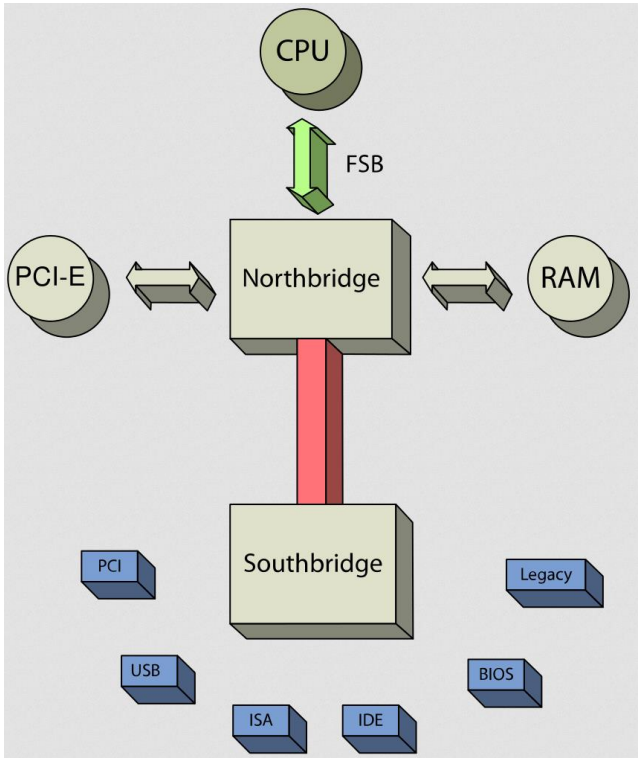
Lecture 11

Device Drivers

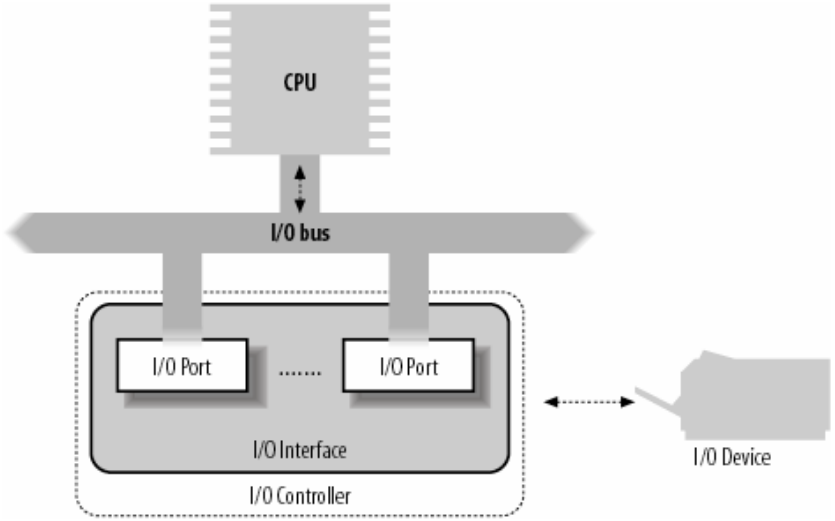
Input-Output

- Any subsystem output of processor and memory is considered I/O
  - Flexible configuration by user
  - OS must handle and abstract diversity
- A typical arrangement in a PC:



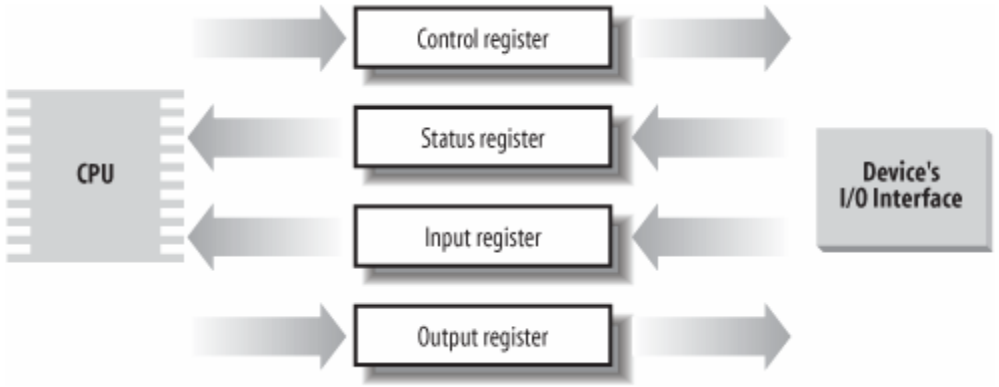
I/O Port I/O on x86

- Intel x86 uses 16 bits to address I/O devices and 8, 16 or 32 bit for data
  - Each of the addressed location is called a “I/O port”
- Four special x86 instructions perform I/O
  - `in, ins, out, outs`
- Transfers data from a device to a register in the processor



