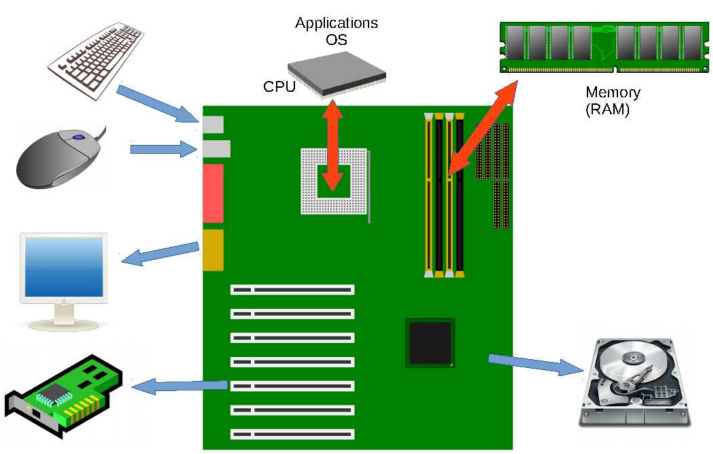
Operating Systems

V. Input / Output

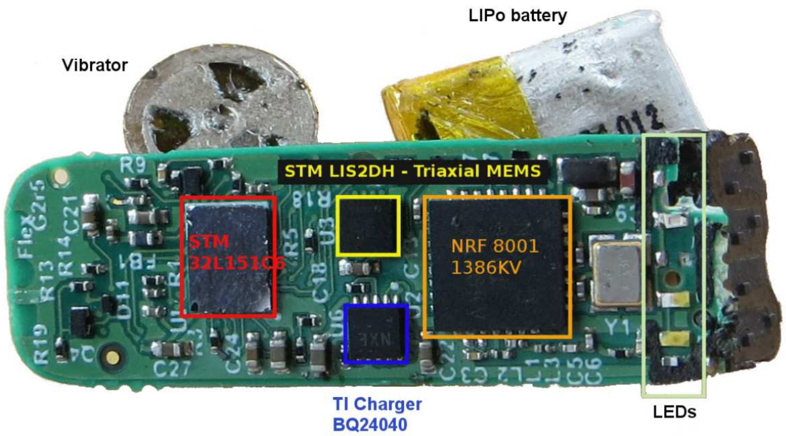
Ludovic Apvrille
ludovic.apvrille@telecom-paris.fr
Eurecom, office 470

perso.telecom-paris.fr/apvrille/OS/

Devices of a Computer System



Fitbit: What are the I/O Components?



Devices | From a hardware point of view | From a software point of view | Examples: disks, device drivers

Issues

OS manages devices

- Protection
- Sharing
- Ease-of-use
- Performance

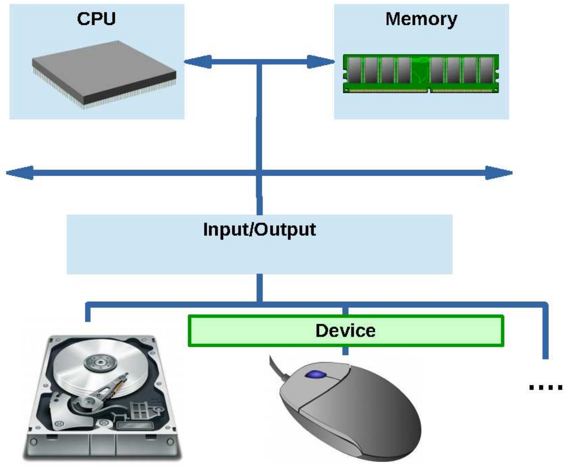
OS = interface between devices and other parts of the system

