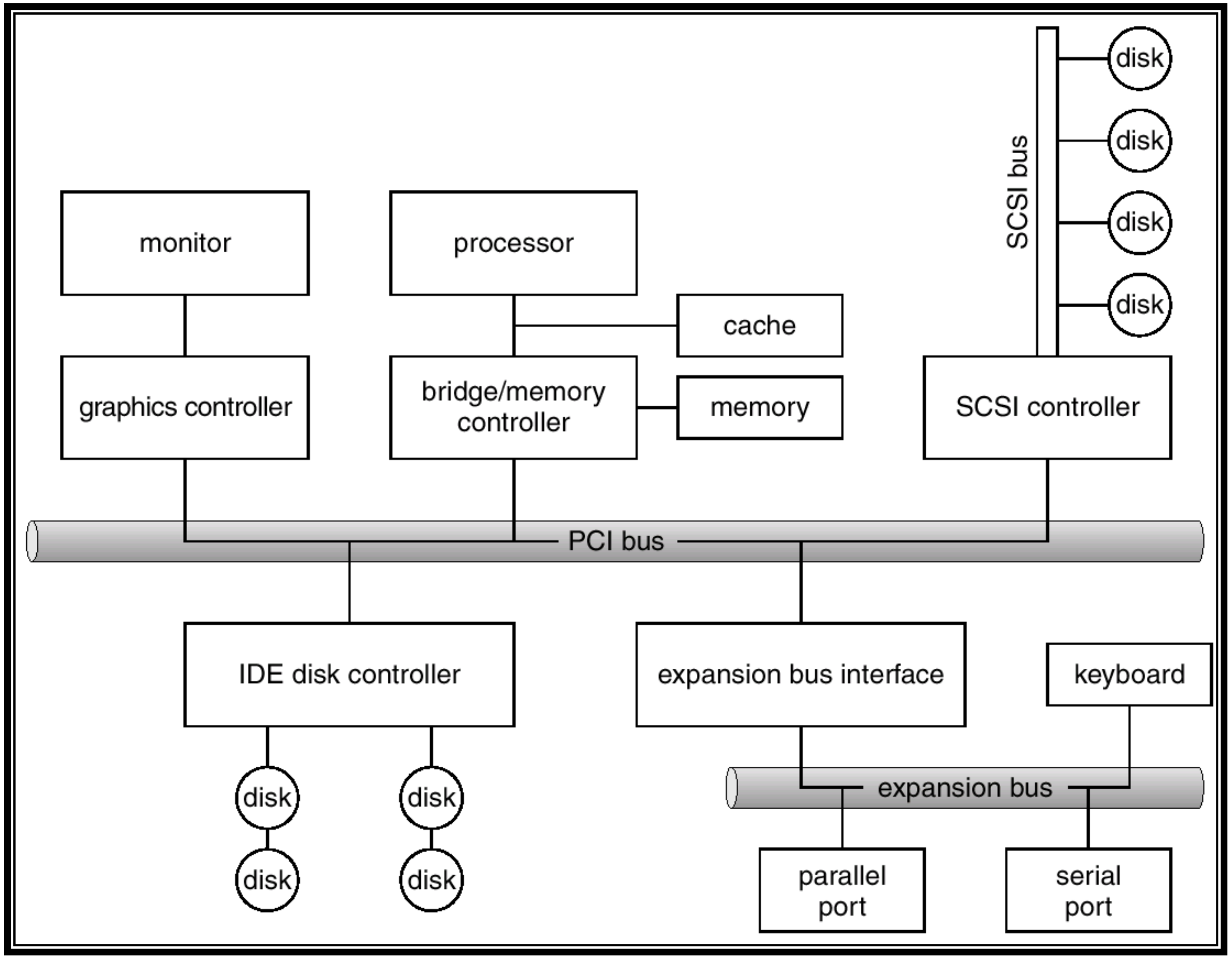
Operating Systems

Lesson 5: Input / Output

Last rev.: December 2016

INPUT/OUTPUT

Input/output



Silberschatz, Galvin and Gagne 2002 , section 13.3
(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html)

Input/output

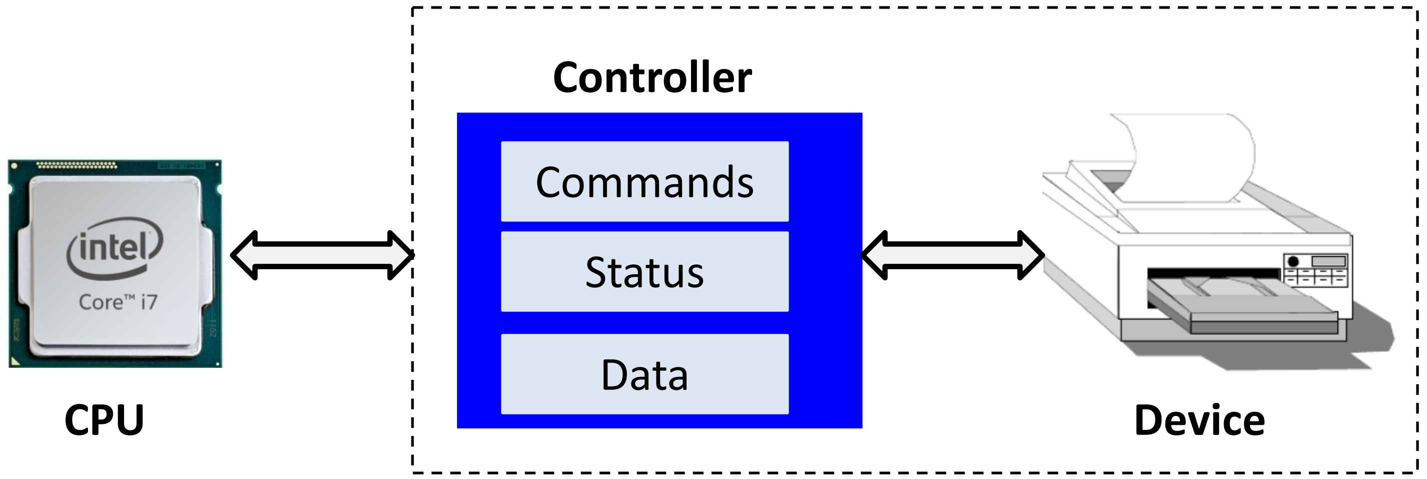
- i/o devices allow CPU to communicate with the external world: keyboards, displays, printers, disks, …
- The communication between the CPU and an external element is similar to the communication with main memory: data are both written and read
- However, the behavior is different: data are not always available (i.e. a keyboard), and the device may not be ready to receive them (i.e. a printer)
- Since the behavior differs, the access methods are also different from those used to access memory

INPUT/OUTPUT

Structure of an i/o system

- In theory, i/o devices would communicate with the CPU through the system buses
- Since the i/o devices are highly heterogeneous, it would be very costly if the CPU had to manage them directly. (the O.S. would have to know “all the details”, CPU much faster than i/o devices,…)
- The devices are connected to a piece of hardware called *device controller* (sometimes called device adapter).
- The device controller admits/receives *abstract CPU commands* and is the responsible of transmitting them to the device (i.e. write block 2534 to disk)
- The CPU is freed of *very low-lever tasks* (i.e. write data in side X of platter Y, in cylinder Z, sector T)

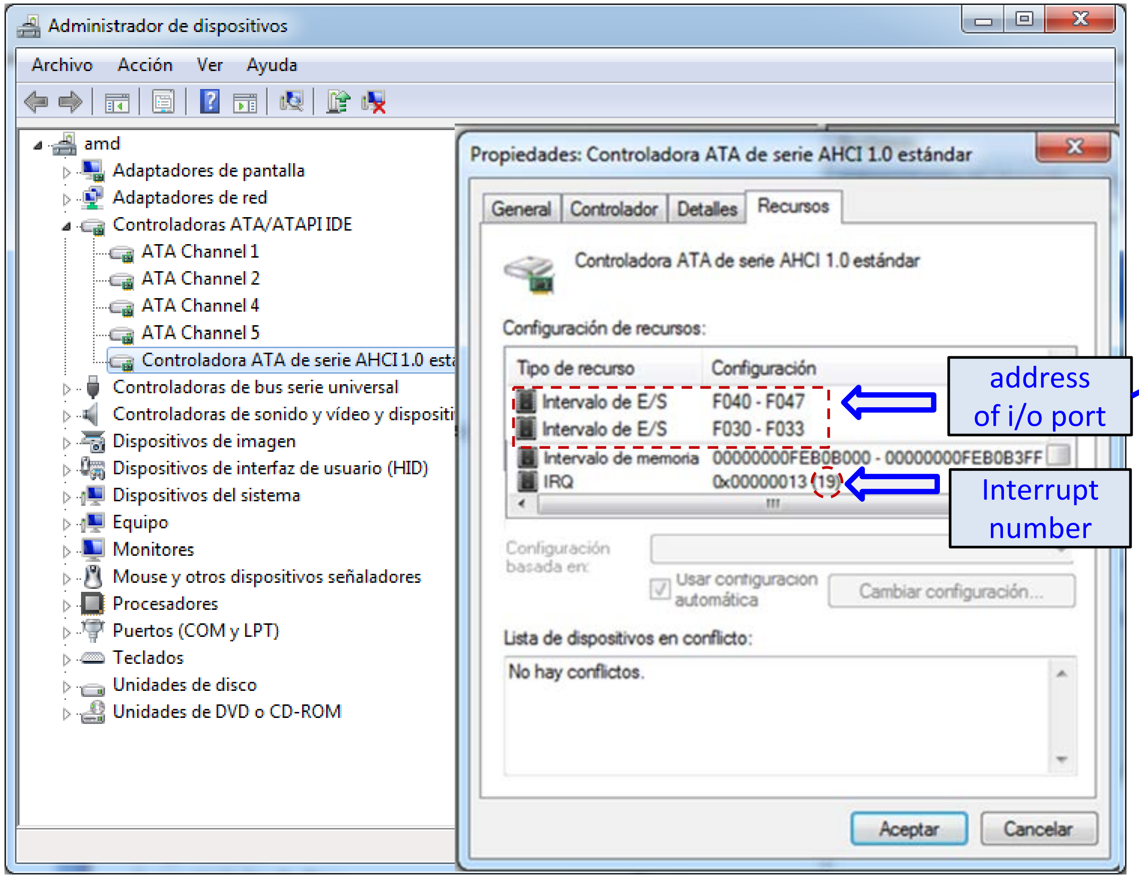
Structure of an i/o system



Structure of an i/o system

- The device controller acts as an interface between the CPU and the i/o device
- Each device controller can handle either one or several devices of the same type (i.e. IDE controller → two HDDs; or one HDD and one DVD unit)
- The controllers communicate with the CPU through some registers or ports. They typically include:
  - A control register: To send commands to the device
  - A status register: To obtain information about the state of the i/o device or the controller, data availability,…
    - i.e. the previous command has already been completed
    - i.e. bit to indicate that there were errors during the last i/o operation
  - Data registers: They can be input registers (data-in register), output registers (data-out register), or bi-directional registers.
    - They are typically registers that can hold from 1 to 8 bytes.
    - Some controllers contain FIFO chips that allow us to store small amounts of burst data until the host (CPU) or the device is not ready to receive them.

Structure of an i/o system



Administrador de dispositivos

Archivo Acción Ver Ayuda

- amd
  - Adaptadores de pantalla
  - Adaptadores de red
  - Controladoras ATA/ATAPI IDE
    - ATA Channel 1
    - ATA Channel 2
    - ATA Channel 4
    - ATA Channel 5
    - Controladora ATA de serie AHCI 1.0 está
  - Controladoras de bus serie universal
  - Controladoras de sonido y vídeo y dispositi
  - Dispositivos de imagen
  - Dispositivos de interfaz de usuario (HID)
  - Dispositivos del sistema
  - Equipo
  - Monitores
  - Mouse y otros dispositivos señaladores
  - Procesadores
  - Puertos (COM y LPT)
  - Teclados
  - Unidades de disco
  - Unidades de DVD o CD-ROM

Propiedades: Controladora ATA de serie AHCI 1.0 estándar

General | Controlador | Detalles | Recursos

Controladora ATA de serie AHCI 1.0 estándar

Configuración de recursos:

Tipo de recurso	Configuración
Intervalo de E/S	F040 - F047
Intervalo de E/S	F030 - F033
Intervalo de memoria	00000000FEB0B000 - 00000000FEB0B3FF
IRQ	0x00000013 (19)

Configuración basada en:

Usar configuración automática

Cambiar configuración...

Lista de dispositivos en conflicto:

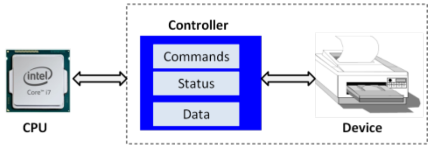
No hay conflictos.

Aceptar Cancelar

Windows 7 -- device manager -- screenshot

- We will see that the registers of the controller are located in these addresses.
- Accessing this addresses range allows the communication with the "ATA controller"

Structure of an i/o system



- The device controller is in charge of
  - Coordinating the data flow between either the CPU or the memory, and the peripheral device.

