



Scuola Superiore Sant'Anna



I/O subsystem

Giuseppe Lipari



Input – Output and Device – Drivers



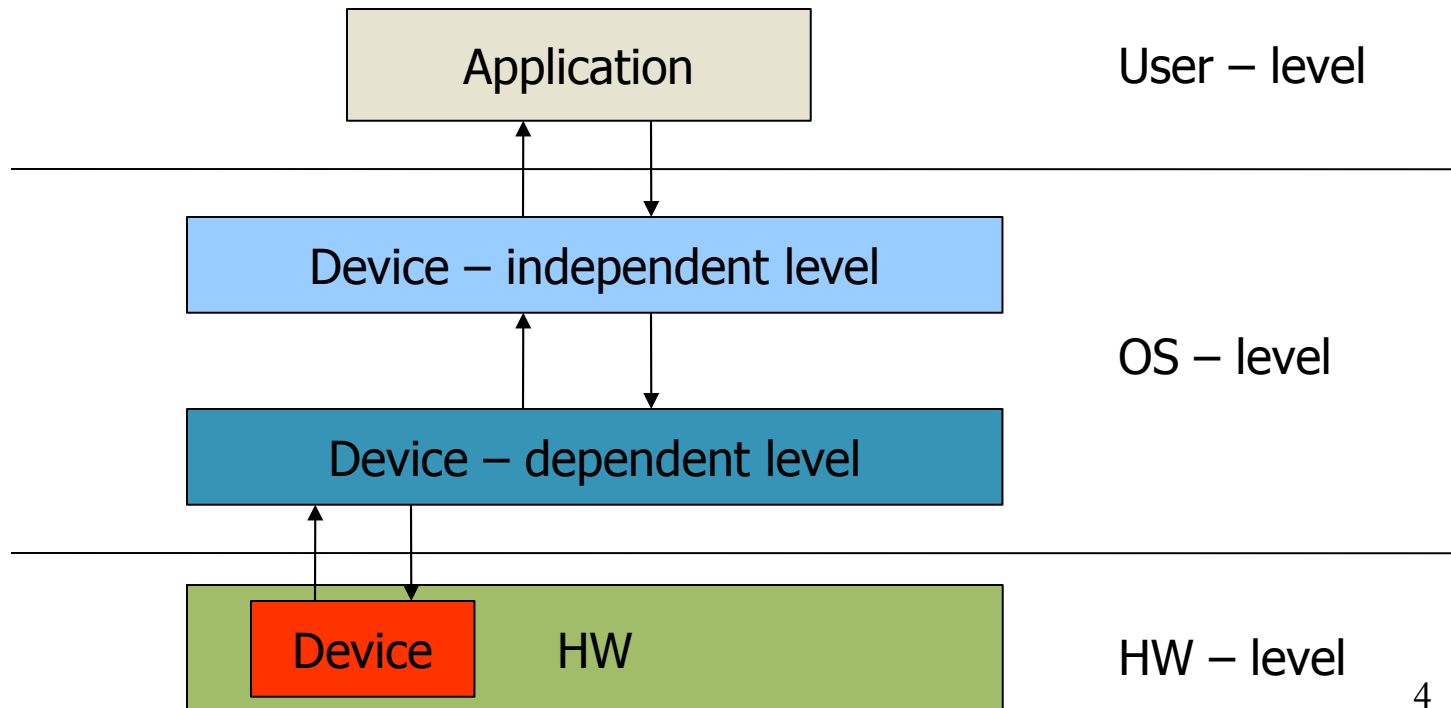
Objectives of the I/O subsystem

- To hide the complexity
- From the variability of the devices
 - Provide a consistent API
 - Open(), Read(), Write(), Close(), Lseek()
 - Provide a naming system
 - For example, the file system
 - Optimize access
 - For example, through buffering and caching



Logical organisation

- We can distinguish two logical levels
 - Device – dependent level
 - Strictly depends on the particular device
 - Device – independent level
 - Provides an uniform interface to the user





Device - independent level

- In this layer, the following functionalities are provided
 - Buffering
 - It is needed to transfer data to user space
 - Used to optimize the reading/writing on a device
 - Error handling
 - Many different kind of errors!
 - Errors can be reported to the user in a meaningful way
 - Allocation and spooling
 - How to allocate a device to a process requesting the service
 - for example, printing, visualizing, input from the mouse/keyboard, etc.
 - Naming
 - Each device is assigned a logical “identifier”
 - Data on the device can be accessed through the identifier



Device – dependent level

- In this level
 - The device driver interacts directly with the HW
 - It interacts also with the device-independent level
 - It provides specific access modes to the device
 - It provides ways to communicate with the upper layer
 - The interface between the d-d layer and the d-i level is specific for the device
 - Usually, it is a set of functions very similar to the user/level functions `open()`, `read()`, `write()`, `close()`, etc.



