender addresses, (b) accessed from another end port (application) and receiver addresses, (c) access is in FIFO mode (first-in first-out) only after a connect function is executed to connect two sockets.



Virtual Device Examples...

- File device: A device from which the blocks of characters are accessed similar to a disk in a tree like format (folder, subfolder,...). For example, a named file at the memory stick.



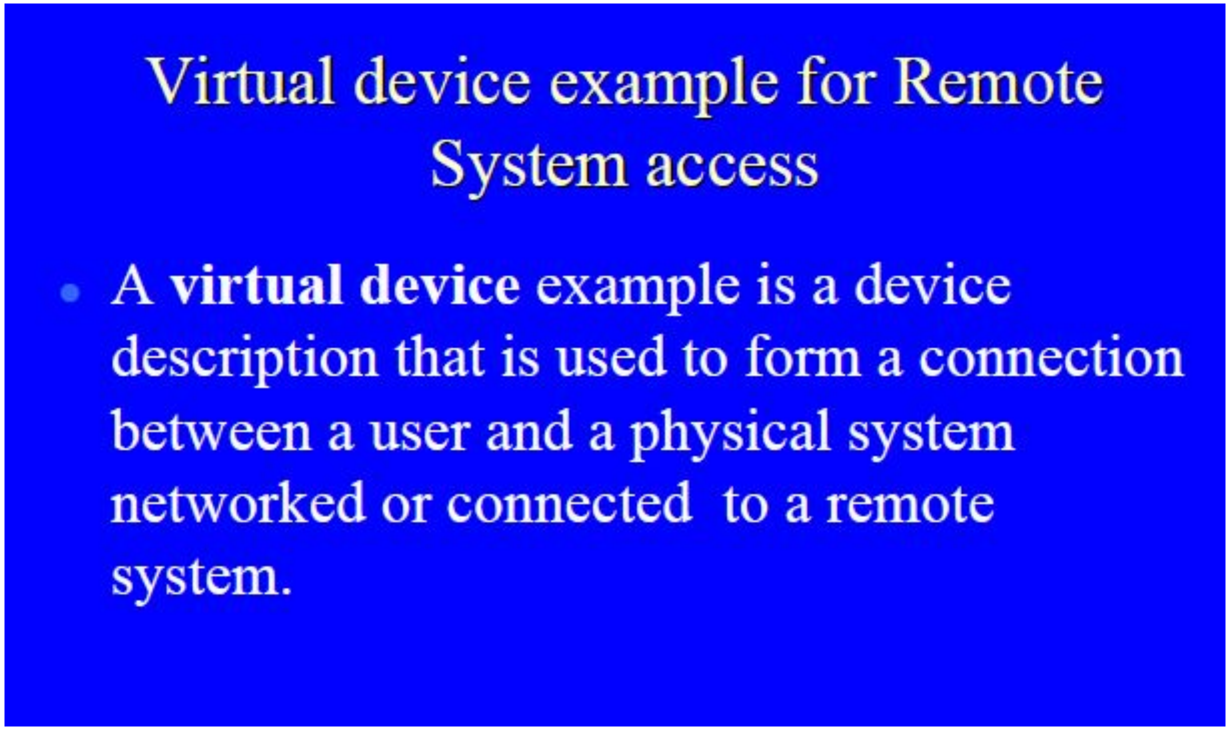
Virtual Device Examples

- RAM disk Device: A set of RAM memory blocks used like a disk, which is accessed by defining addresses of directory, subdirectory, second level subdirectory, folder and subfolder

Difference between various types of virtual devices

- Pipe needs one address at an end,
- Socket one addresses and one port number at an end, and
- File and disk can have multiple addresses. Reading and writing into a file is from or to current cursor address in the currently open folder.

- Just as a file is sent *read* call, a device must be sent a driver command when its input buffer(s) is to be read.
- Just as a file is sent *write* call, a device needs to be sent a driver command when its output buffer is to be written.



Virtual device example for Remote System access

- A **virtual device** example is a device description that is used to form a connection between a user and a physical system networked or connected to a remote system.