    Polling i/o,
    Interrupts i/o
    DMA
  - Communication with the CPU (*CPU ←→ controller*):
    - decoding the commands provided by the CPU,
    - interchange of i/o data with the CPU,
    - Recognizing the device address: It must realize that the data coming through the buses belong to (are sent to) this device (and not to to any other device).
  - Communication with the i/o device (*controller ←→ device*):
    - Sending commands
    - Data interchange
    - Receiving status information
  - Temporal storage of data (buffer) due to the fact that the speeds (of the CPU and of the device) are very different.
    - i.e. The CPU is much faster → the controller keeps/receives the data that CPU is sending and that must be written in the device… then the controller sends such data to the slow i/o device "step-by-step".
  - Error detection

INPUT/OUTPUT

Structure of the i/o software

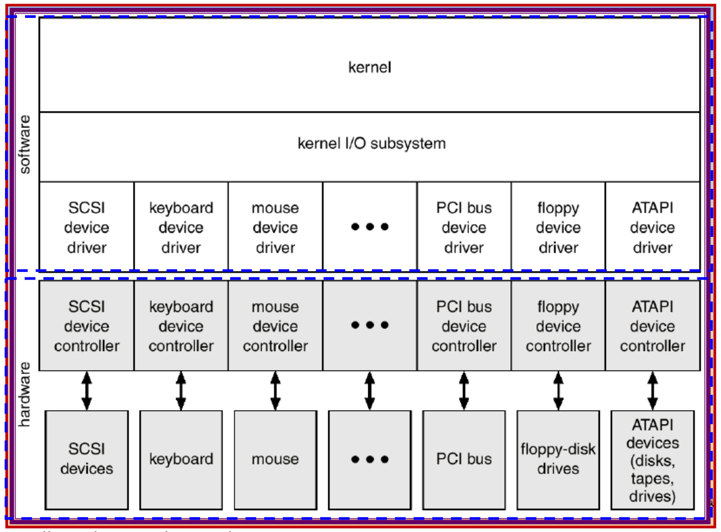
- How can a program write to a HDD if it does not know the type of disk that is being dealt with?
- How can we add a new device without disturbing the operation of a computer?

ABSTRACTION + ENCAPSULATION + LAYERS

- Abstraction
  - Eliminates the differences between devices by identifying a few generic classes/types.
  - For each class, an interface (with some standard functions) is defined.
- Encapsulation
  - The actual differences are encapsulated within the *device drivers:*
    - *Device drivers internally adapt to the particularities of the device, but*
    - *Device drivers export one of the standard interfaces (their functions)*
- *Layers*
  - …

Structure of the i/o software

- *Layers*
  - *The i/o software is structured into layers.*
  - *Device-driver layer hides differences among i/o controllers from kernel*



Silberschatz, Galvin and Gagne 2002
(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html)

- Calls to the i/o subsystem can see the behavior of the different devices within a few generic classes
  - The applications do not note the differences when dealing with one hardware or another.
- The i/o subsystem (in the O.S.) is over the *device driver layer*, that hides the differences between de different device controllers
- The fact that we have an i/o subsystem that is independent from the hardware
  - Helps O.S. developers (they do not have to deal with “infinite” devices), and
  - Helps hardware manufacturers:
    - They create a hardware that is compatible with a given existing interface (host/controller)
    - Or they write/create the *device driver* that establishes the interface with the new device and the O.S.’s in which it must operate

Structure of the i/o software

- The i/o software is structured into layers

USER MODE:

- **User level software**: It provides the interface with the user. At this level we find tasks such as data formatting.

KERNEL MODE:
-Block/char
-Buffers/cache
-Dev management:
+ naming,
+ protection,
+ access ctrl,
+ i/o scheduling

- **Device independent software**: This layer is in charge al all the tasks related to space allocation, privilege control, cache usage (if it exists)...
- **Device driver**: This is the software layer that communicates with the device controller, and it is the unique piece of software that actually does it. Therefore, in order to allow a device to be used by an O.S. only the corresponding *device driver* is needed. All the knowledge about the specific details of the device are included in the *device driver*.
- **Interrupt handler**: It handles all the interrupts that come from the device.

Structure of the i/o software

- **Example**: Let's see the work done in the different layers during a call to:

  `fprintf(fich,"%1.6f",x)`

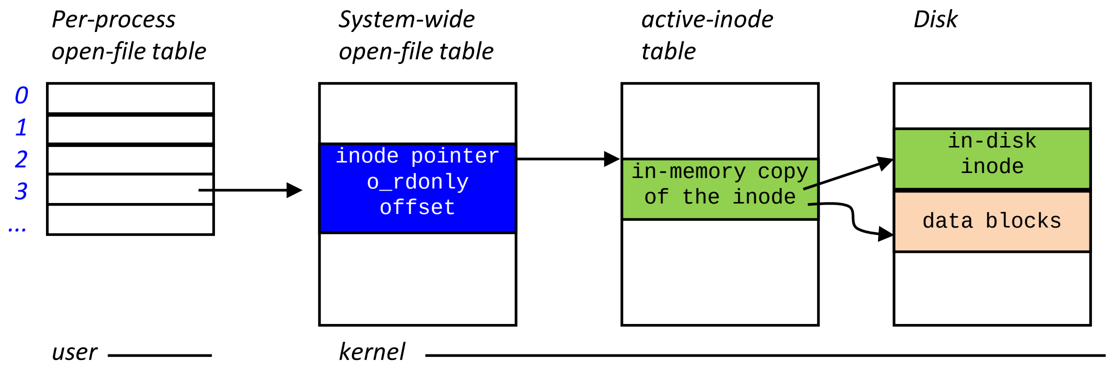
- User level software:
  - It generates the expected string from **x**: one digit, one point, 6 decimal digits, and '\0'
    - Therefore, it performs data formatting. For example, if *x=1.324242*, a 9-bytes string is created.
  - From the `FILE * fich` parameter, it obtains the corresponding file descriptor (fd) that will be needed in the subsequent *write* system call.
    - Therefore, it handles the process of preparing everything to the system call. (*write (fd, ptr_to_1.324242, 8)*)

strlen(ptr_to_1.324242)

Structure of the i/o software

- **Example**: Let's see the work done in the different layers during the call to:
  `fprintf(fich,"%1.6f",x)`

- Device independent software:
  - It receives a file descriptor (fd), a source address from where data must be transferred to the device (buffer) , and the number of bytes to transfer (N). *write (fd, buffer, N)*
  - *Locate the block* From the file descriptor (fd), it accesses the in-memory copy of the file inode and, using the offset in the open-files table, it determines in which disk block (and the offset within that block) data must be written.
  - *-Bring block to cache -Write data to block -Assign additional space if needed* It checks if that block is already cached (otherwise it is brought into memory), and transfers data from the user memory address to the cache. It marks that block from the cache as modified. If a new data block must be assigned (the file grows), it is done at this layer
  - *Finally, transfer block to disk (now or later)* Finally, (depending on the cache usage) a call to the device driver will be performed to write that block to disk.



Structure of the i/o software

- **Example**: Let's see the work done in the different layers during the call to:

  `fprintf(fich,"%1.6f",x)`

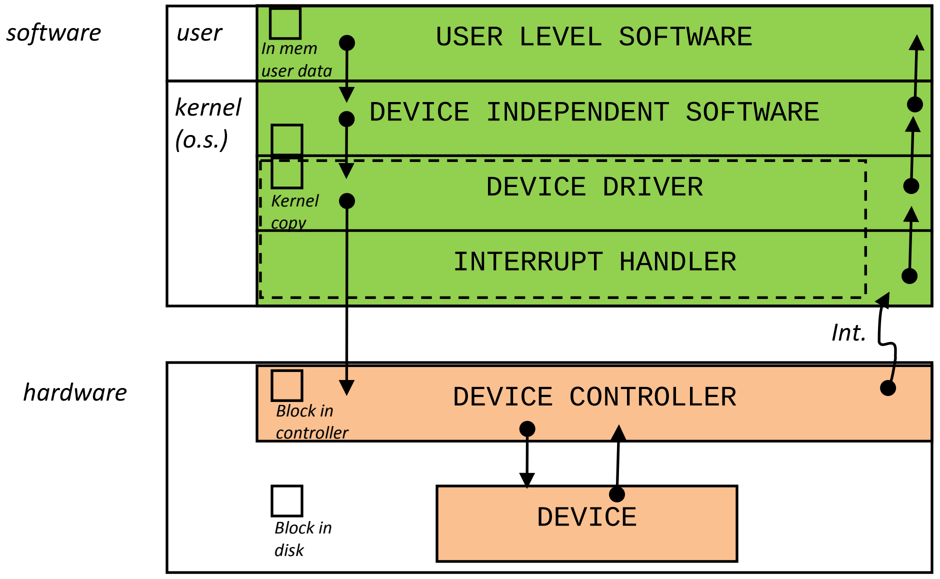
- Device driver:
  - It receives the command to write **a block** into a given device
  - It determines which physical device must be accessed
  - It determines the sector or sectors (as a logical block can be associated to K sectors, *i.e. k*512 bytes*) within the disk (as well as their coordinates regarding cylinder and side) corresponding to that block.
  - It sends the appropriate commands to the registers of the device controller (in order to perform the data transfer).

- Interrupt handler:
  - When the device completes the request, it generates an interrupt that is managed by the interrupt handler
  - It wakes up the process that requested the i/o (in case it was a synchronous i/o)
  - It frees the resources occupied by the i/o operation

Structure of the i/o software

- **Example**: Let's see the work done in the different layers during the call to:

  `fprintf(fich,"%1.6f",x)`



i/o system call: data formatting

naming, protection, blocks, cache, buffers, space allocation

Access to registers of the controller, check state

Processes interrupt when i/o finishes, wakes up driver..

Leads i/o operation

INPUT/OUTPUT

Types of i/o

- Depending on the communication method between the CPU and the devices we can distinguish:
  - Explicit i/o
  - Memory mapped i/o.

- Depending on the perception that the process has with respect to how the i/o is performed:
  - Synchronous i/o
  - Asynchronous i/o

Types of i/o

Communication method between the CPU and the devices

- Memory-mapped devices:
  - Devices appear within the addresses space. Addresses space is shared between both the memory and the devices
  - It does not typically becomes a problem because i/o space << than addresses space
  - To access the controller registers we use instructions of type ***MOV*** (as any other memory read/write operation)
  - Example: motorola 68000

MOV cpu-reg, address

- Separated i/o port space:
  - We have a range of explicit i/o addresses (this system is called explicit i/o)
  - Each i/o control register is assigned an i/o port number. The set of all the i/o ports makes up the i/o port space
  - Typically, there are special instructions to access this i/o space (i.e. **IN/OUT** in intel); a different way is that a control register activates this space (i.e. powerpc).
  - Example: MOV R1, 4 → reads mem word 4, and sets value into R1

    IN R1, 4 → reads i/o port 4, and sets value into R1
  - In this systems, there could also be memory-mapped devices

IN cpu-reg, port
OUT port, cpu-reg

Types of i/o

Perception that the process has with respect to how the i/o is performed

- Synchronous and Asynchronous i/o
  - Synchronous: The process "feels" that it has to wait until the i/o operation completes
  - Asynchronous: The process "feels" that it has NOT to wait ... and someone will notify to it that the i/o operation has completed.

- Since the CPU is much faster than the i/o devices, once the i/o operation has started, the O.S. gives the CPU to a different task and leaves the process that waits for the i/o into "waiting state"
  - The process feels that the i/o is synchronous but it is actually asynchronous.
  - When the i/o completes, the process will move from i/o queue to ready queue.

- Many O.S.'s permit also to perform asynchronous i/o's. The process initiates the i/o and continues its execution. The process will be notified by the O.S. when the i/o operation has completed.
  - For example, notified by: (a) an interrupt, (b) changing the value of a variable within its user space

INPUT/OUTPUT

Input/output

Structure of an i/o system

Structure of the i/o software

Types of input/output

Methods for input/output (polling, interrupts, and DMA**)**

Disk scheduling

Input/output in UNIX

Methods for i/o

i/o with polling (programmed i/o)

- It is the simplest way to perform an i/o operation
- To gain synchronization the CPU must ask (poll) to the device if it has a new piece of data to deliver (in), or if it is ready to receive a new piece of data (out).
- Busy waiting: CPU time is wasted each time we ask the device (the more frequent we ask the more time is spent).
- It is slow, and data could even be lost if the device is not queried frequently enough.