If possible, device-independent interface

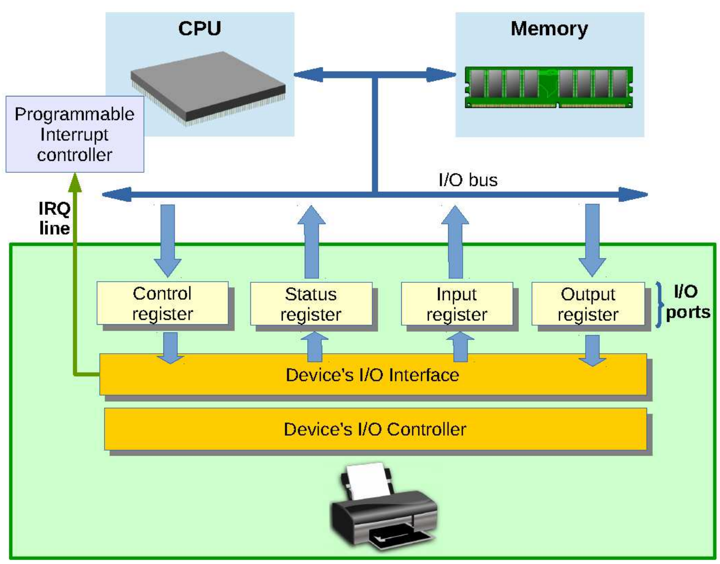
- OS sends commands to devices
- OS gets information / results from devices
- OS handles errors

→ High fraction of all lines of code of OS

I/O Architecture



I/O Architecture (Zoom)



I/O Ports

- Each device connected to the I/O bus has its own set of I/O addresses
- Read / write: assembly language instructions
  - If device has special registers: *in*, *ins*, *out*, *outs*
    - Privileged assembly instructions (use of syscalls for user processes)
  - If I/O ports mapped in main memory: *mov*, etc.
    - Protected memory

I/O Interface

= Hardware circuit between a group of I/O ports and the corresponding device controller

- Values in the I/O ports translated to commands and data
- Detects changes in the device's state
  - Updates I/O ports accordingly
  - May send an interrupt request through an IRQ line

Two types of interfaces

- Custom I/O interfaces
  - For one specific hardware: keyboard, disk interface, etc.
- General-purpose
  - USB, Thunderbolt, SATA (Serial Advanced Technology Attachment)

Device Controller

Interprets high-level commands received from the I/O interface

Executes specific actions by issuing proper sequences of electrical signals

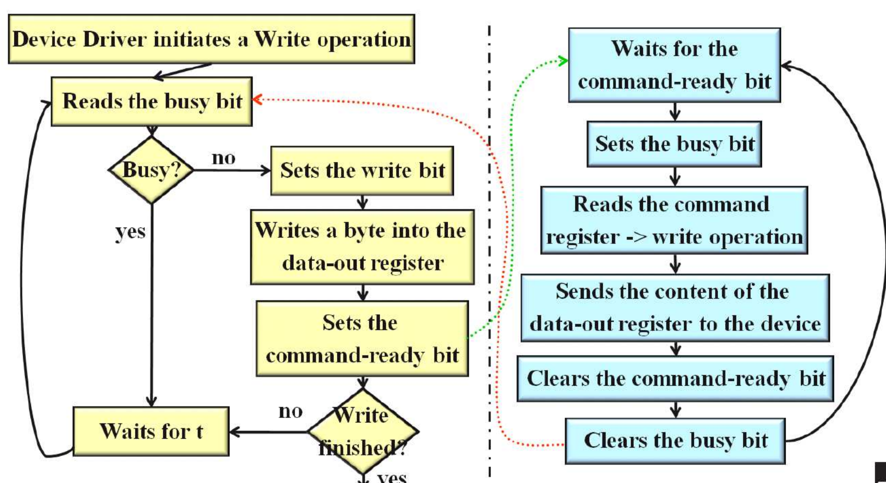
Interprets signals coming from the device