Virtual device driver File name (VxD)

- Driver filename in Windows OS is used where the V stands for virtual and D stands for device.The “d" can be replaced with other characters; for example, VdD means a display driver.

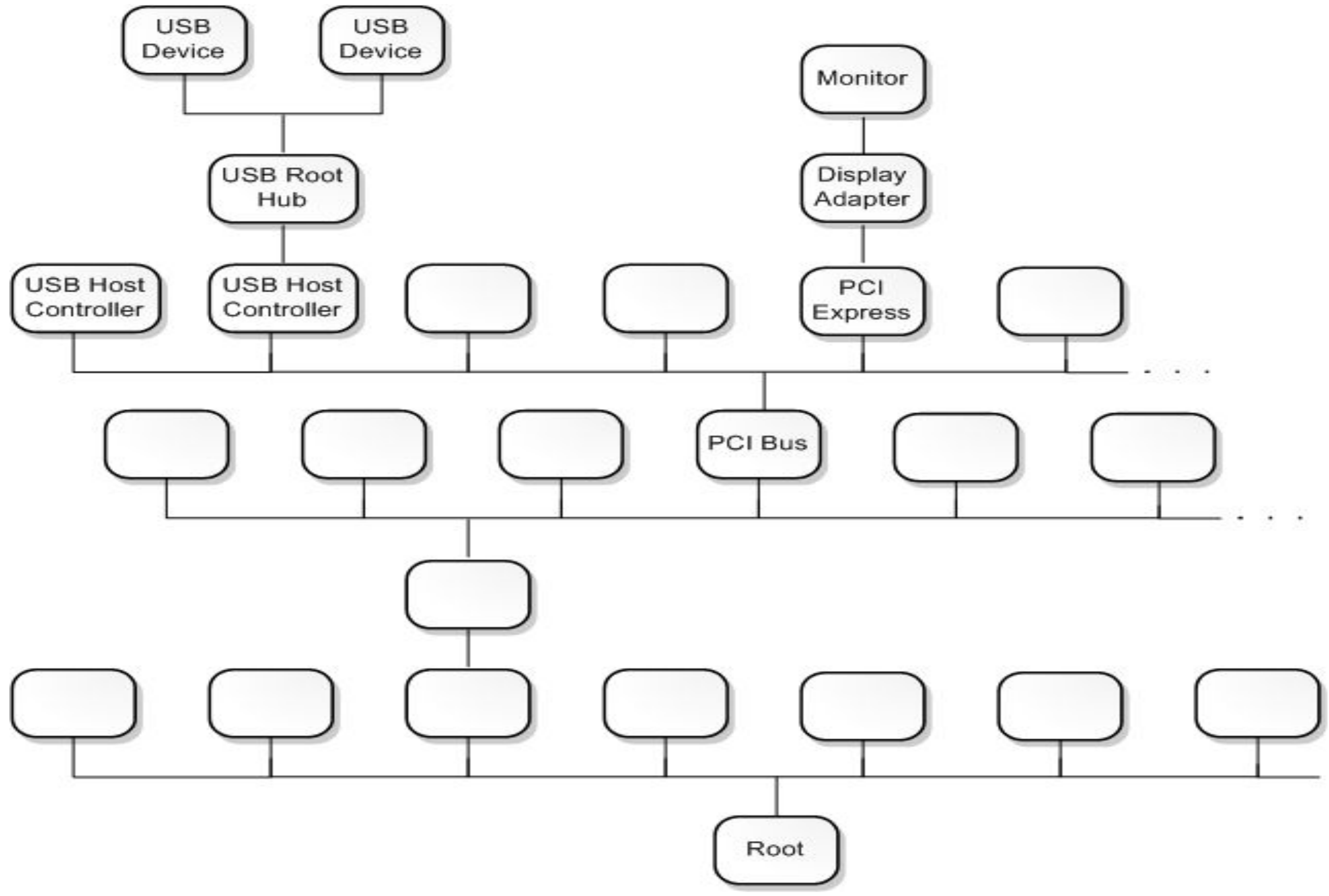
VxD(virtual xxx driver)

- A virtual device driver (VxD) is a software device driver that emulates hardware and other devices so that multiple applications running in protected mode can access hardware interrupt channels, hardware resources and memory without causing conflicts.
- To share arbitrary physical resources amongst these virtual machines, Microsoft introduced dynamically-loadable virtual device drivers.