Methods for i/o

i/o with polling (programmed i/o)

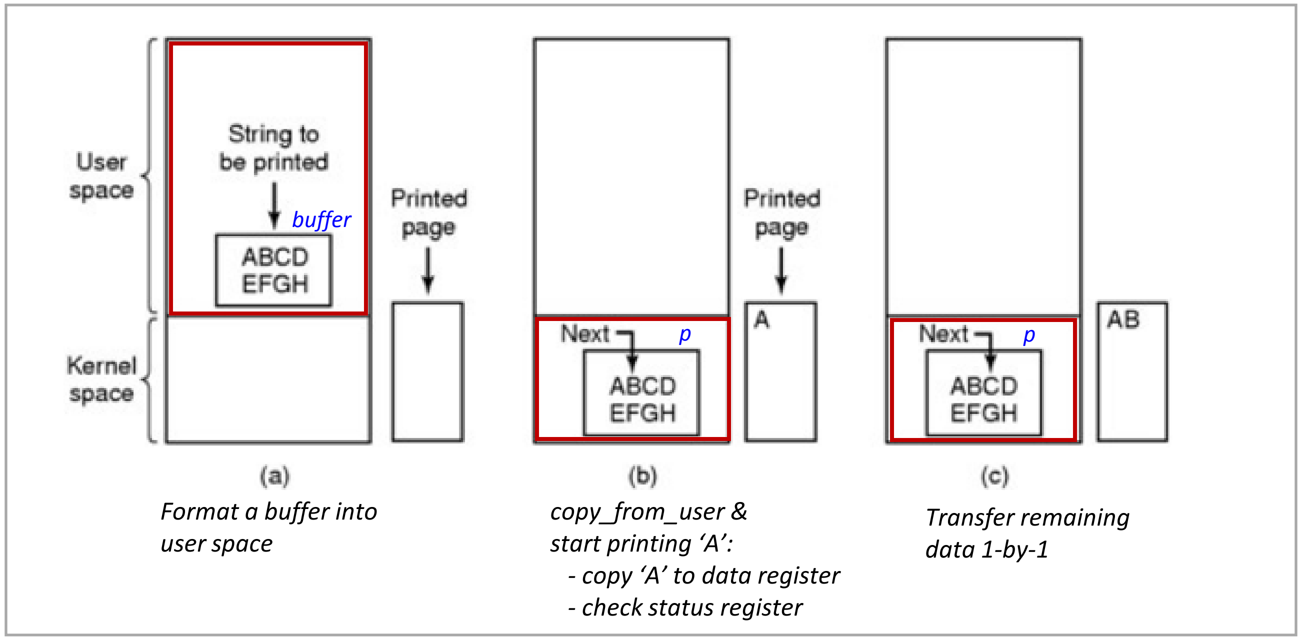
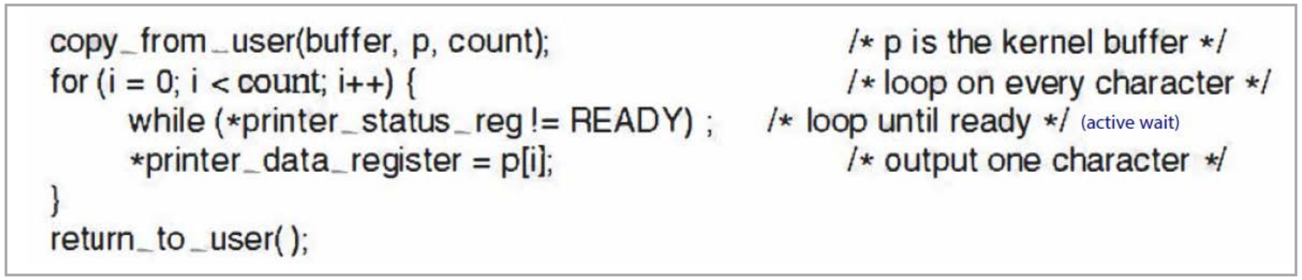


Figure 5.7: Steps for printing 'ABCDEFGH'



```
copy_from_user(buffer, p, count);                   /* p is the kernel buffer */
for (i = 0; i < count; i++) {                       /* loop on every character */
        while (*printer_status_reg != READY) ;    /* loop until ready */ (active wait)
        *printer_data_register = p[i];              /* output one character */
}
return_to_user( );
```

Figure 5.8: Writing a string to a printer using programmed i/o

(Figs. Source: Tanembaum, Modern Op. Systems, 3er ed, S. 5.2)

Methods for i/o

Interrupt-driven i/o

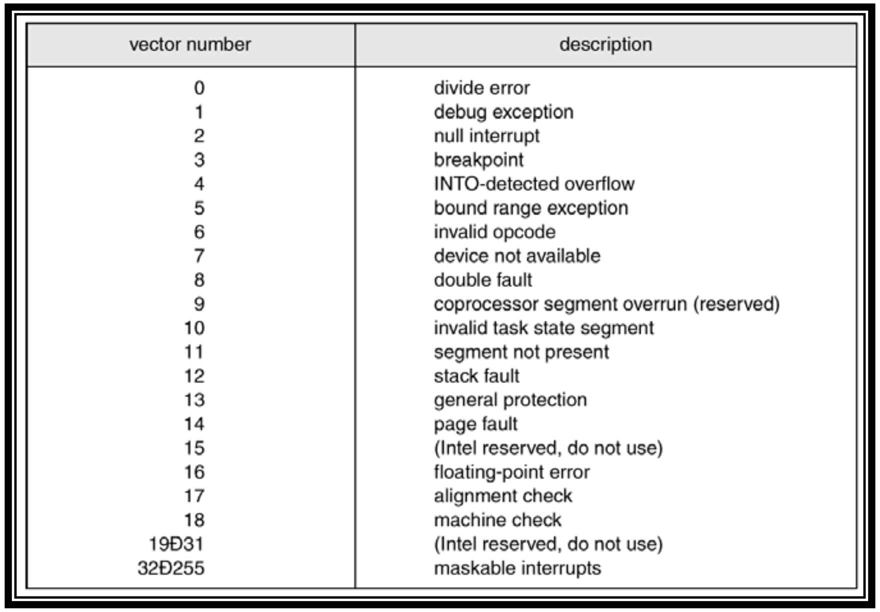
- The device asks for the CPU attention with an interrupt
- When the interrupt is detected, the O.S.
  - Stops the current process and saves its state
  - Transfers control to the interrupt service procedure (interrupt handler)
  - Executes the interrupt service procedure
  - After that, it continues the execution in the position where the interrupt stopped a prior process.

- Those interrupt service procedures are in memory in some addresses pointed to from *interrupt vectors*

- During the initialization process, the O.S. install those routines
  - It checks which devices are connected
  - Installs all the service procedures (handlers) in the interrupt vector
    - The interrupt vector dispatches interrupts to correct handler

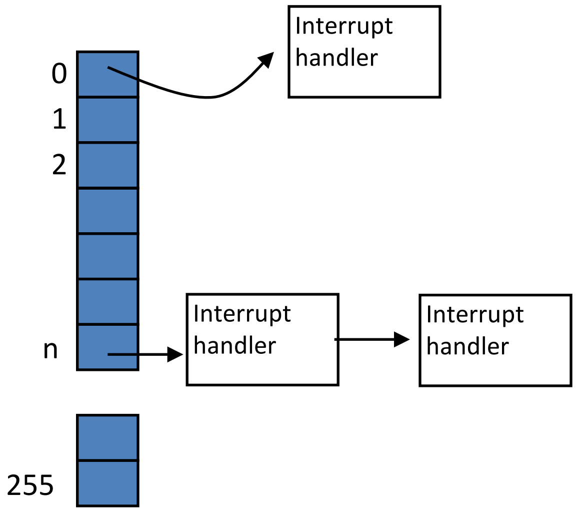
Methods for i/o

Interrupt-driven i/o



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19Ð31	(Intel reserved, do not use)
32Ð255	maskable interrupts

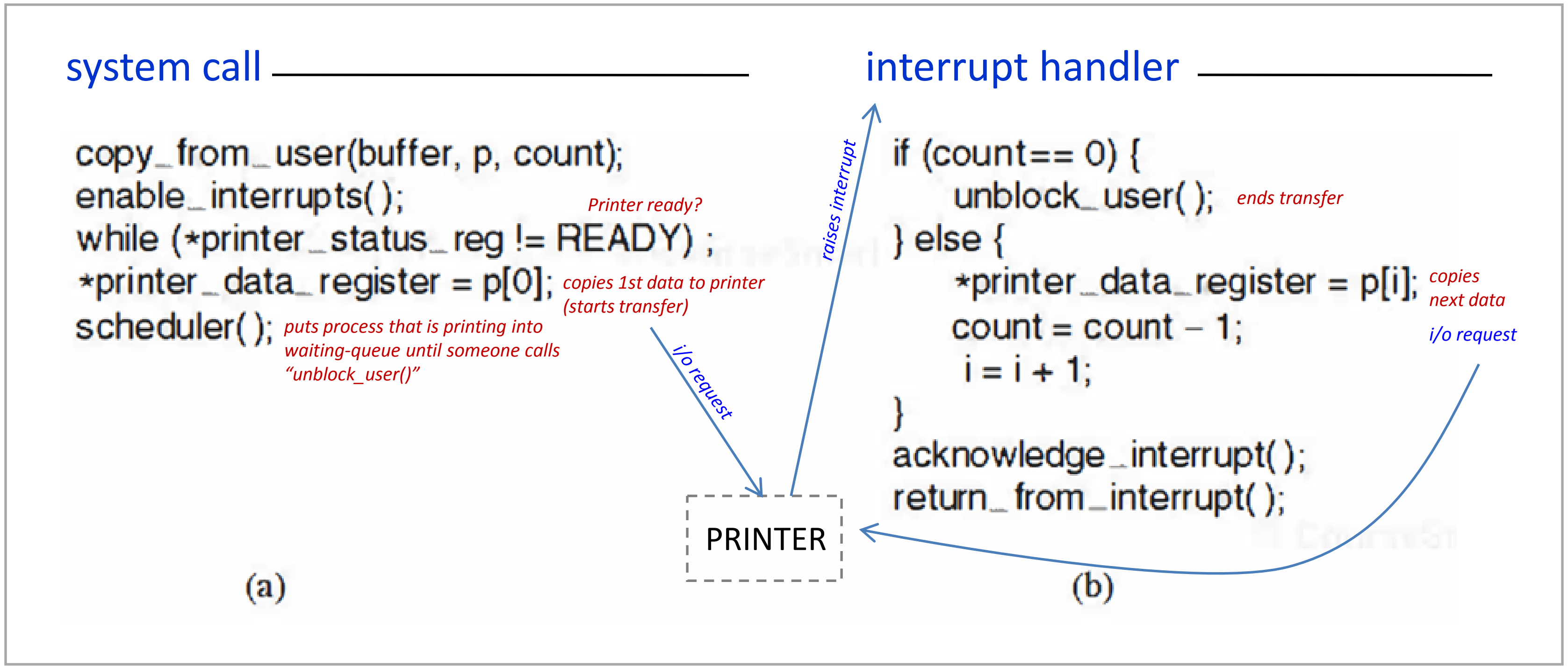
Interrupt vector in Intel Pentium



Each entry in the interrupt vector points to a interrupt handler (or chain of handlers) that must be used to give service to that interrupt

Methods for i/o

Interrupt-driven i/o



```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

```
if (count== 0) {
      unblock_user( );
} else {
      *printer_data_register = p[i];
      count = count − 1;
       i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

system call
interrupt handler
Printer ready?
copies 1st data to printer (starts transfer)
puts process that is printing into waiting-queue until someone calls "unblock_user()"
i/o request
raises interrupt
ends transfer
copies next data
i/o request
PRINTER
(a)
(b)

Figure 5.9: Writing a string to a printer interrupt-driven i/o. (a) code executed at the time the system call is made. (b) Interrupt service procedure for the printer

- The system call ends up with a call to the scheduler()
- The device (printer) raises an interrupt when it is ready and the interrupt handler drives the transfer of the next piece of data

(Fig. Source: Tanembaum, Modern Op. Systems, 3er ed, S. 5.2)*

Methods for i/o

i/o using DMA (Direct Memory Access)

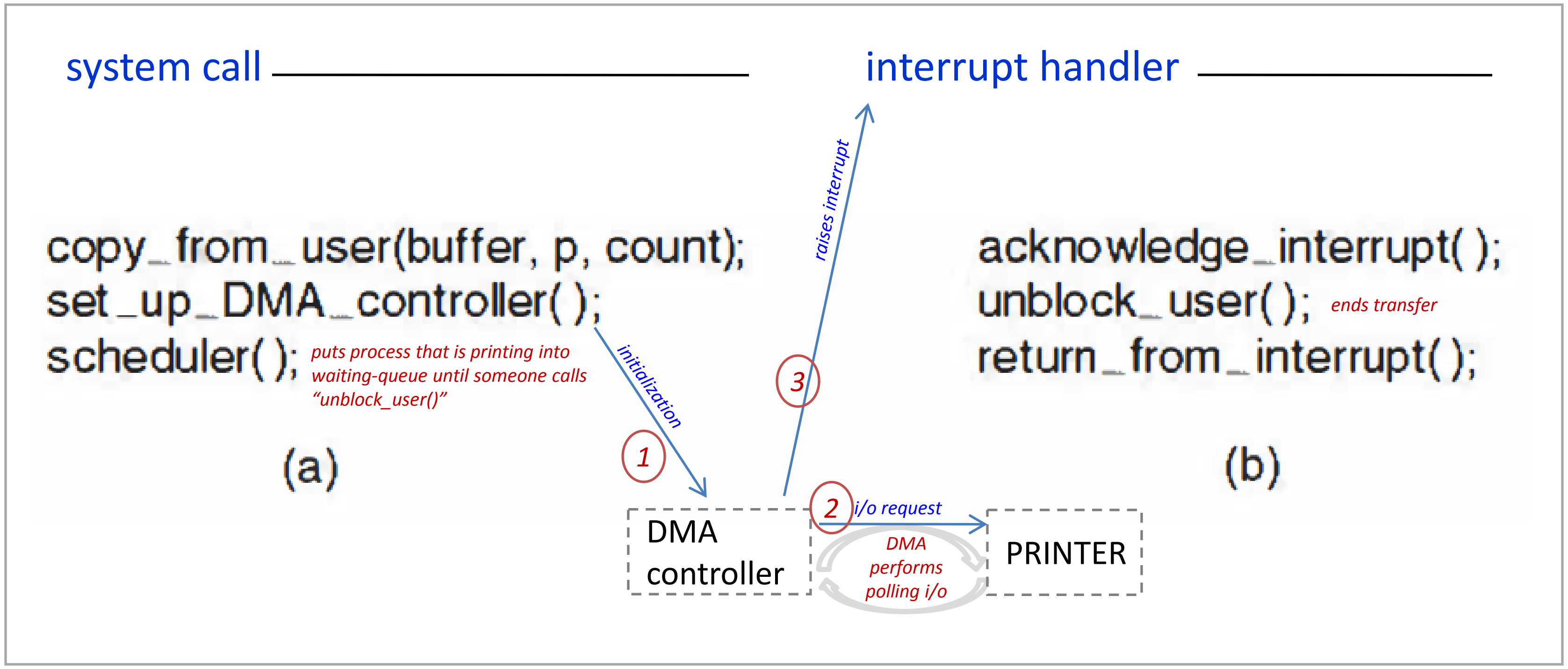
- In polling i/o and interrupt-driven i/o the CPU leads the data transfer when those data are ready: either checking the status register, or being notified by the device with an interrupt (and handling such an interrupt).
- This has the following drawbacks:
  - The transfer speed is limited by the speed of the CPU when moving those data from the CPU register/memory ←→ controller/device
  - While a data transfer is being done, the CPU must lead that transfer and cannot be attending other stuff
- These are minor issues in slow devices where few data are transferred, but they become a major issue with devices receiving/transferring a huge amount of information (i.e. a hard disk drive).
- Solution: **DMA Controller** (Direct Memory Access)