Modifies the value of the status register through the I/O interface

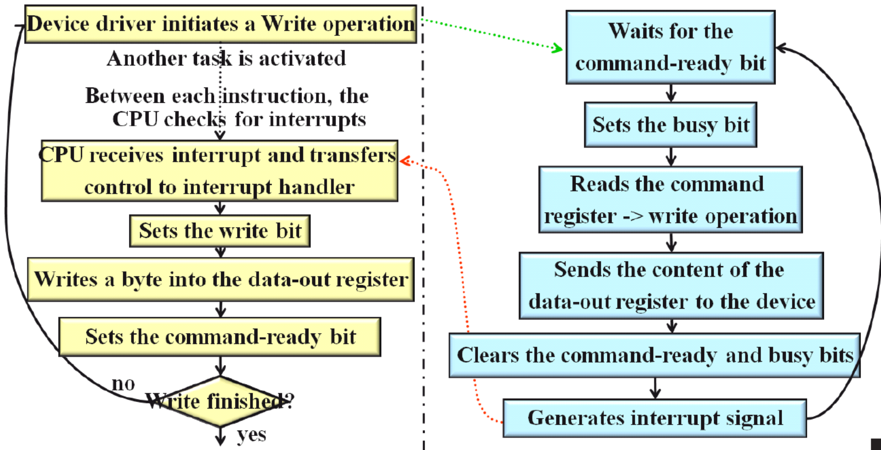
Example: Disk controller

- Receives commands such as "Write this block of data"
  - Issues low-level orders such as "Position the disk head on the right track", etc.
- Very complex because management of files stored in local cache memory

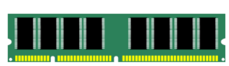
Polling: OS and Device Controller



Interrupt: OS and Device Controller



Memory Transfer



- From main memory to a device memory
- From a device memory to main memory

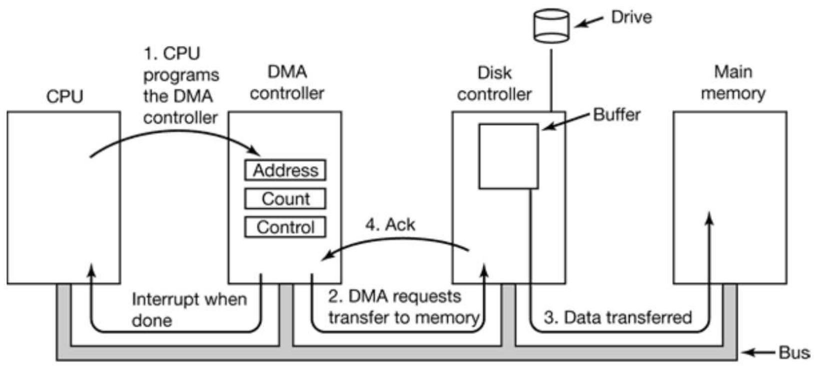
With e.g. *mov* instructions

- Efficient for a small amount of data
- Quite inefficient for a large amount of data
  - CPU is busy during the whole transfer

→ Direct Memory Access

- Direct Memory Access Controller: External entity which can be programmed to transfer data
- DMAC programming is expensive, its use is reserved for large transfers

Steps in a DMA Transfer



(Source: Tanenbaum)

Question: In the figure, what is in fact meant by "CPU"?

Characteristics of Devices

Character stream vs. block stream

- **Character**: data transfer of one byte only
- **Block**: data transfer of blocks of bytes

Sequential access vs. random access

- **Sequential**: data transfer done in a fixed order forced by the device
- **Random**: data transfer can concern any of the available data

Synchronous vs. asynchronous

- **Synchronous**: predictable response times for data transfer
- **Asynchronous**: irregular or unpredictable response times

Characteristics of Devices (Cont.)

Sharable vs. dedicated

- **Sharable**: Can be used concurrently by several processes
- **Dedicated**: Cannot be used concurrently