Source: Bovet, Daniel P., Understanding the Linux Kernel.

Generic I/O Device



Source: Bovet, Daniel P., Understanding the Linux Kernel.

Resource

- How can one find which I/O port is for what?
- Linux manage this in a tree-like structure rooted at **`ioport_resource`**
  - Use a tree to reflect hierarchy of the device connections
- API's for resource management:
  - **`request_resource()`**
  - **`allocate_resource()`**
  - **`release_resource()`**

I/O interface

- An I/O interface is a hardware circuitry inserted between a group of I/O ports and their corresponding device controller
- Translates the values in the I/ O ports into commands and data for the device
- Detects changes in the device state and correspondingly updates the I/ O port
- May involve a IRQ line to the PIC so as to issue hardware interrupts

Types of I/O interface

- Custom interface – devoted to one specific hardware
  - Example: keyboard, graphics, disk, mouse, network
- General purpose interface
  - Example: parallel port, serial port, PCMCIA, SCSI, USB

Device Controllers

- Interprets the high-level commands received from the I/ O interface and forces the device to execute specific actions by sending proper sequences of electrical signals to it

- Converts and properly interprets the electrical signals received from the device and modifies (through the I/ O interface) the value of the status register

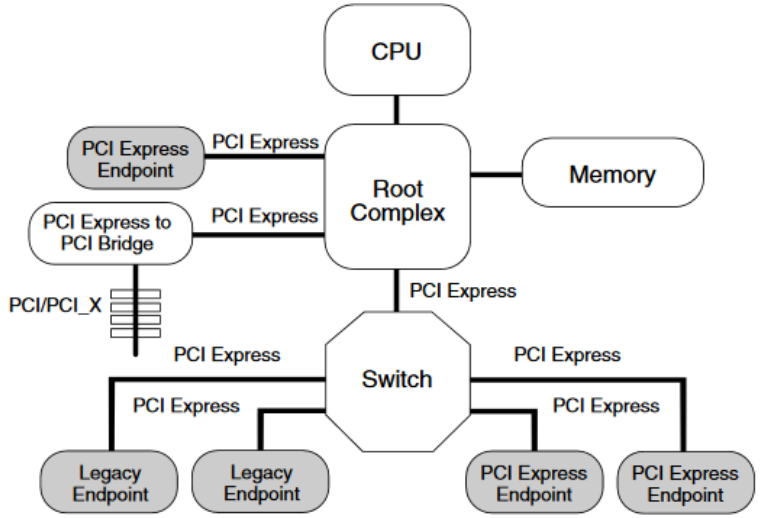
Memory-mapped I/O

- Instead of a distinct space, I/O is embedded inside the memory space
  - Uses loads and stores

- Suitable for fast devices and DMA

PCI Express

- Goal: high performance I/O
- Created in 2004 by a consortium
- Introduces a lane
  - Up to 32 lanes (x1, x2, x4, x8, x12, x16, x32)
  - Each lane is full-duplex transport for a byte
- Complex communication protocol resembling a network protocol



Device Drivers

Device Drivers

- Device drivers
  - Black boxes to hide details of hardware devices
  - Use standardized calls
    - Independent of the specific driver
  - Main role
    - Map standard calls to device-specific operations
  - Can be developed separately from the rest of the kernel
    - Plugged in at runtime when needed

The Role of the Device Driver

- Implements the *mechanisms* to access the hardware
  - E.g., show a disk as an array of data blocks
- Does not force particular *policies* on the user
  - Examples
    - Who many access the drive
    - Whether the drive is accessed via a file system
    - Whether users may mount file systems on the drive