Polling i/o: CPU… *- starts transmission of each piece of data and,* *- waits until the item was actually transmitted*	*Interrupt-drive i/o: CPU…* *- starts transmission of each piece of data and,* *- executes Interrupt Service Procedure when an interrupt is raised by the device*	*i/o using DMA: CPU…* *- initializes i/o operation (once for a set of data),* *- executes the unique Interrupt Service Procedure when an ending interrupt is raised by the DMA controller*

(Fig. Source: Tanembaum, Modern Op. Systems, 3er ed, S. 5.2)

Methods for i/o

i/o using DMA (Direct Memory Access)



```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

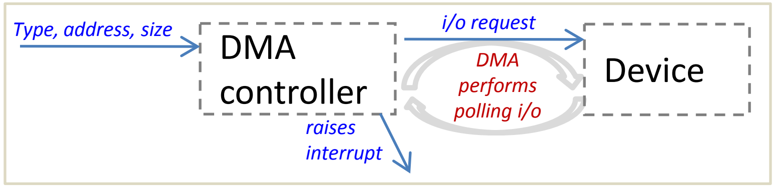
Figure 5.10: Writing a string using DMA. (a) code executed when the print system call is made. (b) Interrupt service procedure for the printer

- The system call ends up with a call to the scheduler()
- The DMA controller frees the CPU from leading the transfer of the buffer of data. CPU has only to attend the final interrupt that notifies that data transfer is complete

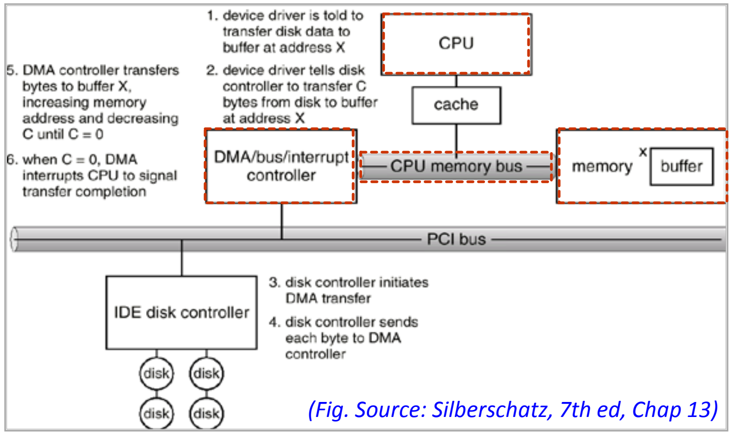
(Partial Fig. Source: Tanembaum, Modern Op. Systems, 3er ed, S. 5.2)

Methods for i/o

i/o using DMA (Direct Memory Access)



- The DMA controller is in charge of the data transfer to the device
- The DMA controller provides all the address signals and all the bus-control signals
- CPU must provide to the DMA controller the type of i/o operation (read / write), the memory address for the data transfer, and the amount of bytes to be transferred.
- When the data transfer completes the DMA controller raises an interrupt to notify the CPU
- The bus must be shared between the CPU and the DMA controller. There are several sharing methods
  - Burst mode DMA
  - Cycle-stealing DMA
  - Transparent Bus


1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
CPU
cache
DMA/bus/interrupt controller
CPU memory bus
memory
x
buffer
PCI bus
IDE disk controller
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
disk
disk
disk
disk
(Fig. Source: Silberschatz, 7th ed, Chap 13)

Methods for i/o

i/o using DMA (Direct Memory Access)

- Burst-mode DMA (*ráfaga)*
  - Once the DMA seizes the memory bus (that is, it has the control of the bus), it transfers a whole block, and the CPU has to wait until the transfer completes.
    - CPU has only to wait if it needs the bus, but it can continue accessing instructions and data from cache (L1,L2…)
  - It is the fastest method
    - Since "gaining access to the bus takes -some-time-" each time it seizes the bus it sends several words (block)
  - This method is used in secondary storage devices, such as disks.
- Cycle-stealing DMA
  - Each time DMA seizes the bus, it transfers a word and then returns bus control to CPU
  - The data transfer is done with a sequence of DMA cycles interleaved with CPU cycles. *For example, every "x" CPU instructions executed the DMA tries to gain bus control*
- Transparent Bus
  - The DMA does only use the bus when the CPU is not using it (i.e. when the CPU is decoding an instruction, using the ALU,…)
  - The CPU can still access both the data and instructions in the **cache** while the DMA is doing data transfer.