Speed of operations

Speed ranges from a few bytes per second to gigabytes per second

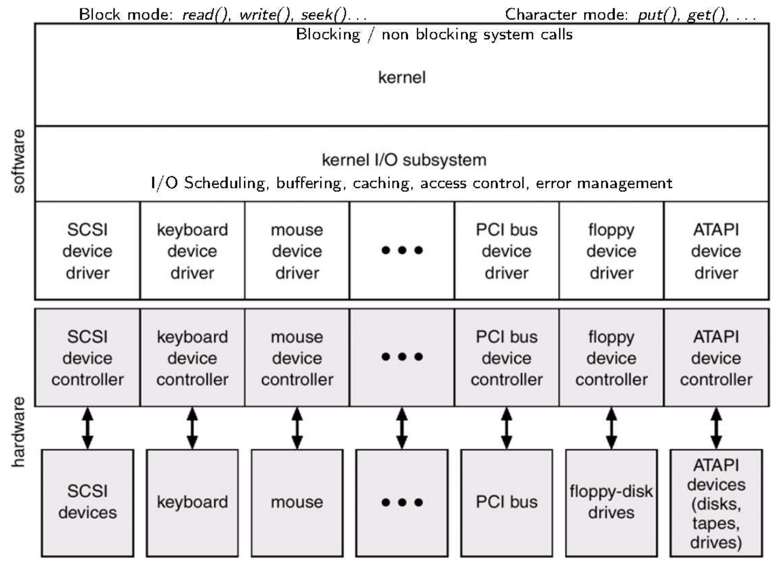
Read-write, read only, write only

- Some devices perform both input and output
- Others support only one data direction

Examples

Characteristic	Variation	Example
Data-transfer mode	Character Block	
Access method	Sequential Random	
Transfer schedule	Synchronous Asynchronous	
Sharing	Dedicated Shareable	
Device speed	Latency Transfer rate Delay between operation	
I/O direction	Read only Write only Read and Write	

Kernel I/O Subsystem



Blocking and Non-Blocking System Calls

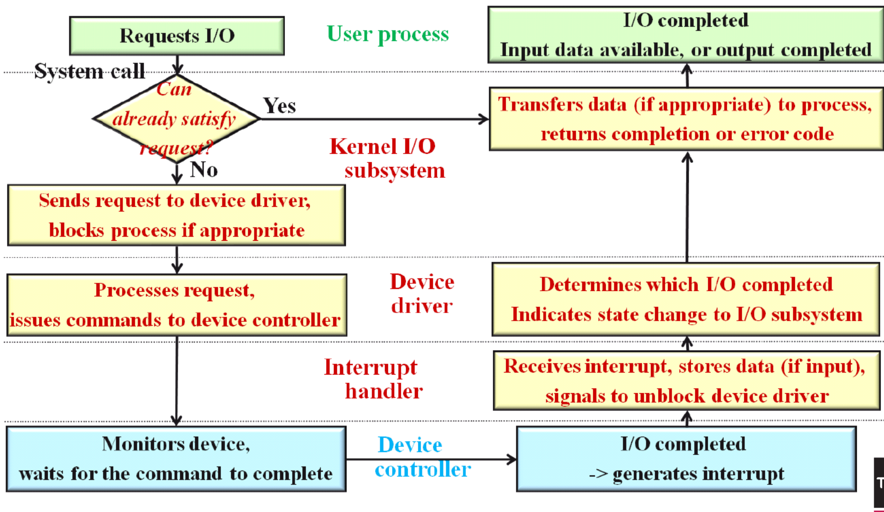
Blocking system call

- Application is suspended
- Application is moved to a wait queue
- When the I/O is completed, the application is moved to a run queue

Non-blocking system call

- E.g., used for keyboard / mouse input
- Returns immediately
- Answer is provided as ...
  - Modification of variables
  - Signal
  - Call-back routine