Policy-Free Drivers

- A common practice
  - Support for synchronous/asynchronous operation
  - Be opened multiple times
  - Exploit the full capabilities of the hardware
- Easier user model
- Easier to write and maintain
- To assist users with policies, release device drivers with user programs

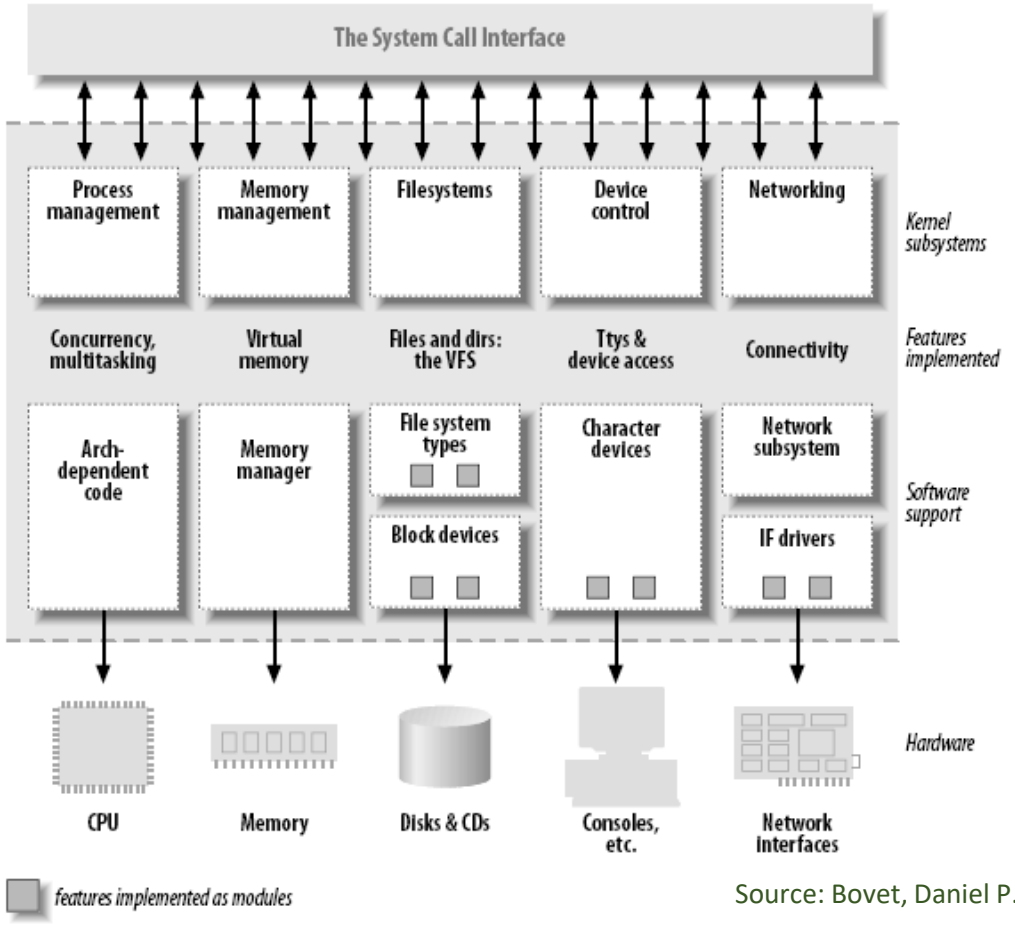
Splitting the Kernel

- Process management
  - Creates, destroys processes
  - Supports communication among processes
    - Signals, pipes, etc.
  - Schedules how processes share the CPU
- Memory management
  - Virtual addressing

Splitting the Kernel

- File systems
  - Everything in UNIX can be treated as a file
  - Linux supports multiple file systems
- Device control
  - Every system operation maps to a physical device
    - Few exceptions: CPU, memory, etc.
- Networking
  - Handles packets
  - Handles routing and network address resolution issues

Splitting the Kernel



Source: Bovet, Daniel P., Understanding the Linux Kernel.