```
LOAD R1, 0xFFF0      --> memory access
LOAD R2, 0XFFE0      --> memory access
SUM R3, R2,R1        --> SUM in ALU
SAVE R3, 0XFFD0      --> memory access
```

INPUT/OUTPUT

Input/output

Structure of an i/o system

Structure of the i/o software

Types of input/output

Methods for input/output

Disk sheduling

Input/output in UNIX

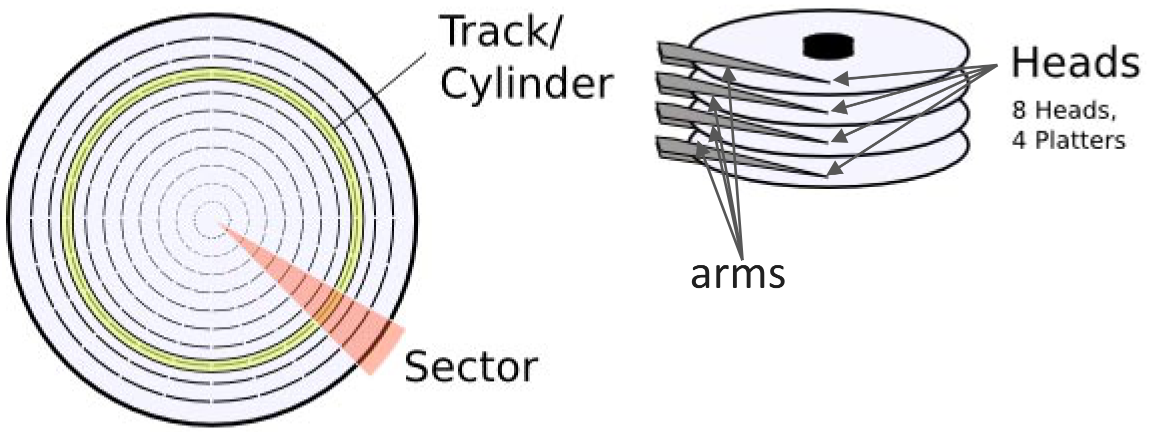
Disk scheduling

Disks

- A traditional disk is formed by a set of *platters* that spin together
- Each of the surfaces of the platters is commonly named a *side*
- Each side/surface is composed of a series of circular concentric *tracks*
- The set with the same track along the different sides makes up a *cylinder*
- Each cylinder contains a series of *sectors*
  - The sector is the basic i/o unit in a hard disk
  - Each sector has typically 512 bytes
- Each sector is perfectly referred to by its three coordinates: cylinder, side, and sector. Typically, the numbering of sides and cylinders starts with 0, and sectors are numbered from 1 on.
- Only the device handler knows the physical features of the disk. The O.S. sees the disk as a sequence of logical blocks.
  - For example: a 1.4Mbytes floppy disk contains 80 cylinders, 2 sides, and 18 sectors per track (2880 sectors of 512bytes each). If we format it with 4k blocks the O.S. will see it as a sequence of 360 blocks, and each block contains 8 sectors.
- The space assignment (and free space management) refers to blocks, not to sectors.

Disk scheduling

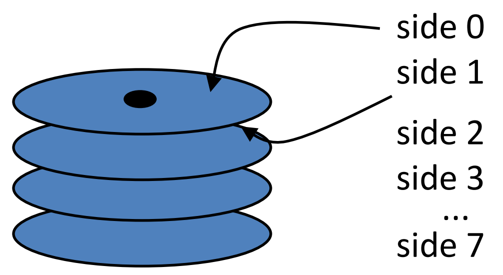
Disk structure

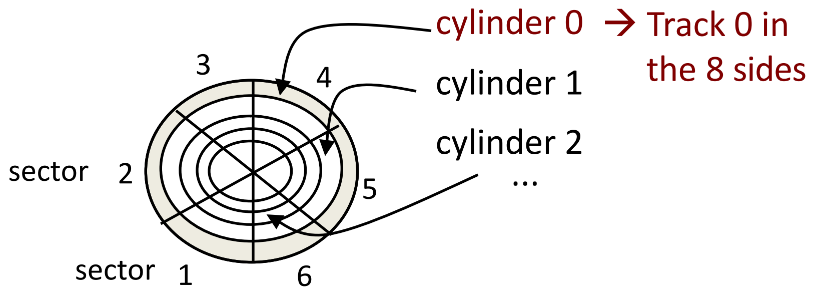


Disk scheduling

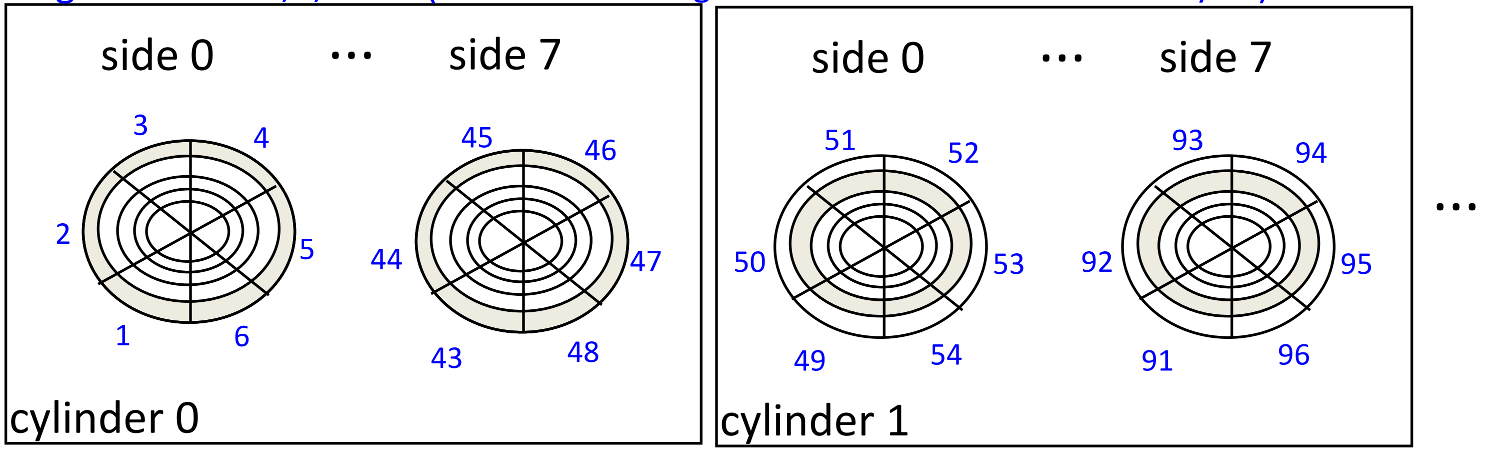
Disk structure

- Example:
  - Logical blocks 1,2,3,…N →correspondence to physical sectors (cylinder, side, sector)





Logical blocks 1,2,3..N (assume size of logical block = size of sector = 512bytes)



Disk scheduling

Disks

- Large disks can be split in one or more areas (i.e. partitions) that can be used as independent file systems
- Traditionally, the first sector** (side 0, cylinder 0, sector 1) contains a table that indicates de different areas of the disk
- The name and format of that table differs from one O.S. to another:
  - Partition table (MBR type), with 4 entries in windows and linux O.S's

    ** GUID Partition Table type (GPT) allows up to 128 entries (but uses more than 512 bytes for the partition table).
  - Disklabel, with either 8 or 16 entries in BSD systems…

Disk scheduling

Disk scheduling

- In a system where multiple i/o requests to disks are generated, such i/o requests can be planned/scheduled. Note that the O.S. is responsible of using hardware efficiently, and for disk drives, that means basically having a fast access time.
- Access time has two main components
  - **Seek-time:** time required to move the heads to the cylinder containing the desired block
  - Rotational-latency: time waiting for the disk to rotate the desired sector to the disk head
  - Other times such as block-transfer time can be assumed to be constant.
- We try to minimize seek-time ≈ seek distance
  - We could also try to reduce latencies (rotational scheduling), yet in modern systems, it is usual that the controller reads complete tracks and only the required sectors are transferred
- There are several algorithms to schedule the servicing of disk i/o requests.
  - **FCFS / FIFO** (*First Come First Served).* The requests are served in the same order they arrive.
  - **SSTF** (*Shortest Seek Time First*). The requests with shortest seek time from the current head position are served first. SSTF reduces the average seek-time, but it does not offer neither homogeneous nor predictible times. It may cause starvation (*inanición*) of some requests.

Disk scheduling

Disk scheduling

- **SCAN** (*Elevator algorithm*). The disk head starts at one end of the disk and moves toward the other end. It serves requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
  - The closest request is served but only in one direction
  - The central part of the disk gets more attention than the external part of the disk
- **C-SCAN**. Works similarly to SCAN, but when the head reaches the end of the disk it immediately returns to the beginning of the disk, without servicing any requests on the return trip (as if it were an elevator that stops only when going up).
  - The average wait time for each cylinder is more uniform than in SCAN
- **SCAN / C-SCAN with N steps**. Works as the regular counterparts, but in each trip it services at most N requests.
- **C-LOOK**. Variant of C-SCAN. The head only goes as far as the last request in each direction, then reverses direction immediately without first going all the way to the end of the disk.
  - It does not go up to the last cylinder when going up (it stops at the last cylinder requested)
  - It does not necessarily start going up from cylinder “0” (it starts from the first pending cylinder requested)
- …

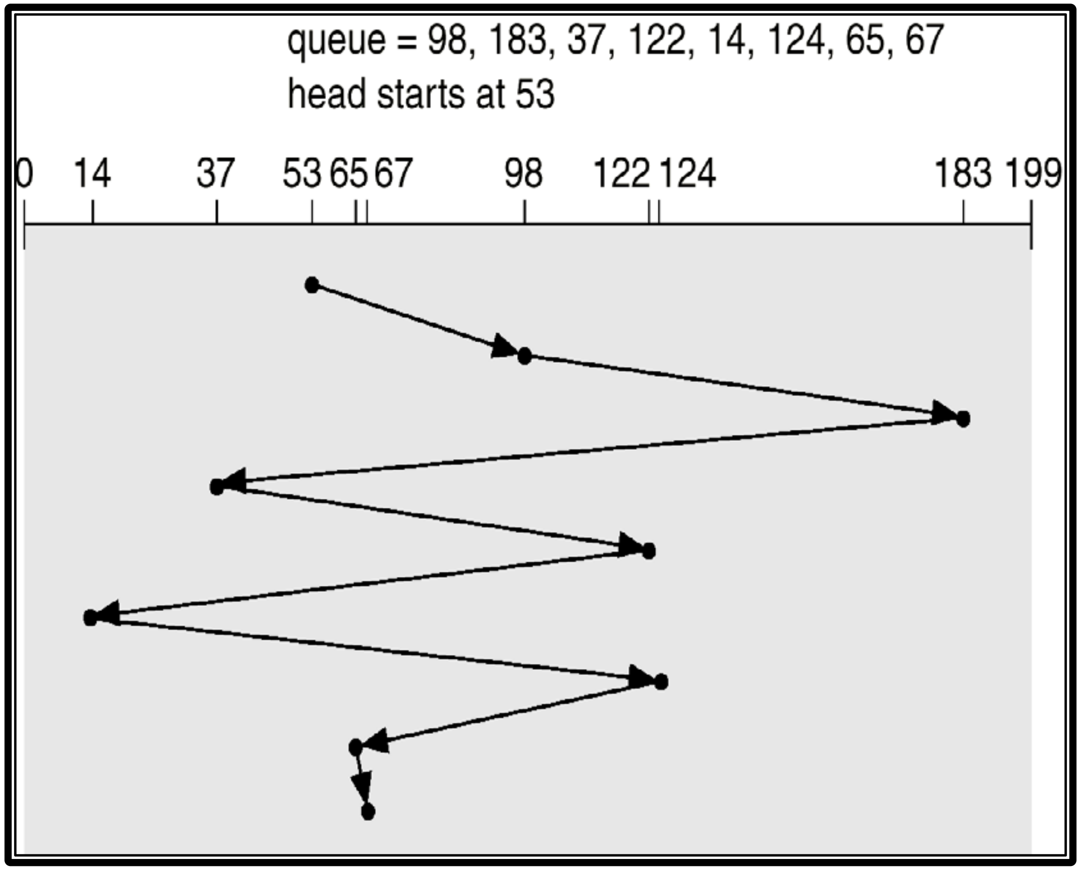
Disk scheduling

Disk scheduling

- Example: Serving requests for cylinders in the following request queue: <98, 183, 37, 122, 14, 124, 65, 67>
- Assume that:
  - There are only 200 cylinders [0..199]
  - The head is currently located over cylinder 53

Disk scheduling

FCFS (FIFO)



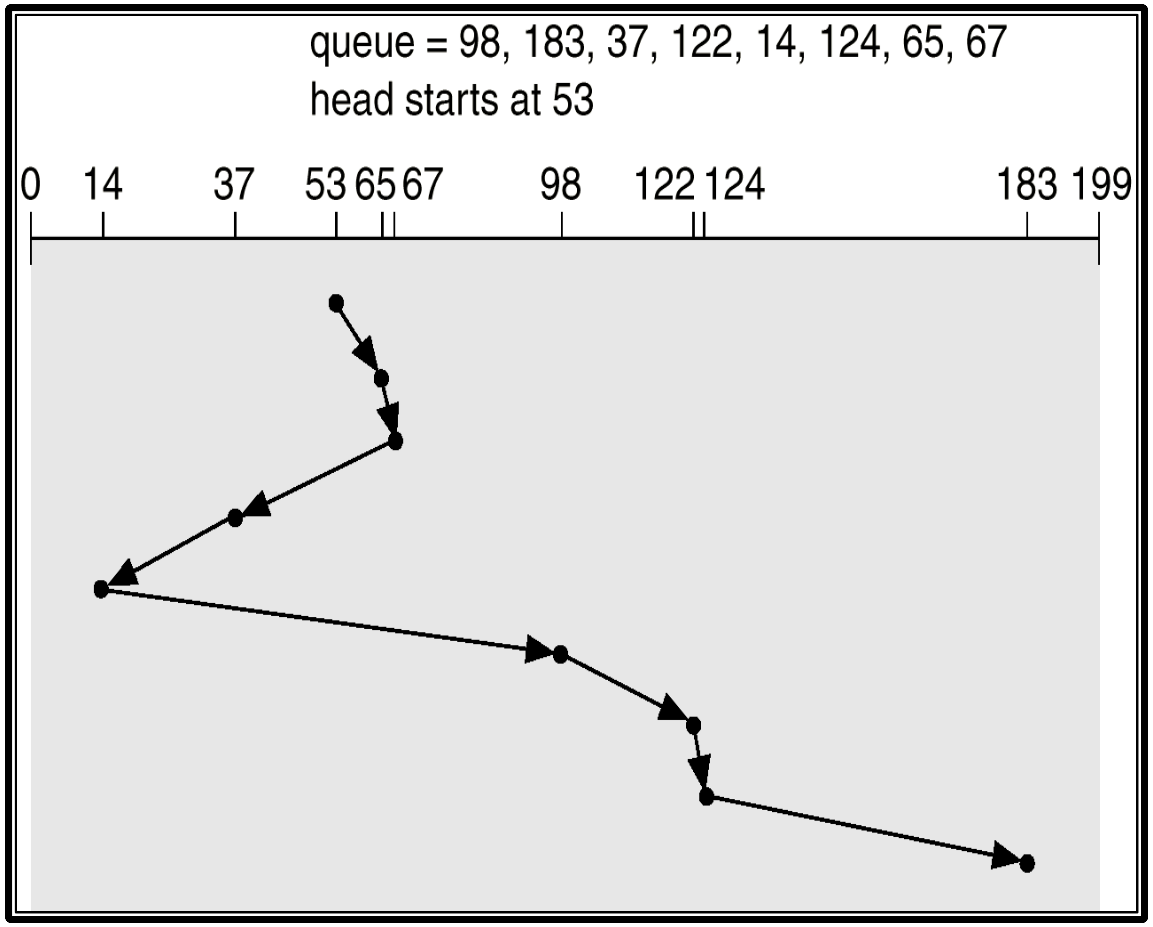
Operating System Concepts: Silberschatz et al, 2002

movement: 640 cylinders

(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html, chap 14)

Disk scheduling

SSTF

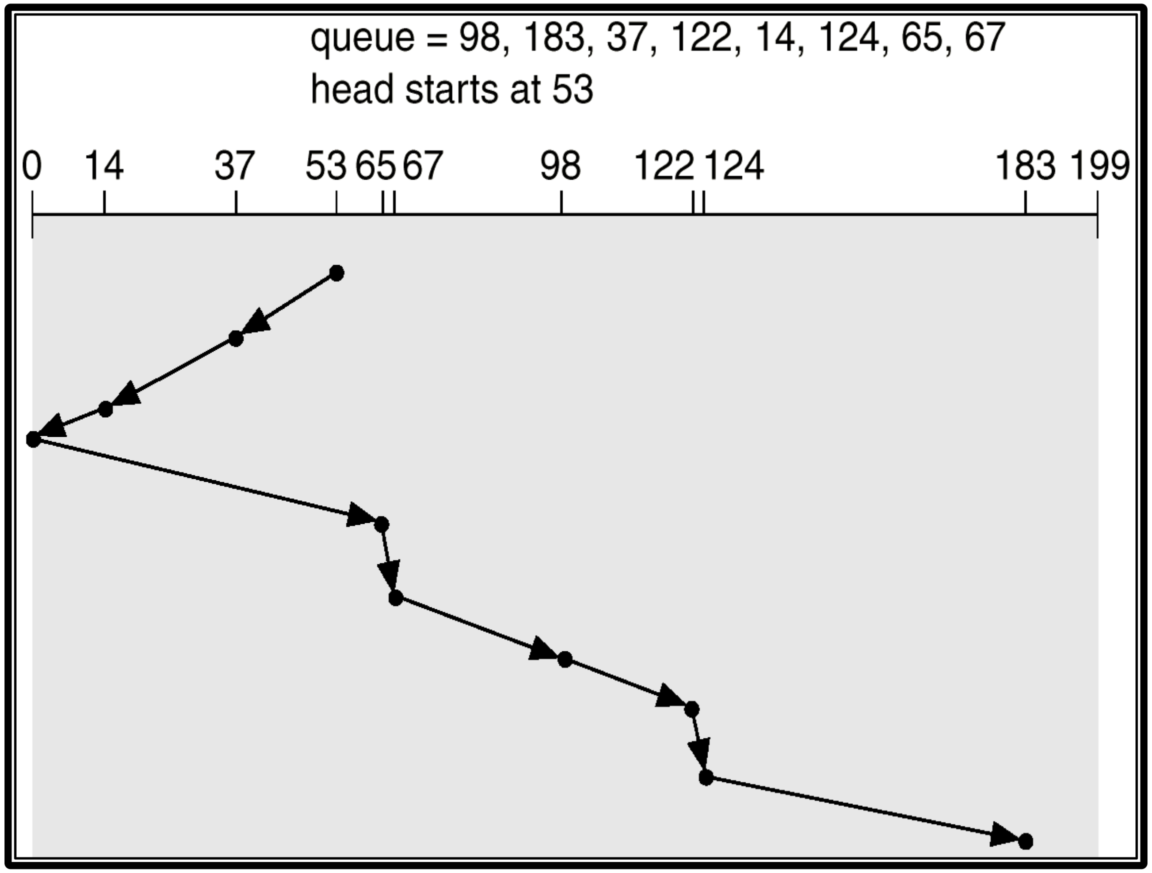


Operating System Concepts: Silberschatz et al, 2002

movement: 236 cylinders, but it could cause starvation

Disk scheduling

SCAN

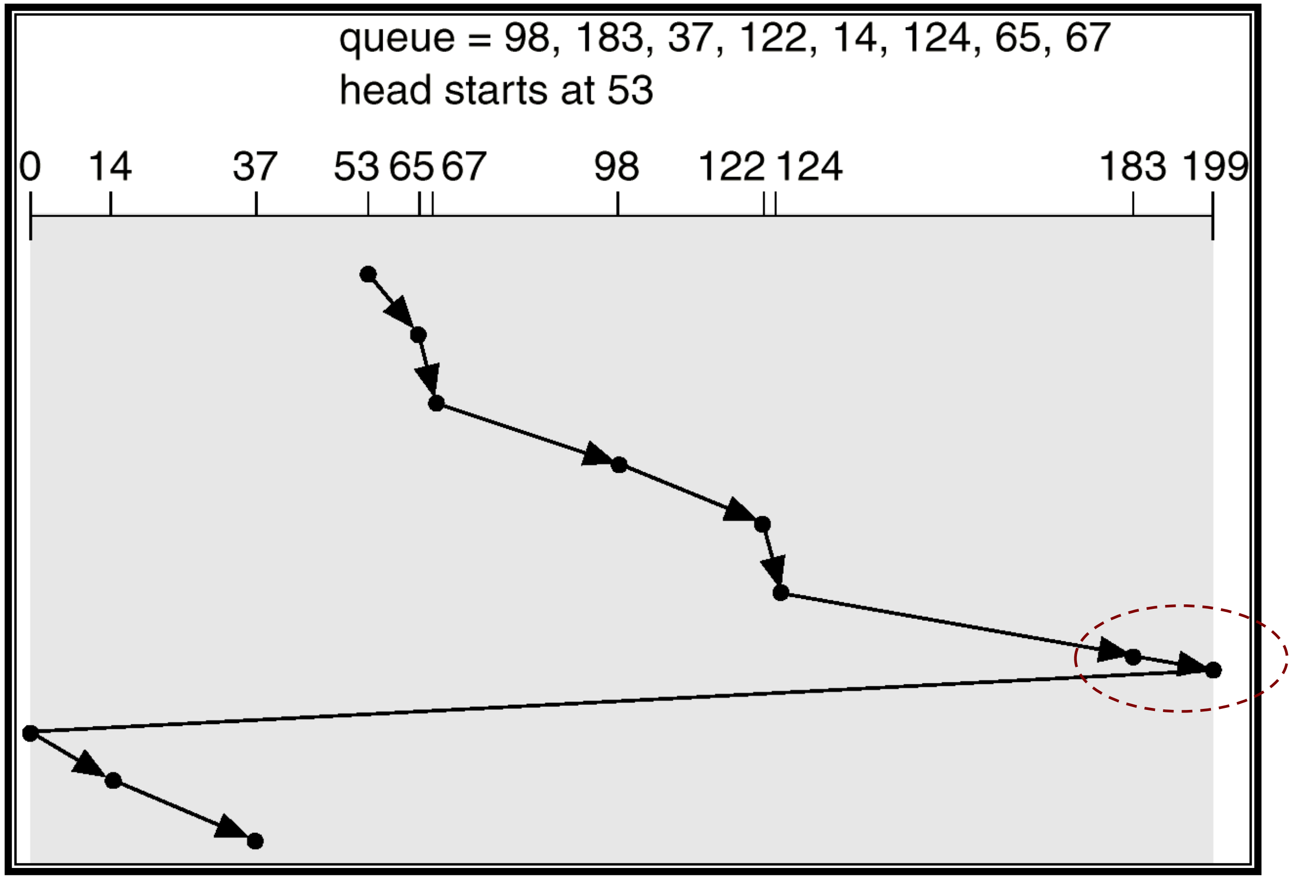


Operating System Concepts: Silberschatz et al, 2002

Initially going down (53→0)
movement: 208 cylinders

Disk scheduling

C-SCAN



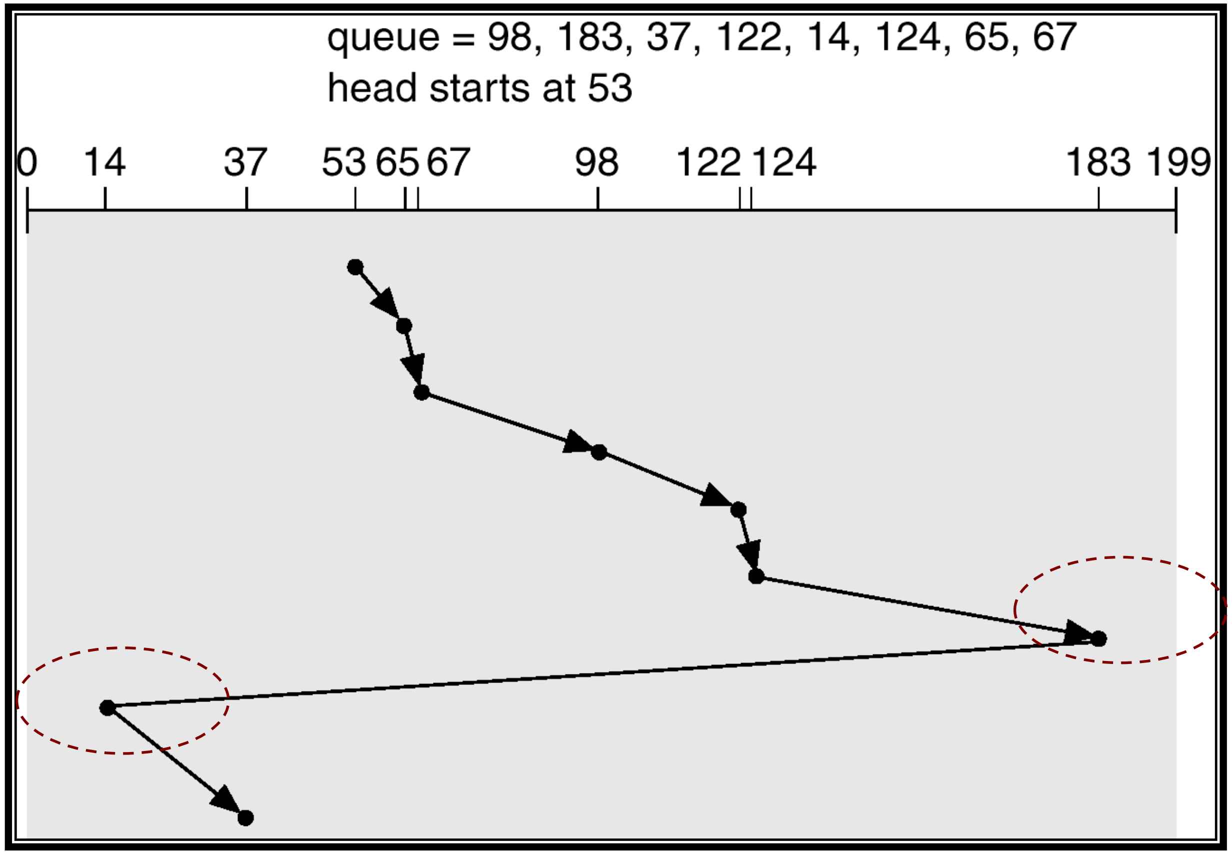
Operating System Concepts: Silberschatz et al, 2002

Initially going up(53+)
movement: 199-53 + 199 + 37 cylinders

Disk scheduling

C-LOOK (C-SCAN variant)

Recall it does not reach the end when going up (only up to the last request) . It starts going up, not from cylinder 0, but from the first pending request



Operating System Concepts: Silberschatz et al, 2002

Initially going up(53+)

movement: 183-53 + (183-14) + (37-14) cylinders

(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html)

Disk scheduling

Disk scheduling

- **CFQ** (*Complete Fair Queuing*). CFQ algorithm assigns different queues to the disk i/o requests from the different processes, and each queue is given time *quanta*. The length of the *quantum* and the number of requests that can be processed in a queue depend on the *i/o priority* of the process.
  - In Linux, ***ionice*** command allow us to modify the *i/o priority* of a process. ⇨
- Other schedulers in Linux; noop, deadline, anticipatory scheduling,...

Disk scheduling