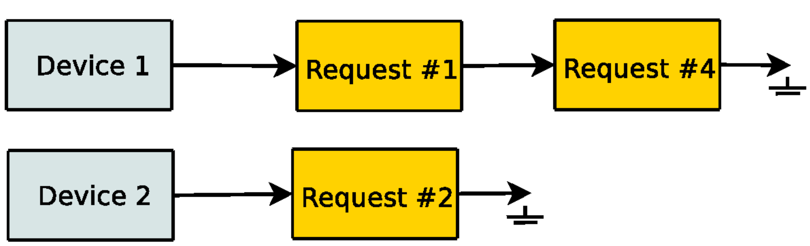
Blocking I/O Request Handling



Kernel I/O Subsystem: Scheduling

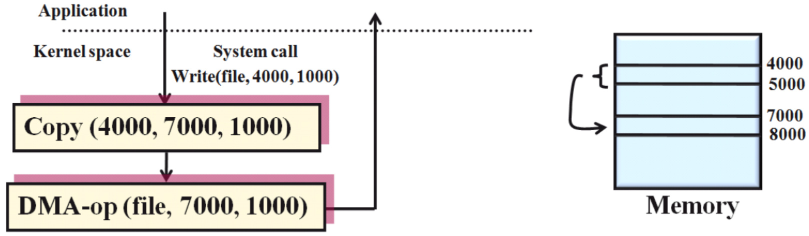
Classification of I/O requests

- I/O requests are listed by device
- Apply scheduling algorithms on requests
  - Algorithms defined according to device characteristics



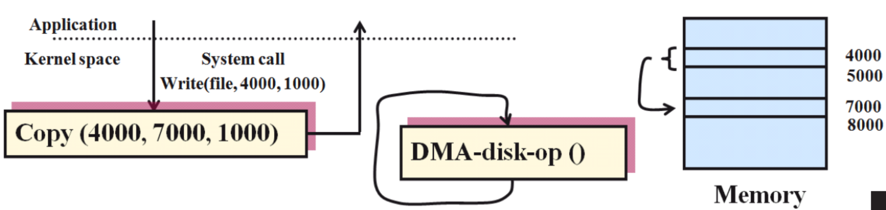
Kernel I/O Subsystem: Buffering

- What's for?
  - Fragmentation or reassembly of data
  - Manipulation of large set of data (more efficient)
- Copy semantics issue!
  - Data that must be copied to a device should be put in another buffer before returning to the application ... Why?



Kernel I/O Subsystem: Caching

- **Cache** = Memory that holds on a faster storage a copy of data stored elsewhere
- Main memory can be used to increase I/O operations efficiency
  - For example, files can be cached in main memory
    - Physical I/O operations on disk are deferred
    - Write operation = buffer copy



Kernel I/O Subsystem: Access Control

Spooling

- Ex: printer
  - Each job is independent and should not interfere with another one
- OS uses lists of jobs for each device which requires spooling
  - FIFO policy: each job is sent one after the other
  - Other policies might be used ...

Exclusive device access

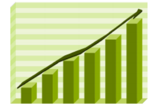
- A process can allocate a device
- If the device is not idle, the process might be queued (Windows) or an error is returned
- The process which obtains the device can authorize other devices / processes / users to access it
  - Parameter in system call

Kernel I/O Subsystem: Error Handling

- I/O system calls return information about I/O completion
- May return:

1. 1 bit of information
2. Error number (UNIX)
3. Sometimes, the device provides more information (it also depends on the interface)

Performance Issue

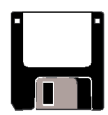


I/O is a major factor in system performance!

How to improve performance?

- Reduce the number of context switches
- Reduce frequency of interrupts
- Use polling if busy-waiting can be minimized
- Increase concurrency with DMA
- Increase the efficiency of large transfers with DMA
- Reduce the number of buffer copies
- Move processing primitives in hardware whenever possible

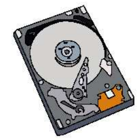
Storage Supports



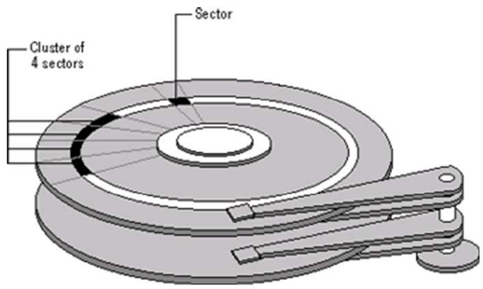
Type	Capacity	Access time	Transfer rate	Price per Byte	Usage
Tape	Huge	Poor	High	Low	Backup
HDD					
SSD					

- **Do you know other storage devices? What about their characteristics?**

Disk Formatting



- Low-level formatting
  - Organization in sectors
- Logical formatting
  - File-system data structure is stored on the disk
    - FAT, FAT32, NTFS, ZFS, etc.



Disk Scheduling

How to efficiently schedule requests on disks?