Loadable Modules

- The ability to add and remove kernel features at runtime
- Each unit of extension is called a *module*
- Use **`insmod`** program to add a kernel module
- Use **`rmmod`** program to remove a kernel module

Classes of Devices and Modules

- Character devices
- Block devices
- Network devices
- Others

Character Devices

- Abstraction: a stream of bytes
  - Examples
    - Text console (**`/dev/console`**)
    - Serial ports (**`/dev/ttyS0`**)
  - Usually supports **`open`**, **`close`**, **`read`**, **`write`**
  - Accessed sequentially (in most cases)
  - Might not support file seeks
  - Exception: frame grabbers
    - Can access acquired image using **`mmap`** or **`lseek`**

Block Devices

- Abstraction: array of storage blocks

- However, applications can access a block device in bytes
  - Block and char devices differ only at the kernel level
  - A block device can host a file system

Network Devices

- Abstraction: data packets
- Send and receive packets
  - Do not know about individual connections
- Have unique names (e.g., **`eth0`**)
  - Not in the file system
  - Support protocols and streams related to packet transmission (i.e., no **`read`** and **`write`**)

Other Classes of Devices

- Examples that do not fit to previous categories:
  - USB
  - SCSI
  - FireWire
  - MTD

File System Modules

- Software drivers, not device drivers
- Serve as a layer between user API and block devices
- Intended to be device-independent

Security Issues

- Deliberate vs. incidental damage
- Kernel modules present possibilities for both
- System does only rudimentary checks at module load time
- Relies on limiting privilege to load modules
  - And trusts the driver writers
- Driver writer must be on guard for security problems

Security Issues

- Do not define security policies
  - Provide mechanisms to enforce policies
- Be aware of operations that affect global resources
  - Setting up an interrupt line
    - Could damage hardware
  - Setting up a default block size
    - Could affect other users

Security Issues

- Beware of bugs
  - Buffer overrun
    - Overwriting unrelated data
  - Treat input/parameters with utmost suspicion
  - Uninitialized memory
    - Kernel memory should be zeroed before being made available to a user
    - Otherwise, information leakage could result
      - Passwords

Security Issues

- Avoid running kernels compiled by an untrusted friend
  - Modified kernel could allow anyone to load a module

Version Numbering

- Every software package used in Linux has a release number
  - You need a particular version of one package to run a particular version of another package
  - Prepackaged distribution contains matching versions of various packages

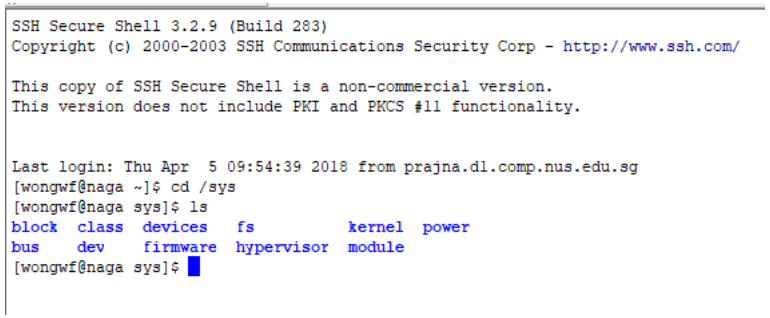
Version Numbering

- Different throughout the years
- After version 1.0 but before 3.0
  - <major>.<minor>.<release>.<bugfix>
  - Time based releases (after two to three months)
- 3.x
  - Moved to 3.0 to commemorate 20th anniversary of Linux
  - <version>.<release>.<bugfix>
  - https://lkml.org/lkml/2011/5/29/204

sysfs

The Linux Driver Model

- The big idea: everything is a file
  - Leverage on VFS
  - Exception: network cards
- Expose the hierarchical relationships between devices
- The components of sysfs
  - **`block`** – block devices
  - **`devices`** – all hardware devices recognized by the kernel
  - **`bus`** – buses
  - **`drivers`** – the device drivers
  - **`class`** – the main category of devices
  - **`power`** – for power management
  - **`firmware`** – handle firmware



Examples

Name	Type	Major	Minor	Description
/dev/fd0	block	2	0	Floppy disk
/dev/hda	block	3	0	First IDE disk
/dev/hda2	block	3	2	Second primary partition of first IDE disk
/dev/hdb	block	3	64	Second IDE disk
/dev/hdb3	block	3	67	Third primary partition of second IDE disk
/dev/ttyp0	char	3	0	Terminal
/dev/console	char	5	1	Console
/dev/lp1	char	6	1	Parallel printer
/dev/ttyS0	char	4	64	First serial port
/dev/rtc	char	10	135	Real-time clock
/dev/null	char	1	3	Null device (black hole)