- In Windows, a device driver is controlled by the operating system's virtual device driver manager (VDDM) and is shared by the applications running within that kernel
- There was no hardware device sharing allowed in most standard DOS applications, so the virtual device driver (VxD) was introduced to prevent device access conflicts. The VxD passed interrupt and memory requests through to the kernel, which in turn allocated the resources as required, always ensuring only a single request thread could access a single interrupt channel of any device at any one time

Device nodes and device stacks

- Windows organizes devices in a tree structure called the *device tree*
- node in the device tree is called a *device node*



Device objects and device stacks

- A *device object* is an instance of a **DEVICE_OBJECT** structure.
- Each device node has an ordered list of device objects, and each of these device objects is associated with a driver.
- The ordered list of device objects, along with their associated drivers, is called the *device stack* for the device node.
- first device object to be created in the device stack is at the bottom, and the last device object to be created and attached to the device stack is at the top

Driver stacks

- In the Windows operating system, WDM drivers are layered in a vertical calling sequence that is called a *driver stack*.
- The topmost driver in the stack typically receives I/O requests from user applications, after the requests have passed through the operating system's I/O manager. The lower driver layers typically communicate with computer hardware.

Device stacks

- Each driver stack supports one or more *device stacks*. A device stack is a set of *device objects* that are created from WDM-defined **DEVICE_OBJECT** structures. Each device stack represents one device. Each driver creates a device object for each of its devices and attaches each device object to a device stack. Device stacks are created and removed as devices are plugged in and unplugged, and each time the system is rebooted.
- When a bus driver detects that child devices have been plugged in or unplugged, it informs the Plug and Play (PnP) manager. In response, the PnP manager asks the bus driver to create a physical device object (PDO) for each child device that is connected to the parent device (that is, the bus). The PDO becomes the bottom of a device stack.
- Next, the PnP manager loads function and filter drivers to support each device (if they are not already loaded), and then the PnP manager calls these drivers so that each can create a device object and add it to the top of the device stack. Function drivers create functional device objects (FDOs), and filter drivers create filter device objects (filter DOs).
- When the I/O manager sends an I/O request to a device's drivers, it passes the request to the driver that created the topmost device object in the device stack. If that driver asks the I/O manager to pass the request to the next-lower driver, the I/O manager uses the device stack to determine the next-lower driver. (The next-lower driver is the driver that created the next-lower device object.)

Function drivers, filter drivers, and bus drivers

- Each driver in a device stack plays the role of either function driver, filter driver, or bus driver.
- main driver for the device stack is called the *function driver for the device stack* or the *function driver for the device node*
- function driver is responsible for handling read requests, write requests, and device control requests
- device object that is associated with a function driver is called a *functional device object* (FDO).

- driver at the bottom of the device stack is called the *bus driver for the device stack* or the *bus driver for the device node*.
- A device object that is associated with the bus driver for a device stack is called a *physical device object* (PDO).
- A driver that is above the function driver or in between the bus and function drivers is called a *filter driver*.
- A device object that is associated with a filter driver is called a *filter device object* (Filter DO).

.inf, PnP

- When the drivers for a device are installed, the installer uses information in an information (INF) file to determine which driver is the function driver and which drivers are filters.
- Typically the INF file is provided either by Microsoft or by the hardware vendor.
- After the drivers for a device are installed, the Plug and Play (PnP) manager can determine the function and filter drivers for the device by looking in the registry.

Buses and bus drivers

1. A *bus* is any parent node in the device tree. In other words, any device node that has one or more child nodes is a bus.

The *bus driver* for a device node is the driver associated with the PDO at the bottom of the node's device stack.

2. A *bus* is a physical device that can have other physical devices connected to it. For example, the PCI bus is a set of conductors and circuitry on the motherboard, and devices are connected to the PCI bus by being built into the motherboard or by being plugged into expansion slots. Physical USB devices can be connected to a USB host controller.

A *bus driver* is the function driver associated with a physical bus