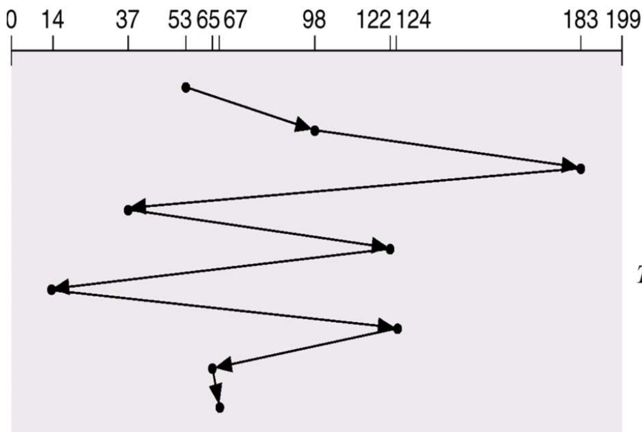
→ Understand how hardware works, and use it efficiently

- Seek time
  - Time to move the disk arm (head) to the right cylinder
- Rotational latency
  - Time to move to the desired sector
- Bandwidth
  - Total number of bytes transferred divided by the total time between the first request for service and the completion of the last transfer

Let's work on the seek time!

Disk Scheduling: First-Come First-Served (FCFS)

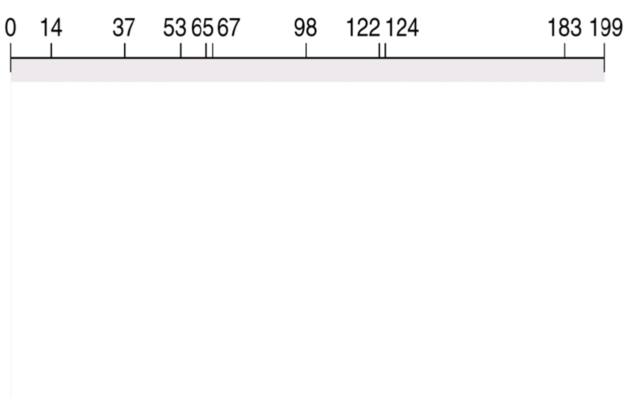
queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



Total movement: 640 cylinders

Disk Scheduling: Shortest-Seek-Time-First (SSTF)

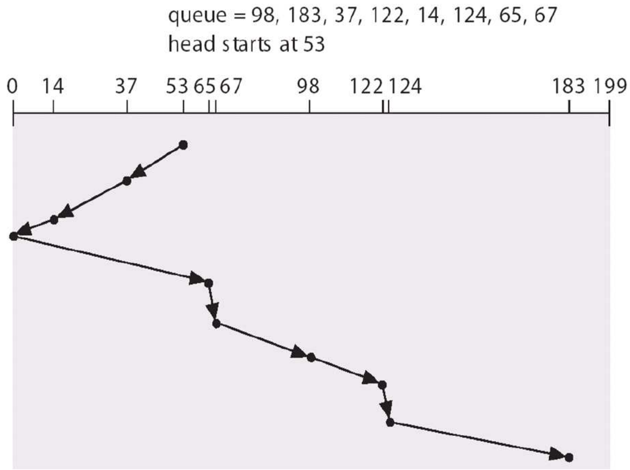
queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



Total movement: 236 cylinders

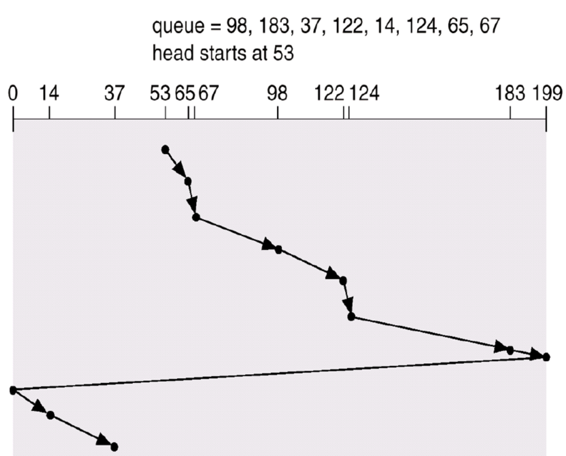
Limitations?