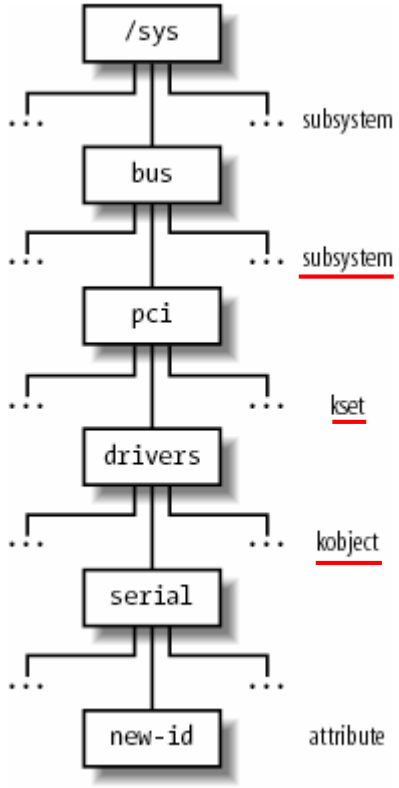
Character vs block devices

- The data of a block device can be addressed randomly, and the time needed to transfer a data block is small and roughly the same
  - Examples: hard disks, floppy disks , CD-ROM drives, and DVD players

- The data of a character device either
  - cannot be addressed randomly,
  - or they can be addressed randomly, but the time required to access a random datum largely depends on its position inside the device

Kobjects

- Core data structure for device drivers
- Each kobject corresponds to a directory in sysfs
- Embedded inside containers
  - Descriptors for buses, devices, and drivers
- Kobjects are grouped into ksets of the same type (container)
- Ksets may form subsystems (hierarchy)
- All the above must be registered with the kernel



Source: Bovet, Daniel P., Understanding the Linux Kernel.

Components of the Linux device model (1)

- Each device has a **`device`** object
  - Associates with a kobject
- Each driver has a **`device_driver`** object

Type	Field	Description
`char *`	`name`	Name of the device driver
`struct bus_type *`	`bus`	Pointer to descriptor of the bus that hosts the supported devices
`struct semaphore`	`unload_sem`	Semaphore to forbid device driver unloading; it is released when the reference counter reaches zero
`struct kobject`	`kobj`	Embedded kobject
`struct list_head`	`devices`	Head of the list including all devices supported by the driver
struct module *	owner	Identifies the module that implements the device driver, if any (see Appendix B)
`int (*)(struct device *)`	`probe`	Method for probing a device (checking that it can be handled by the device driver)
`int (*)(struct device *)`	`remove`	Method invoked on a device when it is removed
`void (*)(struct device *)`	`shutdown`	Method invoked on a device when it is powered off (shut down)
`int (*)(struct device *, unsigned long, unsigned long)`	`suspend`	Method invoked on a device when it is put in low-power state
`int (*)(struct device *, unsigned long)`	`resume`	Method invoked on a device when it is put back in the normal state (full power)

Fields in **`device_driver`**

Components of the Linux device model (2)

- Buses are represented by **`bus_type`** objects

Type	Field	Description
`char *`	`name`	Name of the bus type
`struct subsystem`	`subsys`	Kobject subsystem associated with this bus type
`struct kset`	`drivers`	The set of kobjects of the drivers
`struct kset`	`devices`	The set of kobjects of the devices
struct bus_attribute *	bus_attrs	Pointer to the object including the bus attributes and the methods for exporting them to the *sysfs* filesystem
struct device_attribute *	dev_attrs	Pointer to the object including the device attributes and the methods for exporting them to the *sysfs* filesystem
struct driver_attribute *	drv_attrs	Pointer to the object including the device driver attributes and the methods for exporting them to the *sysfs* filesystem
`int (*)(struct device *, struct device_driver *)`	`match`	Method for checking whether a given driver supports a given device
`int (*)(struct device *, char **, int, char *, int)`	`hotplug`	Method invoked when a device is being registered