```
IONICE(1)                          User Commands                          IONICE(1)

NAME
       ionice - set or get process I/O scheduling class and priority

SYNOPSIS
       ionice [-c class] [-n level] [-t] -p PID...
       ionice [-c class] [-n level] [-t] command [argument...]

DESCRIPTION
This program sets or gets the I/O scheduling class and priority for a program.  If no arguments or just -p
is given, ionice will query the current I/O scheduling class and priority for that process.

When command is given, ionice will run this command with the given arguments.  If no class is specified,
then command will be executed with the "best-effort" scheduling class  (-c 2).  The default priority level
is 4 (-n 4).

As of this writing, a process can be in one of three scheduling classes:
      • Idle (-c 3):  A  program  running  with idle I/O priority will only get disk time when no other
      program has asked for disk I/O for a defined grace period.  The impact of an idle I/O process on
      normal system activity should be zero. This scheduling class does not take a priority argument.
      Presently, this scheduling class is permitted for an ordinary user (since kernel 2.6.25).

      • Best-effort (-c 2): This is the effective scheduling class for any process that has not asked for a
      specific I/O priority.  This class takes a priority argument from 0-7, with a lower number being
      higher priority.   Programs  running  at the same best-effort priority are served in a round-robin
      fashion.
            • Note  that  before  kernel 2.6.26 a process that has not asked for an I/O priority formally
            uses "none" as scheduling class, but the I/O scheduler will treat such processes as if it were
            in the best-effort class.  The priority within the best-effort class will be dynamically derived
            from the CPU nice level of the process: io_priority = (cpu_nice + 20) / 5. [-c 2 :nice=0 →-n 4]

            • For kernels after 2.6.26 with the CFQ I/O scheduler, a process that has not asked for an I/O
            priority inherits its CPU scheduling class.  The I/O priority is derived from the CPU nice level
            of  the  process  (same as before kernel 2.6.26).

      • Realtime (-c 1): The  RT  scheduling  class is given first access to the disk, regardless of what
      else is going on in the system.  Thus the RT class needs to be used with some care, as it can starve
      other processes.  As with the best-effort class, 8 priority levels are defined denoting how big a time
      slice a given process will receive on each scheduling window.  This scheduling class is not permitted
      for an ordinary (i.e., non-root) user.
```

Disk scheduling

```
IONICE(1)                         User Commands                          IONICE(1)

OPTIONS
       -c, --class class: Specify the name or number of the scheduling class to use:
                          0 for none, 1 for realtime, 2 for best-effort, 3 for idle.

       -n, --classdata level: Specify the scheduling class data.  This only has an effect if the class
           accepts an argument.  For realtime and best-effort, 0-7 are valid data (priority levels).

       -p, --pid PID... : Specify the process IDs of running processes for which to get or set the
           scheduling parameters.

       -t, --ignore: Ignore failure to set the requested priority.  If command was specified, run it even in
           case it was not possible to set the desired scheduling priority, which can happen due to
           insufficient privileges or an old kernel version.


EXAMPLES
       # ionice -c 3 -p 89
       Sets process with PID 89 as an idle I/O process.

       # ionice -c 2 -n 0 bash
       Runs 'bash' as a best-effort program with highest priority.
       # ionice -p 89 91
       Prints the class & priority of the processes with PID 89 and 91.
```

```
user@pc:~$ ls -lsR / > /dev/null
user@pc:~$ ps -l
~~   PID  PPID  NI    ~~ CMD
~~  8301  8200   0    ~~ ls

user@pc:~$ ionice -c 3 -p 8301
user@pc:~$ ionice -p 8301
Idle

user@pc:~$ ionice -c 2 -n 0 -p 8301
user@pc:~$ ionice -p 8301
best-effort: prio 0

user@pc:~$ ionice -c 2 -n 1 -p 8301
user@pc:~$ ionice -p 8301
best-effort: prio 1

user@pc:~$ ionice -c 2 -p 8301
user@pc:~$ ionice -p 8301
best-effort: prio 4 [def prio]

user@pc:~$ sudo ionice -c 1 -p 8301
user@pc:~$ ionice -p 8301
realtime: prio 4
```

```
NOTES
       Linux supports I/O scheduling priorities and classes since 2.6.13 with the CFQ I/O scheduler.
```

INPUT/OUTPUT

I/O in Unix

I/o devices in Unix

- In UNIX devices appear as a file (typically in directory /dev). In some systems (i.e. solaris) this file is a symbolic link to the place where the actual device file is located.
- Devices are given an *inode*. Apart from indicating if such device is either a *block* or *character* device, this *inode* keeps two numbers (*major number* and *minor number*). This numbers indicate respectively *the device handler that is used to access the device*, and *which unit* *(among those managed by such device handler) is used.*

```
user@pc:~$ ls -li /dev/sda /dev/sda1 /dev/sda2 /dev/snd/seq /dev/snd/pcmC0D2c

 1209 brw-rw----  1 root disk    8, 0 nov 22 19:45 /dev/sda
 1216 brw-rw----  1 root disk    8, 1 nov 22 19:45 /dev/sda1
 1217 brw-rw----  1 root disk    8, 2 nov 22 19:45 /dev/sda2
10353 crw-rw---T+ 1 root audio 116, 1 nov 22 19:45 /dev/snd/seq
12340 crw-rw---T+ 1 root audio 116, 2 nov 22 19:45 /dev/snd/pcmC0D2c
```

- To access devices we can use the same calls used for accessing files (open, read, write) provided the caller process has the appropriate privileges.
- We can also access to the features (and additional functionalities) of the devices with the ***ioctl*** call.

I/O in Unix

I/o devices in Unix *(Example: major/minor numbers)*

Example: `$ls –li /dev/…`

- … shows the permissions (`brw-rw----`), the owner (`root`) ,the group (`disk, cdrom, tty,…`), the major device number (8), the minor device number (0), the date and hour, and the device name (`/dev/xxx`)
- When we access a device file,…
  - … the major number is used to select which device driver is called to perform the i/o operation. The minor number is passed as a parameter to this call. How the minor number is interpreted depends only on the driver. This behavior can be found in the driver documentation, which usually describes how minor numbers are used.

```
user@pc:~$  ls -li /dev/sd* /dev/sr0 /dev/tty1

 7588 brw-rw----  1 root disk   8,   0 sep 15 11:39 /dev/sda
 9329 brw-rw----  1 root disk   8,   1 sep 15 11:39 /dev/sda1
 9330 brw-rw----  1 root disk   8,   2 sep 15 11:39 /dev/sda2
 9331 brw-rw----  1 root disk   8,   3 oct 26 14:51 /dev/sda3
 9332 brw-rw----  1 root disk   8,   4 sep 15 11:47 /dev/sda4
~
10332 brw-rw----  1 root disk   8,  16 sep 15 11:39 /dev/sdb
10335 brw-rw----  1 root disk   8,  32 sep 15 11:39 /dev/sdc
11347 brw-rw----  1 root disk   8,  48 sep 15 11:39 /dev/sdd
11348 brw-rw----  1 root disk   8,  64 sep 15 11:39 /dev/sde
11366 brw-rw----  1 root disk   8,  80 sep 15 11:39 /dev/sdf
~
 1248 brw-rw----  1 root disk   8,  96 sep 15 11:39 /dev/sdg
 1249 brw-rw----  1 root disk   8,  97 sep 15 11:39 /dev/sdg1
 1250 brw-rw----  1 root disk   8,  98 sep 15 11:39 /dev/sdg2
 1251 brw-rw----  1 root disk   8, 101 sep 15 11:39 /dev/sdg5
~
 9302 brw-rw----+ 1 root cdrom 11,   0 nov  7 11:03 /dev/sr0
 1043 crw-rw----  1 root tty    4,   1 sep 15 11:47 /dev/tty1
```

```
Minor numbers sda*:
0+0  → sda
0+1  → sda1
0+2  → sda2
0+3  → sda3
0+4  → sda4
```

```
Minor numbers sdg*:
96+0  → sdg
96+1  → sdg1
96+2  → sdg2
96+5  → sdg5
```

I/O in Unix

I/o devices in Unix *(ioctl call)*

- We can also access to the features (and additional functionalities) of the devices with the ***ioctl*** call.