Disk Scheduling: SCAN (Elevator Algo.)

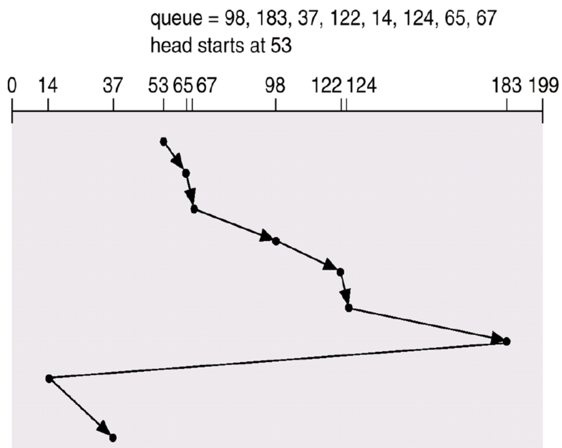


Total movement:
?

Disk Scheduling: C-SCAN



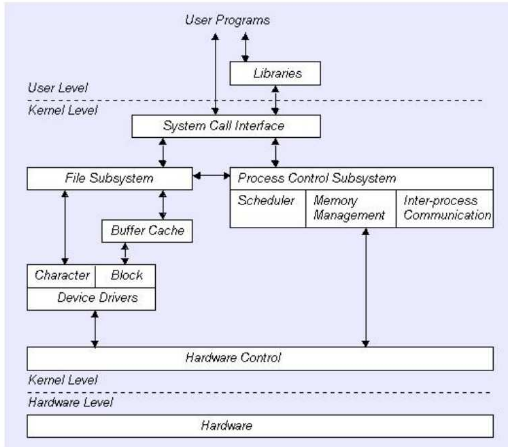
Disk Scheduling: C-LOOK



Disk Scheduling: Conclusion

- SSTF is common
- If heavy load on the disk:
  - SCAN and C-SCAN perform better
- Performance depends on the number and types of requests
- Requests for disk service can be influenced by the file-allocation method
- Either SSTF or LOOK is a reasonable choice for the default algorithm
- The disk-scheduling algorithm should be written as a separate module of the Operating System, allowing it to be replaced with a different algorithm if necessary
- A disk-operation scheduling algorithm is sometimes provided within the disk

Device Drivers in the Linux Kernel



This picture is excerpted from *Write a Linux Hardware Device Driver*, Andrew O'Shauqhnessy, Unix world

Device Files

- Devices can be accessed throughout a file-based interface
  - *write(), read()*
  - */dev/lp0*
- Type
  - Either block or character
- Major number
  - 1 . . . 255: identifies the device type
  - Same major number and same type $\Rightarrow$ same device driver
- Minor number
  - Specific device among a group of devices sharing common features

Examples of Device Files

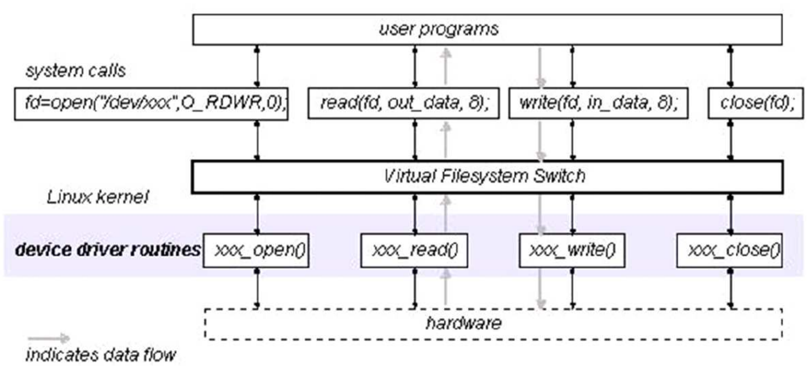
Block device *Major number* *Minor number*

```
[apvrille@rocky][/dev]$ ls -l sda*
brw-rw---- 1 root disk 8, 0 Oct 27 14:41 sda
brw-rw---- 1 root disk 8, 1 Oct 27 14:41 sda1
brw-rw---- 1 root disk 8, 2 Oct 27 14:41 sda2

[apvrille@rocky][/dev]$ ls -al rtc*
lrwxrwxrwx 1 root root      4 Oct 27 14:40 rtc -> rtc0
crw------- 1 root root 254, 0 Oct 27 14:40 rtc0

[apvrille@rocky][/dev]$ ls -al std*
lrwxrwxrwx 1 root root 15 Oct 27 14:40 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root 15 Oct 27 14:40 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root 15 Oct 27 14:40 stdout -> /proc/self/fd/1

[apvrille@rocky][/dev]$ ls -al console*
crw------- 1 root root 5, 1 Oct 27 14:40 console
```