Fields in **`bus_type`**

Components of the Linux device model (3)

- Each class of devices has a **`class`** object
- All **`class`** objects belong to the **`class_subsys`** subsystem associated with **`/sys/class`**
- One **`class_device`** descriptor for each logical device of the class
- A **`device`** can belong to multiple **`class_device`**
  - Example: a sound card is a hardware device that usually includes a DSP, a mixer, a game port interface, and so on.
- Device drivers in the same class are expected to offer the same functionalities to the user mode applications

Device number

- Linux maintains an official list of allocated device numbers
  - Documentation/devices.txt
- Major number
  - 12 bits
- Minor number
  - 20 bits
- Each logical device in the system should have an associated device file with a well-defined device number
- When a device driver registers, it will specify (static) or request to be allocated (dynamic) a range of device numbers that it will handle

Dynamic device file creation

- Simply too many possible devices
- From kernel 2.6 – create on demand dynamically
- Uses the **`udev`** toolset
- Supports device hotplugging

The role of VFS

- Device files exist in the system directory but are intrinsically different from ordinary files
  - VFS hides the difference

- VFS changes the default file operations of a device file when it is opened to do the appropriate device operations

Monitoring I/O

- Duration of I/O is unpredictable
  - Question: how do we know an I/O request is terminated or completed?
- Two modes:
  - Polling

```
for (;;) {
    if (read_status( device) & DEVICE_END_OPERATION) break;
        if (--count = = 0) break;
}
```

  - Interrupt
    - Lower overhead
    - Can only be done if device has hardware for doing a IRQ

Character Device Drivers

Character devices

- Usually simpler hardware
- No caching
- No re-reads of same bytes
- No sophisticated communication protocol involved

Character device drivers

- Kernel maintains a hash table chrdevs containing intervals of device numbers with handlers
- On initialization, a character device driver much specify what range of device numbers it will handle, or ask for that range to be dynamically allocated
- Character devices can transfer large number of bytes in a single I/O
  - Use buffering
  - Good news: same buffer never refused – makes for easier management

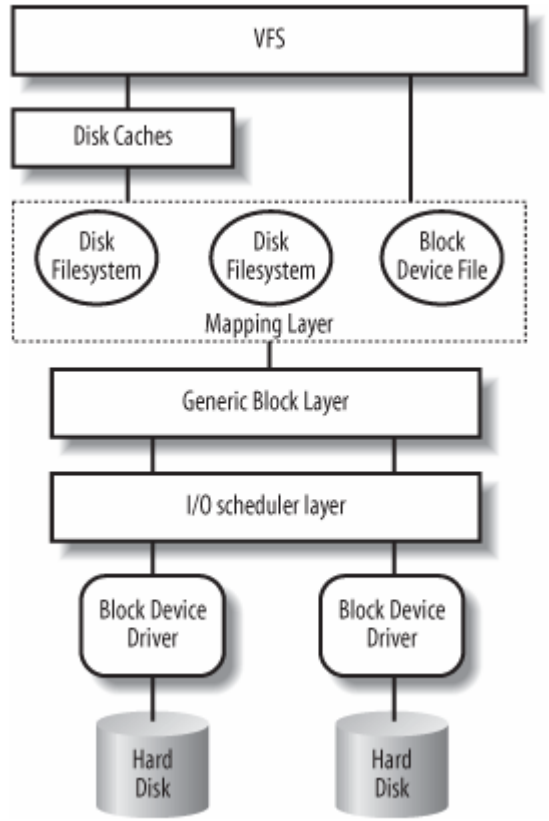
Block Device Drivers

Block devices

- Usually more complex hardware
- Access is slow
- Applications can repeatedly read or write the same block of data
- Example: disks

Block I/O

- VFS needs to check if block is already in some kind of caches
  - If yes, serves request from memory cache
- If no, determine physical location of block via mapping layer
  - From block size, compute file block numbers
  - From file's inode data determine position on disk in terms of logical block number
- Issues operations to a generic block layer which issues block I/O requests
- I/O scheduler layer implements I/O policies
- Finally, block device drivers perform actual operation on device