Device drivers

- A device drivers is
 - a software module of the OS that contains code dedicated to the interaction with a particular device
 - Disk device driver
 - Keyboard device driver, etc.
- Why device drivers
 - Abstraction
 - Portability
 - Modularity of the OS code



Variability

- The device driver subsystem
 - It is one of the most complex parts of the OS
 - The complexity is due to the huge variability of devices
- Classification of I/O devices
 - Human readable
 - mouse, keyboard, video, printer, etc.
 - Machine readable
 - Disk, controllers, sensors, actuators
 - Communication
 - digital networks, modems



Characteristics of a device

- Data rate
 - There can be huge differences in the speed of data transfer
- Applications
 - the way in which a device is used is important to the device driver implementation
 - for example, disk for normal user interaction, disk for web servers, disk for video servers, disk for virtual memory management, etc.
- Complexity of control
 - From very simple (e.g. the printer) to very complex (e.g. the disk)
- Unit of transfer
 - Character devices, block devices, packet devices
- Data representation
 - Parity, encoding, error corrections, etc.
- Error conditions
 - The way in which errors may occur, and the way they are reported may vary a lot



I/O device controller

- Let us start from the HW
- Every device has a controller that consists of
 - a dedicated small processor for I/O control
 - a set of registers mapped in memory
- Commands
 - To send commands to a I/O device, write on the registers
 - To read the status of the I/O operation, read from the registers
- The CPU is NOT synchronised with the I/O processors!
 - Data can arrive from the device in a asynchronous way



Controller Registers

- A device controller should possess at least the following registers
- Control register CR
 - It is used to send commands to the controller
 - For example, start a output data transfer, start a input data transfer, etc
 - We assume that a **start** bit is defined
- Status Register SR
 - It is used to check the status of the current operation and of the device
 - For example, if the transfer has been completed, if an error occurred, etc.
 - We assume that a **flag** bit is defined to signal to completion of any operation
- It can have also a data register/buffer and a Configuration register



External process

- We can model the I/O controller as a process
 - it is not a real software process! it is only a model for its behaviour

```
external process {  
    while (true) {  
        while (start==0);  
        <execute command>  
        <update status register>  
        flag = 1;  
    }  
}
```



Character – based devices

- First, we will discuss character based devices
 - example, the keyboard, the serial line, etc
- To transfer n bytes
 - n different I/O operations
- This is the simplest case



Polling

- In polling,
 - the operating system directly checks the flag bits
 - active waiting!

```
...  
for (i=0; i<n; i++) {  
    <prepare transfer>  
    start = 1;  
    while (flag==0);  
    <check status>  
}  
...
```



Polling

- If the device is very fast, then polling may be the best approach
 - In fact, an interrupt based mechanism involves at least a pair of process switches
 - If the duration of the operation is less than two process switches, then polling is the most optimized solution
- Almost all devices are much slower than the processor
 - usually, a data transfer takes much more than two process switches
 - For these devices, an interrupt-based mechanism is the best approach



Interrupt based I/O

- In the controller
 - The controller interrupt mechanism must be configured
 - We must specify which interrupt line is dedicated to the device
 - Different devices send data on different interrupts lines
 - All the lines enter in the interrupt controller
 - The interrupt controller will raise the interrupt pin of the processor
- In the processor
 - The processor interrupt mechanism must be configured
 - The processor has an “interrupt vector”
 - For every interrupt line, the address of a function (interrupt handler) is specified
 - When the interrupt is raised, the corresponding interrupt handler is invoked



Interrupt handling mechanism

- The device driver is written as a process
 - It is a very special kind of process
 - It is not a user process, but a kernel process, because it runs in the same address space of the operating system
 - In the following, it will be called Internal Process, to distinguish from the external process
- We define a semaphore *available*
 - the internal process is blocked on *available* waiting for the completion of the operation
 - when the interrupt handler executes, it performs the signal on *available*



Interrupt handler

Interrupt handler

```
Semaphore available(0);  
  
void handler()  
{  
    available.signal();  
}
```