```
IOCTL(2)                          Linux Programmer's Manual                          IOCTL(2)

NAME
       ioctl - control device

SYNOPSIS
       #include <sys/ioctl.h>

       int ioctl(int d, int request, ...);

DESCRIPTION
       The  ioctl() function manipulates the underlying device parameters of special files.  In particular, many
operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests.  The
argument d must be an open file descriptor.
       The second argument is a device-dependent request code.  The third argument is an untyped pointer to memory.  It's
traditionally char *argp (from the days before void* was valid C), and will be so named for this discussion.
       An  ioctl()  request  has  encoded in it whether the argument is an in parameter or out parameter, and the size of
the argument argp in bytes.  Macros and defines used in specifying an ioctl() request are located in the file
<sys/ioctl.h>.

RETURN VALUE
ERRORS
CONFORMING TO
       No  single  standard.   Arguments,  returns, and semantics of ioctl() vary according to the device driver in
question (the call is used as a catch-all for operations that don't cleanly fit the UNIX stream I/O model). See
ioctl_list(2) for a list of many of the known ioctl() calls.  The ioctl() function call appeared in Version 7 AT&T UNIX.

NOTES
       In  order  to use this call, one needs an open file descriptor.  Often the open(2) call has unwanted side effects,
that can be avoided under Linux by giving it the O_NONBLOCK flag.

SEE ALSO
       execve(2), fcntl(2), ioctl_list(2), open(2), sd(4), tty(4)
```

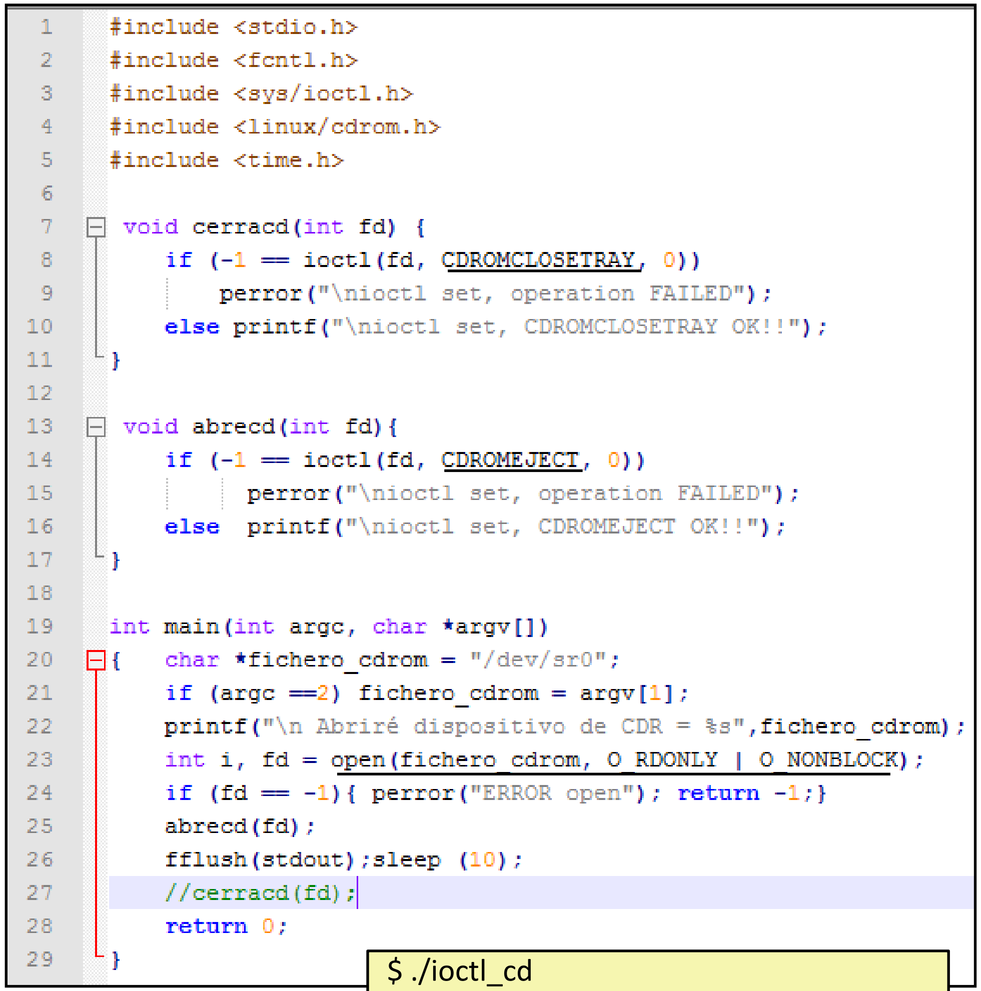
Including control functions that are not usually available via library-functions. For example, eject a cd-rom, change the connection speed of a modem (baud), etc.

I/O in Unix

I/o devices in Unix

Example: using ***ioctl*** *to deal with a cd-rom device*

$ **man ioctl_list** → see all available io_calls



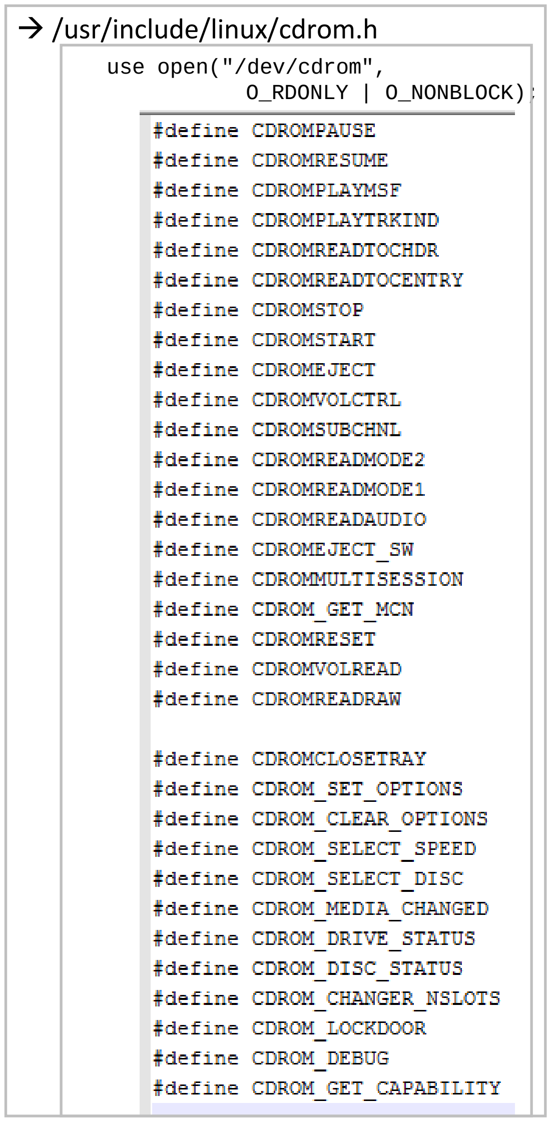
```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/cdrom.h>
#include <time.h>

 void cerracd(int fd) {
    if (-1 == ioctl(fd, CDROMCLOSETRAY, 0))
        perror("\nioctl set, operation FAILED");
    else printf("\nioctl set, CDROMCLOSETRAY OK!!");
}

 void abrecd(int fd){
    if (-1 == ioctl(fd, CDROMEJECT, 0))
          perror("\nioctl set, operation FAILED");
    else  printf("\nioctl set, CDROMEJECT OK!!");
}

int main(int argc, char *argv[])
{   char *fichero_cdrom = "/dev/sr0";
    if (argc ==2) fichero_cdrom = argv[1];
    printf("\n Abriré dispositivo de CDR = %s",fichero_cdrom);
    int i, fd = open(fichero_cdrom, O_RDONLY | O_NONBLOCK);
    if (fd == -1){ perror("ERROR open"); return -1;}
    abrecd(fd);
    fflush(stdout);sleep (10);
    //cerracd(fd);
    return 0;
}
```

```
$ ./ioctl_cd
  Abriré dispositivo de CDR = /dev/sr0
  ioctl set, CDROMEJECT OK!!
  ioctl set, CDROMCLOSETRAY OK!!
```



→ /usr/include/linux/cdrom.h

```
    use open("/dev/cdrom",
                  O_RDONLY | O_NONBLOCK);

#define CDROMPAUSE
#define CDROMRESUME
#define CDROMPLAYMSF
#define CDROMPLAYTRKIND
#define CDROMREADTOCHDR
#define CDROMREADTOCENTRY
#define CDROMSTOP
#define CDROMSTART
#define CDROMEJECT
#define CDROMVOLCTRL
#define CDROMSUBCHNL
#define CDROMREADMODE2
#define CDROMREADMODE1
#define CDROMREADAUDIO
#define CDROMEJECT_SW
#define CDROMMULTISESSION
#define CDROM_GET_MCN
#define CDROMRESET
#define CDROMVOLREAD
#define CDROMREADRAW

#define CDROMCLOSETRAY
#define CDROM_SET_OPTIONS
#define CDROM_CLEAR_OPTIONS
#define CDROM_SELECT_SPEED
#define CDROM_SELECT_DISC
#define CDROM_MEDIA_CHANGED
#define CDROM_DRIVE_STATUS
#define CDROM_DISC_STATUS
#define CDROM_CHANGER_NSLOTS
#define CDROM_LOCKDOOR
#define CDROM_DEBUG
#define CDROM_GET_CAPABILITY
```

I/O in Unix

I/o calls in Unix

- In UNIX, we use unformated i/o: we either read or write bytes. i/o calls receive a file descriptor, a memory address, and the number of bytes that must be transferred (as we will see below)
- **open**:

```
int open(const char *path, int flags, mode_t mode)
```

  - Receives a filename (*path*), an opening mode (*flags*, that can take values: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, … `O_APPEND`, `O_TRUNC`, `O_CREAT`, `O_EXCL`), and permissions (*mode*, in case the file is being created: i.e. `0777... S_IRUSR, S_IWUSR,...`)
  - Returns an integer that corresponds to the *file descriptor* (-1 on error). Such *file descriptor* will be used in subsequent calls to ***read*** and ***write***.

- **close**:

```
int close (int fd)
```

  - Closes a file descriptor (*fd*) previously obtained with *open*.

- **read and write**:

I/O in Unix

I/o calls in Unix

- **read and write**:

```
ssize_t read  (int fd,       void *buf, size_t count)
ssize_t write (int fd, const void *buf, size_t count)
```

  - Receive a file descriptor (*fd*), a source/target memory address (*buff*), and the number of bytes that must be transferred (*count*).
  - Return the number of bytes actually transferred (either read or written).
  - The reading/writing is done in the current position within the file (*off_t cur_pos*). Such *cur_pos* is updated after those calls.
  - The current position in the file (*cur_pos)* can be modified with a call to ***lseek***.

- **lseek**:

```
off_t lseek (int fd, off_t offset, int whence)
```

  - Receives a file descriptor (*fd*), the amount of bytes to move (*offset*) with respect to the third parameter (*whence*). This third parameter can take the values:
    - SEEK_SET: offset is related to the beginning (position=0) of the file → sets cur_pos = offset
    - SEEK_CUR: offset is related to the current position in the file → sets cur_pos = cur_pos + offset
    - SEEK_END: offset is related to the end of the file (size=N) → sets cur_pos = N-offset

I/O in Unix

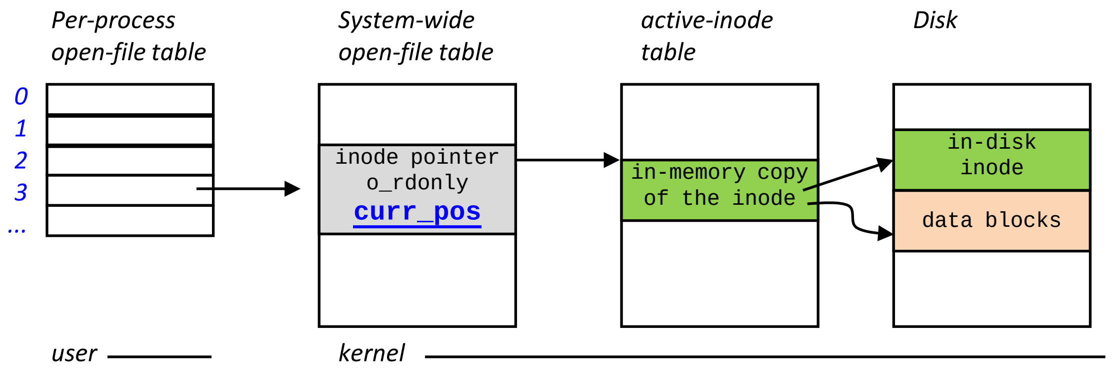
I/o calls in Unix

Example: open, close, read, write, lseek

```
int fd;
fd = open(file_name, .....);

read(fd, .....);
write(fd, ....);
lseek(fd, offset, SEEK_SET);
read(fd, .....);

close(fd);
```



I/O in Unix

I/o calls in Unix

- **pread and pwrite**:

Related to the beginning of the file