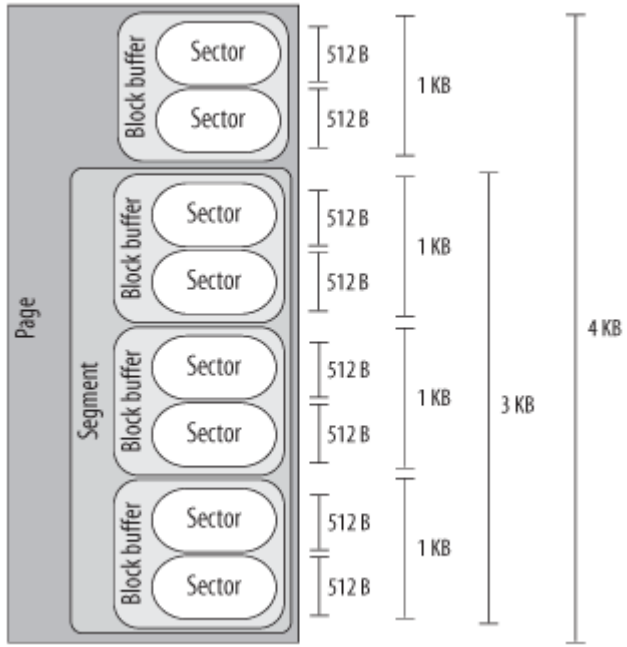


Source: Bovet, Daniel P., Understanding the Linux Kernel.

Size matters

- Hardware transfers data in sectors
- VFS uses logical units called blocks
- Block device drivers can work with adjacent data in segments
- Disk caches work with pages

Source: Bovet, Daniel P., Understanding the Linux Kernel.



Typical sizes

The Generic Block Layer

- Put data into buffers
- “Zero copy” – directly put data into user space
- Manage logical volumes
  - LVM, RAID
- Exploit advance features in modern disk controllers
- The `bio` structure contains all information regarding an ongoing block I/O operation

The I/O Scheduler

- Generic block layer fires off random block I/O operations. But random access is bad for performance
- I/O Scheduler tries to group I/O operations for best performance
- However, deferring some I/O operations can lead to complications
  - Throw in the fact that block I/O are asynchronous and interrupt driven and you see the complications that the I/O scheduler has to deal with
- In implementing any scheduling policies, I/O scheduler must ensure deadlocks etc. won't arise

How I/O scheduling works

- Generic block layer invokes I/O scheduler to determine where a new request should be added to a request queue

- I/O scheduler tries to keep request queue sorted by sector
  - Reduces disk seeking
  - Resembles elevator scheduling: hence the I/O schedulers are known as elevators

- After computing priorities, I/O scheduler will finally insert into a dispatch queue – the actual ordering of requests that a device driver should process

The Four I/O Scheduler Algorithms (1)

- NOOP elevator: simplest, no sorting, either FIFO or LIFO
- Complete Fairness Queuing (CFQ) elevator: maintains a large number of queues (default 64)
  - Processes are hashed to queues by process IDs
  - Each queue is sorted by sectors

The Four I/O Scheduler Algorithms (2)

- Deadline elevator: 4 queues (1 read sorted by sector, 1 write sorted by sector, 1 read sorted by deadline, 1 write sorted by deadline)
  - A read (write) request will enter BOTH the read-by-sector (write-by-sector) and read-by-deadline (write-by-deadline)
  - A read request expires in 500 ms after entry into queue; 5 sec for write
  - First (using heuristic) determine whether to do read or write for the next batch of entries in the dispatch queue
    - Reads preferred, unless writes skipped over for too long
  - If, say, read is chosen, check read deadline queue. If head of that queue has expired, dispatch that. Then try to choose requests adjacent to this from the sector queue.
  - If no expired request, dispatch from where last left out in sector queue
    - Reset to head when end of queue is reached

The Four I/O Scheduler Algorithms (3)

- Anticipatory elevator: variant of the deadline elevator, also use the same 4 queues
  - Default expiry times changed to 125 ms for read, 250 for write
  - Instead of one-way moving, may backtrack
    - If seek distance going backward is less than ½ of that going forward
  - Collects statistics about the I/O pattern of individual processes
    - If process P usually do 2 reads, then try to issue 2 reads from P at the same time
    - If only 1 read from P is in the queue, defer it for a short while

END