Internal process

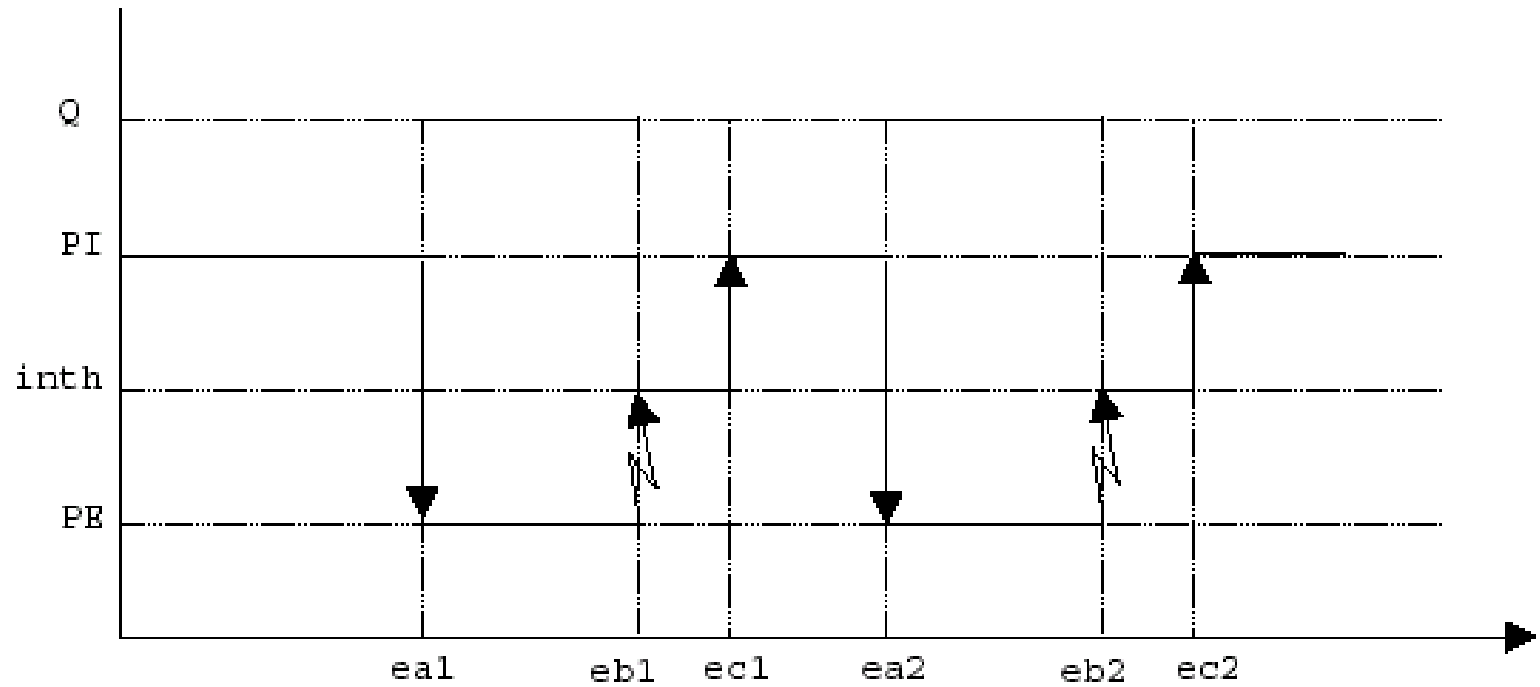
```
...  
for (i=0; i<n; i++) {  
    <prepare transfer>  
    start = 1;  
    available.wait();  
    <check status>  
}  
...
```

External process

```
external process {  
    while (true) {  
        while (start==0);  
        <execute command>  
        <update status register>  
        flag = 1;  
        <raise interrupt>  
    }  
}
```



Temporal evolution





Some intelligence in the handler

- The previous structure has one drawback
 - For each data byte, the internal process is executed
 - This requires two process switches for every byte
- Idea
 - The transfer of subsequent bytes can be done by the interrupt handler itself!
- Device descriptor
 - It is a data structure shared by the internal process and the interrupt handler
 - It contains a buffer of the data to be transferred



Device descriptor

- It contains
 - A semaphore `end_of_transfer`, on which the internal process will be blocked
 - The address of the device controller registers
 - Status and control registers
 - Number of bytes to be transferred
 - Counter for number of transferred bytes
 - Buffer address
 - Where data is put or where data is taken
- All device descriptors are kept in a array
 - The array is indexed by the device identifier
 - The device identifier is an integer that identifies the device



Device descriptor

```
struct IO_descr {  
    Semaphore end_of_transfer;  
    int counter;  
    int nbytes;  
    int *CR;  
    int *SR;  
    void *buffer;  
    int status;  
    // etc.  
};  
  
struct IO_descr dev_descr[NUM_DEVICES];
```



Read operation

```
int read(int ddes, char *pbuf, int nbytes) {  
    dev_descr[ddes].nbytes = nbytes;  
    dev_descr[ddes].counter = 0;  
    dev_descr[ddes].buffer = pbuf;  
    <set start = 1>  
    dev_descr[ddes].end_of_transfer.wait();  
    <check status>  
    if (<error>)  
        < return error_code>  
    return dev_descr[ddes].counter;  
}
```