Access rights

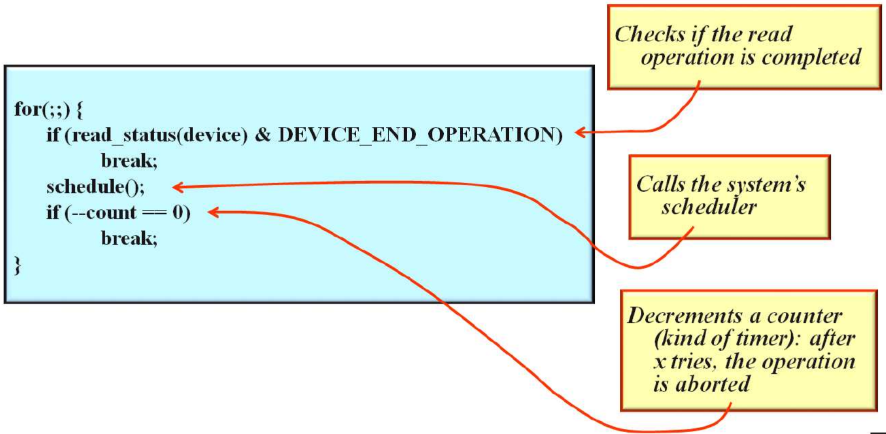
Device drivers

- VFS uses a set of common functions
  - *init()*, *open()*, *read()*, *write()*, *close()*, etc.



- A driver is registered as managing a group of devices
- Monitoring of I/O operations: Polling, Interrupts

Polling (Read Operation)



Interrupts

1. The device driver starts the I/O operation
2. It invokes the *sleep_on()* function (see on next slide)
   - Or *interruptible_sleep_on()*
   - Parameter: I/O wait_queue in which it wants to wait for
3. Then, it sleeps
4. When the interrupt occurs, the interrupt handler invokes *wake_up()* on all processes waiting in the corresponding device queue
5. The waked-up device driver can therefore check for the result of the operation

Interrupts: *sleep_on()* Function

```
static inline void __sleep_on(struct wait_queue
    **p, int state) {
    unsigned long flags;
    struct wait_queue wait = { current, NULL };
    if (!p) return;
    if (current == task[0])
        panic("task[0] trying to sleep");
    current->state = state;
    add_wait_queue(p, &wait);
    save_flags(flags);
    sti();
    schedule();
    remove_wait_queue(p, &wait);
    restore_flags(flags);
}
```

- *Allocates a new* **struct** *to add to the wait queue the current process* (→ `struct wait_queue wait = { current, NULL };`)
- *Error if no wait queue or current = task[0] (first OS process)* (→ `if (!p) return;` … `panic("task[0] trying to sleep");`)
- *Turns off interrupts (if applicable), adds the wait_queue to the queue and turns on interrupts (if on before the call)* (→ `add_wait_queue(p, &wait);`)
- *Allows interrupts to occur* (→ `sti();`)
- *Call to the scheduler* (→ `schedule();`)
- *Removes process from the queue* (→ `remove_wait_queue(p, &wait);`)
- *Restores original conditions previously saved* (→ `save_flags(flags);`, `restore_flags(flags);`)