- This function is invoked by the internal process

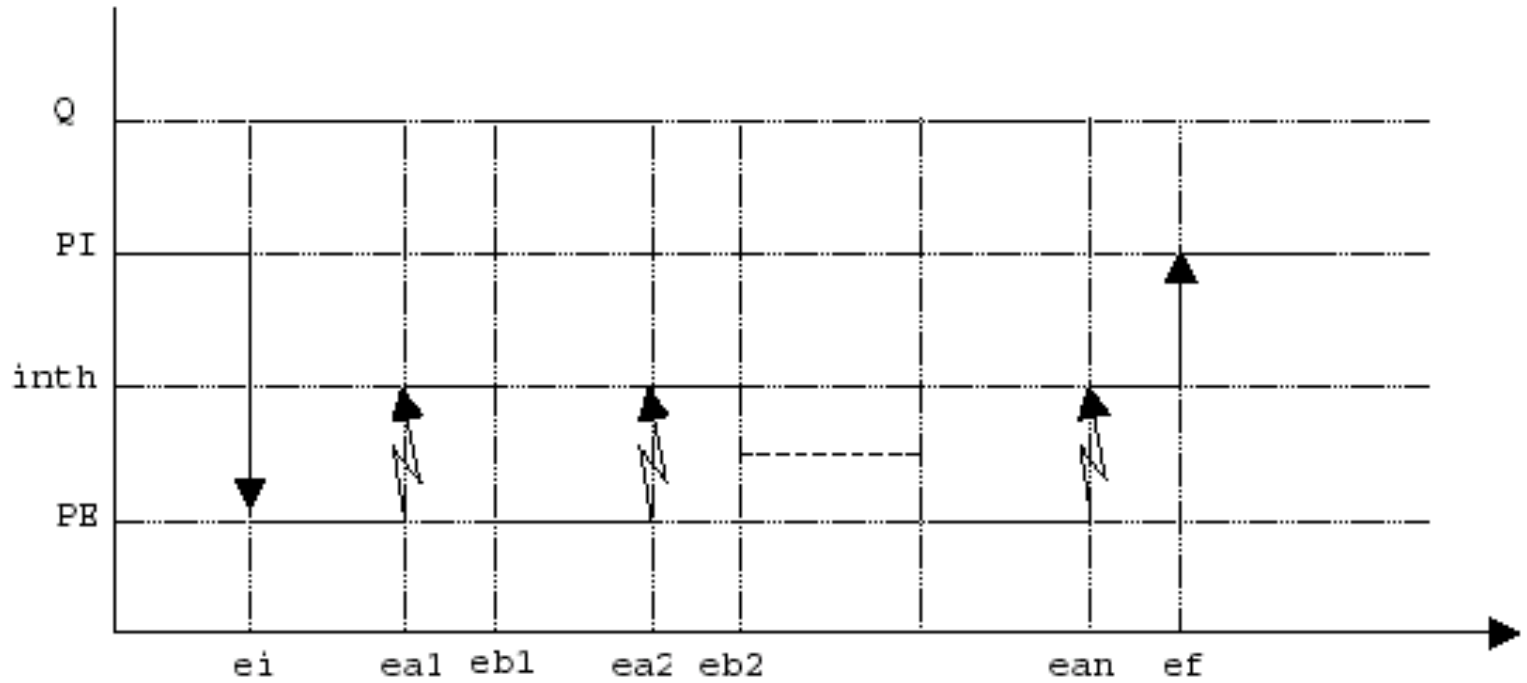


Interrupt handler

```
void handler() {
    char b;
    <check for error>
    if (<no error>) {
        *dev_descr[ddev].buffer = <get data from device>;
        dev_descr[ddev].buffer++;
        dev_descr[ddev].counter++;
        if (dev_descr[ddev].counter < dev_descr[ddev].nbytes)
            <set start bit = 1>
        else
            dev_descr[ddev].end_of_transfer.signal();
    }
    else {
        if (<non critical error>)
            <retry : set start bit = 1>
        else
            dev_descr[ddev].status = error_code
    }
}
```



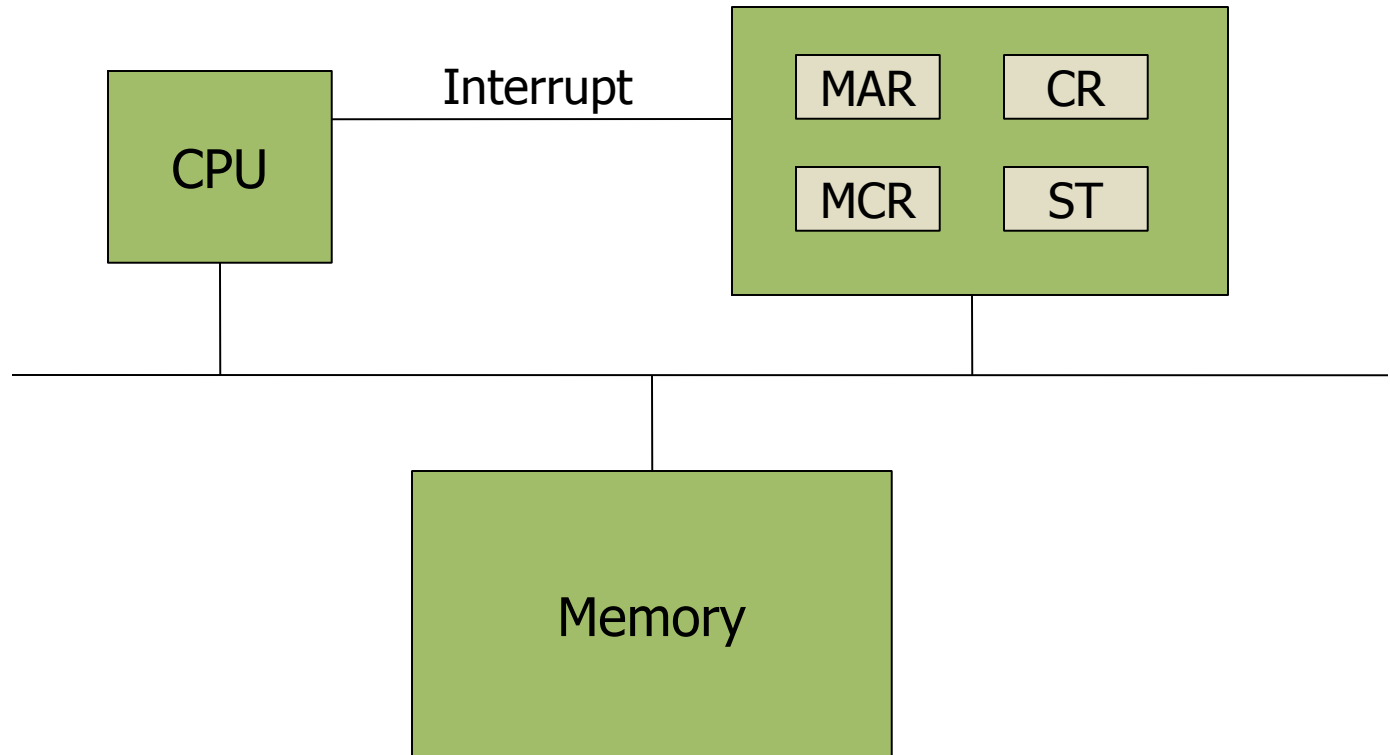

Timing





Direct memory access

- Many I/O controllers support Direct Memory Access
 - Every I/O controller has a DMA interface
 - A memory address register (MAR)
 - A counter
 - It can automatically transfer a block of bytes to/from memory
 - At the end of the transfer, an interrupt is raised
- With DMA
 - The interrupt handler becomes very simple!





Read operation

```
int read(int ddes, char *pbuf, int nbytes) {  
    dev_descr[ddes].nbytes = nbytes;  
    dev_descr[ddes].counter = 0;  
    dev_descr[ddev].buffer = pbuf;  
    <program DMA>  
    <MAR = pbuf>  
    <MCR = nbytes>  
    dev_descr[ddev].end_of_transfer.wait();  
    <check status>  
    if (<error>)  
        < return error_code>  
    return dev_descr[ddev].counter;  
}
```



Interrupt handler

```
void handler() {  
    <check for error>  
    if (<no error>) {  
        dev_descr[ddev].counter=<MCR>;  
        dev_descr[ddev].end_of_transfer.signal();  
    }  
    else {  
        if (<non critical error>)  
            <retry : set start bit = 1>  
        else  
            dev_descr[ddev].status = error_code  
    }  
}
```



DMA

- By using DMA
 - The overhead is further reduced
 - The interrupt is called only once at the end of the operation
 - The controller is more complex
- Overhead
 - The DMA transfers data to memory on the bus
 - The bus and the memory are shared between the processor and the I/O device
 - Conflicts on the bus are resolved by the bus controller
 - However, the overhead of the DMA is not 0!
 - Every read/write operation from/to memory can be delayed because the I/O device could access the memory at the same time!



Buffering

- Goals
- Transfer data to user space
 - Mechanism
 - Data is first loaded into an internal buffer of the OS
 - Then, it is transferred into the user buffer
 - Why
 - For timing reasons: while reading one buffer, I can try to load another buffer
 - For Virtual memory reasons: the memory of the process (which could be subject to swapping and paging) is not directly involved with the I/O hardware operation
- Optimized access
 - Instead of loading data when we need them, load them in advance



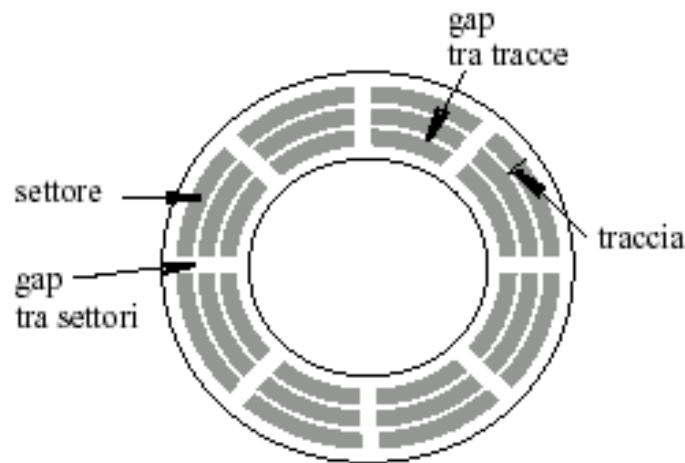
Buffering

- Single buffer
 - One single buffer for each device
 - While the operation is performed, the process cannot access the buffer
- Double buffering
 - Two buffers for each device
 - While the operation is performed on one buffer, the other can be used by the process
 - At the end of the reading, we can swap the buffers
- Circular buffer
 - An extension of the double buffering
 - It is used only in particular cases

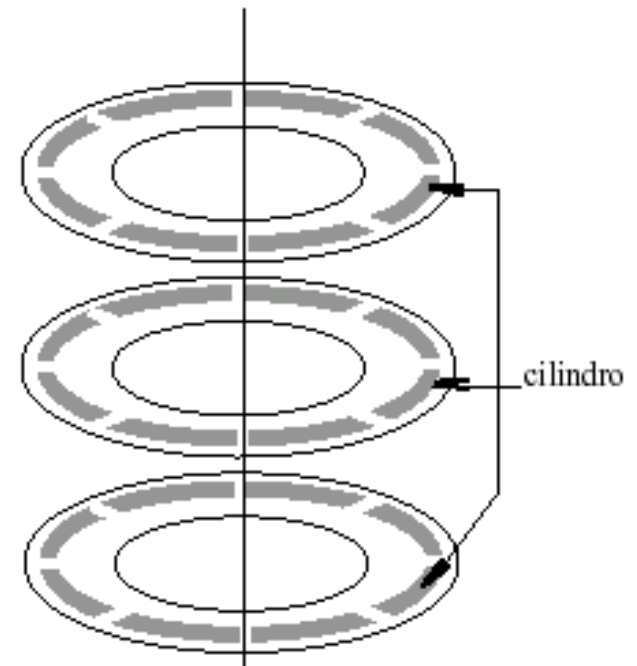


Block devices

- As an example of block devices, we will consider the disk



a) disco singolo



b) disk pack



Disk performance

- Average transfer time $TF = TA + TT$
 - The average access time TA is the time to move the disk arm to the desired track and sector
 - The average byte transfer time TT is the time needed to actually transfer the data
- $TA = ST + RL$
 - the seek time ST is needed to move the arm on the desired track
 - the rotational latency RL is the time waiting for the disk to rotate until the desired sector
- The greatest and most important time is the ST
 - The arm moves more slowly than the disk rotate
- GOAL:
 - move the arm the less that is possible!



Some data

- Western Digital WDE18300
- Cylinders: 13614
- Tracks per cylinder: 8
- Sectors per track: 320
- Bytes per sector: 512
- Capacity: 18.3 Gb
- Times
 - Minimum ST 0.6 msec
 - Average ST 5.2 msec
 - Rotational Latency RL 6 msec, (average 3 msec)
 - Average transfer time TT 19 microsec.



Example

- Let's compute the time to read a 320kb file
 - Suppose that a process wants to read 640 sectors (320 kb)
 - Assume they are stored in two consecutive tracks
 - First Track:
 - $TF = ST + RL + TT = 5.2 + 3 + 6 = 14.2$
 - Second track
 - $TF = ST + RL + TT = 0.6 + 3 + 6 = 9.6$
 - Total : 23.8 msec
 - Assume they are stored in random order
 - Every sector
 - $TF = ST + RL + TT = 5.2 + 3 + 0.019 = 8.219$
 - Total: $8.219 * 640 = 5.26$ seconds!!

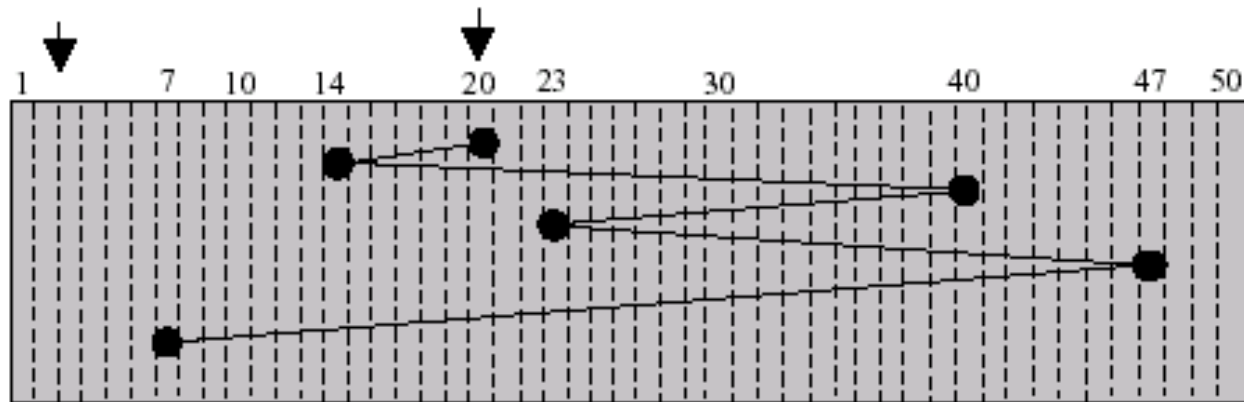


Scheduling

- There is a huge difference in the two times
 - It is much more convenient to store files consecutively
 - However, there is a problem
 - Several processes could request reading from the disk
 - If we serve the requests as they come, we could end up moving the arm up and down even in the case the files are stored consecutively!



FIFO

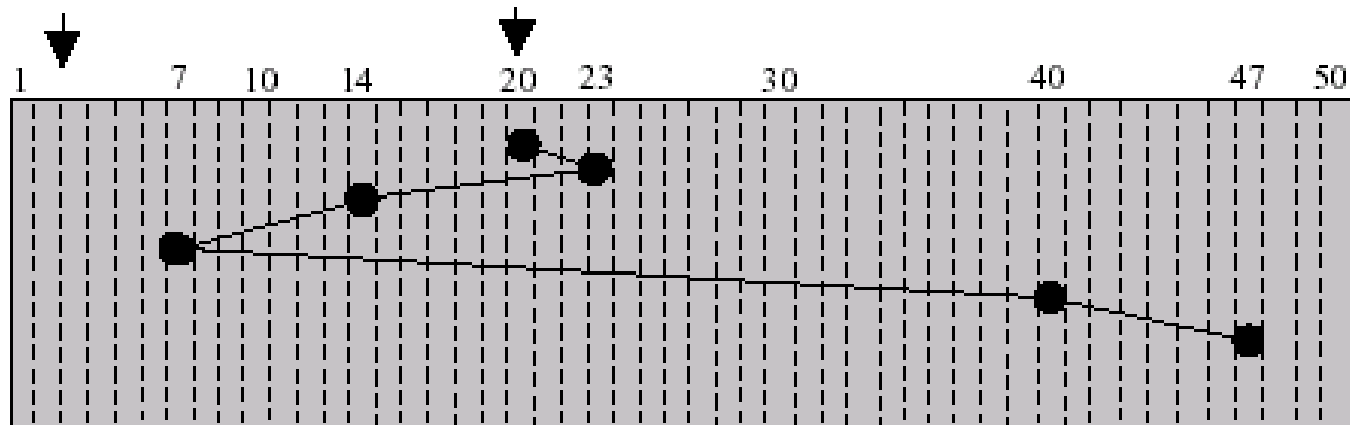


- Example:
 - Requests in the queue: 20, 14, 40, 23, 47, 7
 - With FIFO, we have a very bad performance! (the arm keeps moving randomly)



SSTF

- Shortest Seek Time first
 - Server the request that is closer to the actual position of the arm
 - Requests in the queue: 20, 14, 40, 23, 47, 7
 - Order of service: 20, 23, 14, 7, 40, 47





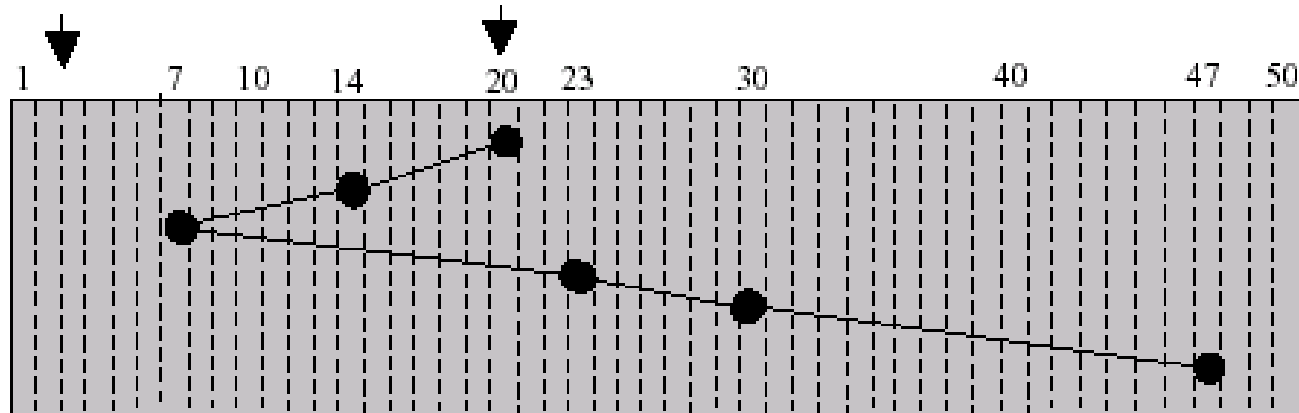
SSTF: problems

- The algorithm is not optimal
 - Minimizing the seek time for the next reading does not imply minimizing the total seek time
 - The algorithm could cause starvation!
 - If the disk is in overload (too many requests), a request of a cylinder that is far away never gets served!



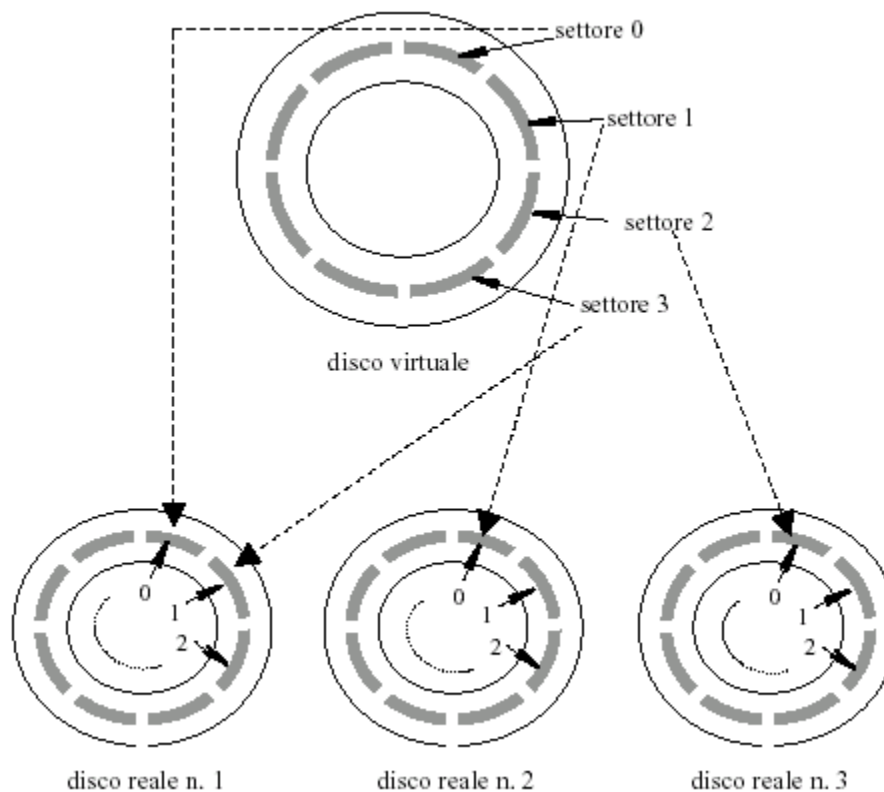
SCAN

- It is called the “elevator algorithm”
 - Keep increasing the cylinder search in one direction, then it changes direction
 - Requests in the queue: 20, 14, 40, 23, 47, 7
 - Order of service: 20, 14, 7, 23, 40, 47



RAID

- It is used for fault tolerant disks
 - a virtual disk is implemented as a set of “real disks”





6 levels

- RAID 0
 - No redundancy: only for performance improvement
 - data can be accessed in parallel!
- RAID 1:
 - Simple duplication, very costly
- RAID 2:
 - Hamming code: quite costly
- RAID 3:
 - One disk for parity (bit level)
- RAID 4:
 - Block level parity
- RAID 5:
 - Block level distributed parity
- RAID 6:
 - Dual Redundancy