```
ssize_t read  (int fd,       void *buf, size_t count, off_t offset)
ssize_t write (int fd, const void *buf, size_t count, off_t offset)
```

  – Similar to *read* and *write* but they also receive the *offset* (bytes from the beginning of the file) where the read or writting is done.
  – These calls do not update the *cur_pos* in the file.

- **[p]readv, [p]writev: Calls for i/o in non-contiguous memory addresses**:

```
ssize_t readv  (int fd, const struct iovec *iov, int iovcnt)
ssize_t writev (int fd, const struct iovec *iov, int iovcnt)
ssize_t preadv (int fd, const struct iovec *iov, int iovcnt, off_t offset)
ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt, off_t offset)
```

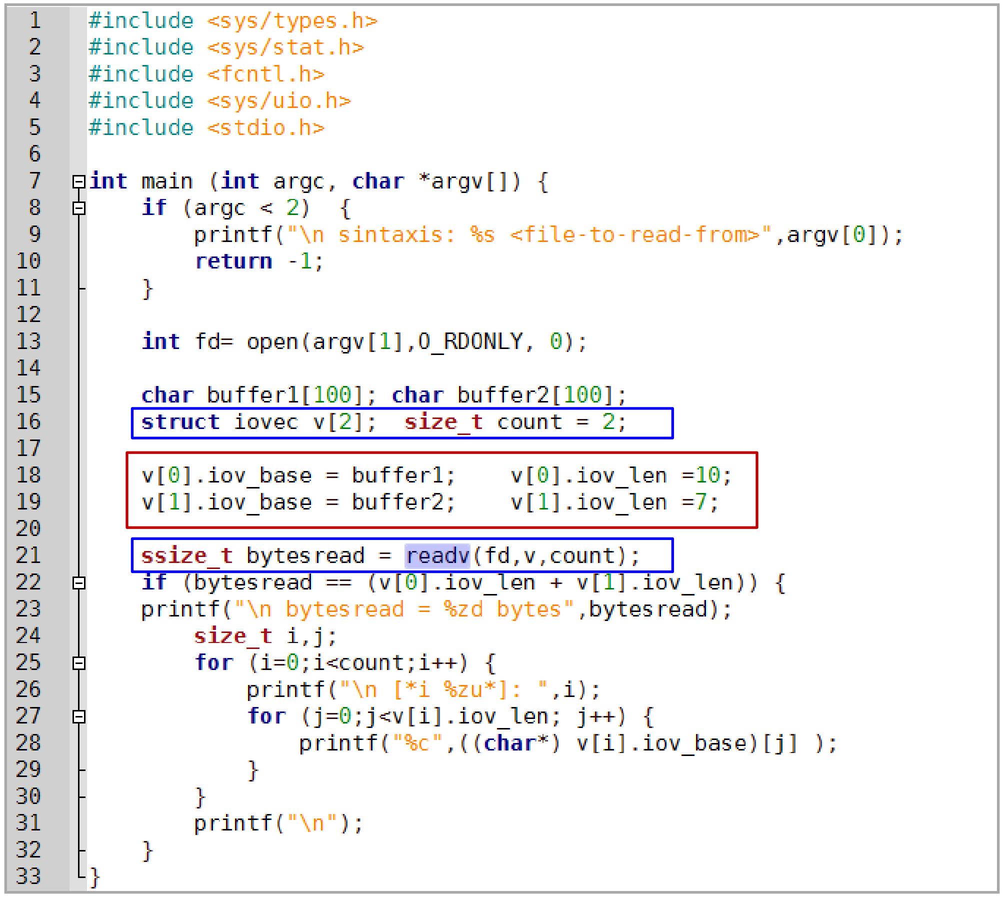
  – These calls are similar to *read, write, pread, pwrite*. However, instead of performing an i/o operation to a unique memory buffer, they perform i/o to several *iovcnt* different buffers that are described through *iov* array.
  – Each `struct iovec` describes a i/o buffer: its memory address *(iov_base),* and the number of bytes to transfer *(iov_len).*

```
struct iovec {
    void *iov_base;   /*starting address*/
    size_t iov_len;   /*number of bytes to transfer*/
}
```

I/O in Unix

I/o calls in Unix

- **readv example**



```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/uio.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    if (argc < 2)  {
        printf("\n sintaxis: %s <file-to-read-from>",argv[0]);
        return -1;
    }

    int fd= open(argv[1],O_RDONLY, 0);

    char buffer1[100]; char buffer2[100];
    struct iovec v[2];  size_t count = 2;

    v[0].iov_base = buffer1;    v[0].iov_len =10;
    v[1].iov_base = buffer2;    v[1].iov_len =7;

    ssize_t bytesread = readv(fd,v,count);
    if (bytesread == (v[0].iov_len + v[1].iov_len)) {
    printf("\n bytesread = %zd bytes",bytesread);
        size_t i,j;
        for (i=0;i<count;i++) {
            printf("\n [*i %zu*]: ",i);
            for (j=0;j<v[i].iov_len; j++) {
                printf("%c",((char*) v[i].iov_base)[j] );
            }
        }
        printf("\n");
    }
}
```

```
farinha@laptop:~$ cat a.txt
1234567890abcdefghijk

farinha@laptop:~$ ./a.out a.txt

  bytesread = 17 bytes
  [*i 0*]: 1234567890  //reads first 10 bytes and puts them into buffer1
  [*i 1*]: abcdefg     //reads  next 7 bytes and puts them into buffer2
```

I/O in Unix

Asynchronous input/output

- In UNIX processes have the chance to make an **asynchronous** i/o
  - They initiate the i/o and are notified when such i/o is completed
- The functions correspond to the *real time POSIX standard* (they must be linked with `-lrt`)
- **aio_read, aio_write, aio_return, aio_error,** and **aio_cancel**:

```
int aio_read (struct aiocb *aiocbp)  //Queues request: returns 0 on success, -1 on error
int aio_write(struct aiocb *aiocbp)  //Queues request: returns 0 on success, -1 on error
ssize_t aio_return(struct aiocb *aiocbp) //gets return status of asynchronous I/O operation
                                         //   (i.e. the same as the synchronous read,write)
int aio_error(const struct aiocb *aiocbp)//shows the error status of a previous async
                                         //operation
int aio_cancel(int fd, struct aiocb *aiocbp) //Tries to cancel a queued async i/o request
```

```
returned value of aio_error:(permits to know if the operation was actually completed && how it did)

        • EINPROGRESS, the operation is not completed yet
                     → We can check that while it returns EINPROGRESS → it is still in progress

        • ECANCELED, the request was canceled.

        •  0, the async i/o operation completed successfully.

        • >0, the async i/o operation failed. This returned value is the same value that would have
                   been stored in the errno variable in the case of a synchronous read(2), write(2)
```

I/O in Unix

Asynchronous input/output

- *struct aiocb* contains the following fields (and maybe others depending on the implementation, see *man aio*)

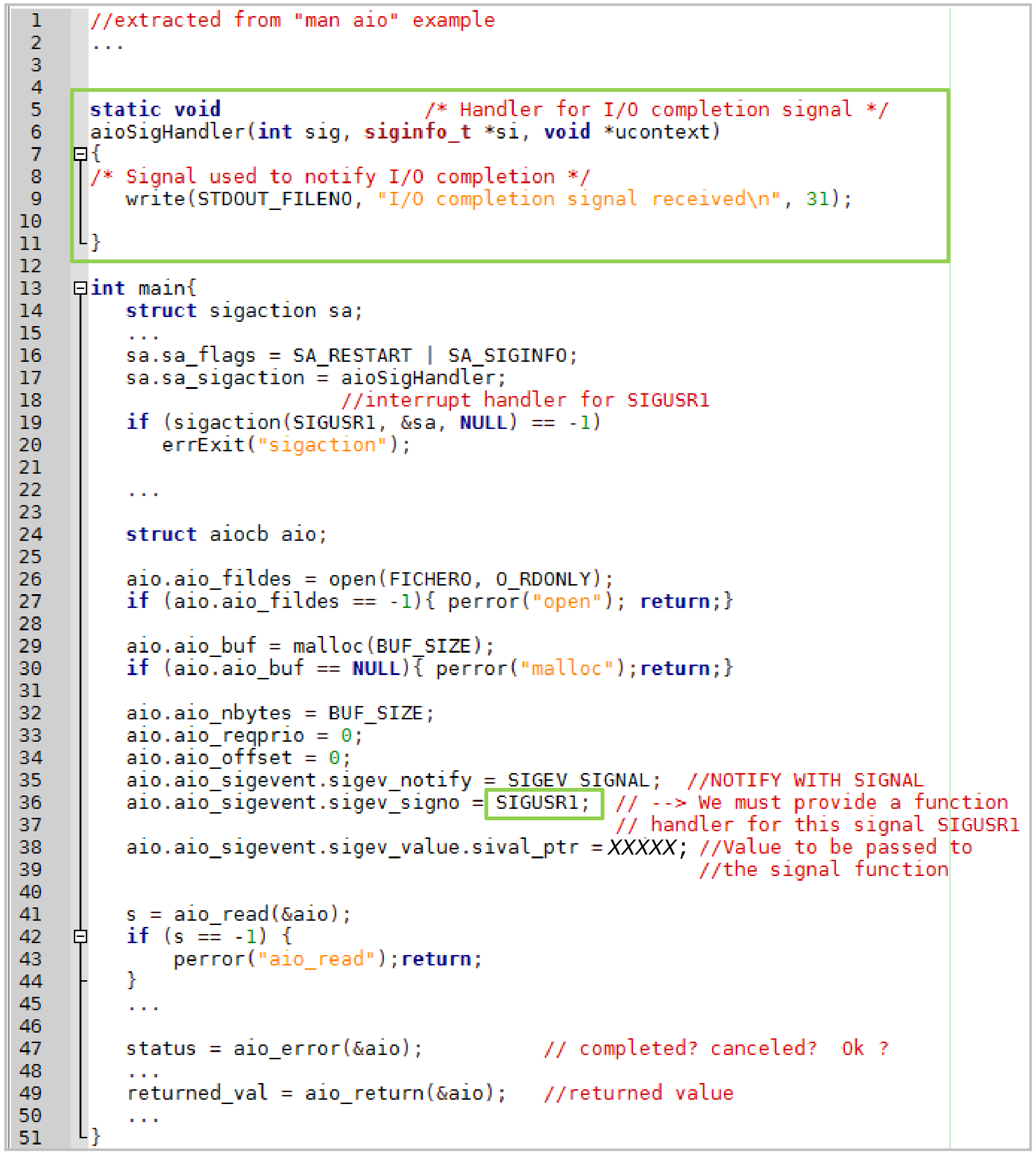
```
struct aiocb {
    int              aio_fildes;     /* File descriptor */
    off_t            aio_offset;     /* File offset */
    volatile void   *aio_buf;        /* Location of buffer */
    size_t           aio_nbytes;     /* Length of transfer */
    int              aio_reqprio;    /* Request priority */
    struct sigevent aio_sigevent;   /* Notification method */
    int              aio_lio_opcode; /* Operation to be performed; lio_listio() only */
    ...
};
```

  – **aio_sigevent**: This field is a structure that specifies how the caller is to be notified when the asynchronous I/O operation completes. Possible values for aio_sigevent.sigev_notify are SIGEV_NONE, SIGEV_SIGNAL, and SIGEV_THREAD. See sigevent(7) for further details.
    - FIELDS: aio_sigevent.sigev_notify = SIGEV_SIGNAL; aio_sigevent.sigev_signo=IO_SIGNAL; aio_sigevent.sigev_value = (value to pass to to the handler); aio_sigevent.sigev_notify_function; aio_sigevent.sigev_notify_attributes; aio_sigevent.sigev_notify_thread_id

I/O in Unix

Asynchronous input/output

- Async read example



```
//extracted from "man aio" example
...

static void                    /* Handler for I/O completion signal */
aioSigHandler(int sig, siginfo_t *si, void *ucontext)
{
/* Signal used to notify I/O completion */
   write(STDOUT_FILENO, "I/O completion signal received\n", 31);

}

int main{
   struct sigaction sa;
   ...
   sa.sa_flags = SA_RESTART | SA_SIGINFO;
   sa.sa_sigaction = aioSigHandler;
                        //interrupt handler for SIGUSR1
   if (sigaction(SIGUSR1, &sa, NULL) == -1)
      errExit("sigaction");

   ...

   struct aiocb aio;

   aio.aio_fildes = open(FICHERO, O_RDONLY);
   if (aio.aio_fildes == -1){ perror("open"); return;}

   aio.aio_buf = malloc(BUF_SIZE);
   if (aio.aio_buf == NULL){ perror("malloc");return;}

   aio.aio_nbytes = BUF_SIZE;
   aio.aio_reqprio = 0;
   aio.aio_offset = 0;
   aio.aio_sigevent.sigev_notify = SIGEV_SIGNAL;  //NOTIFY WITH SIGNAL
   aio.aio_sigevent.sigev_signo = SIGUSR1;  // --> We must provide a function
                                            // handler for this signal SIGUSR1
   aio.aio_sigevent.sigev_value.sival_ptr = XXXXX; //Value to be passed to
                                                    //the signal function


   s = aio_read(&aio);
   if (s == -1) {
       perror("aio_read");return;
   }
   ...

   status = aio_error(&aio);         // completed? canceled?  Ok ?
   ...
   returned_val = aio_return(&aio);   //returned value
   ...
}
```

I/O in Unix

Redirection of input, output, or error

- File descriptors 0, 1, and 2 (STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO) correspond respectively to the standard input, standard output, and standard error for a given process.
- The standard input and the standard ouput/error can be redirected to a given file.
  - For example, in bash this is done with symbols: '<' (input), '>'(output), '&>' (error)

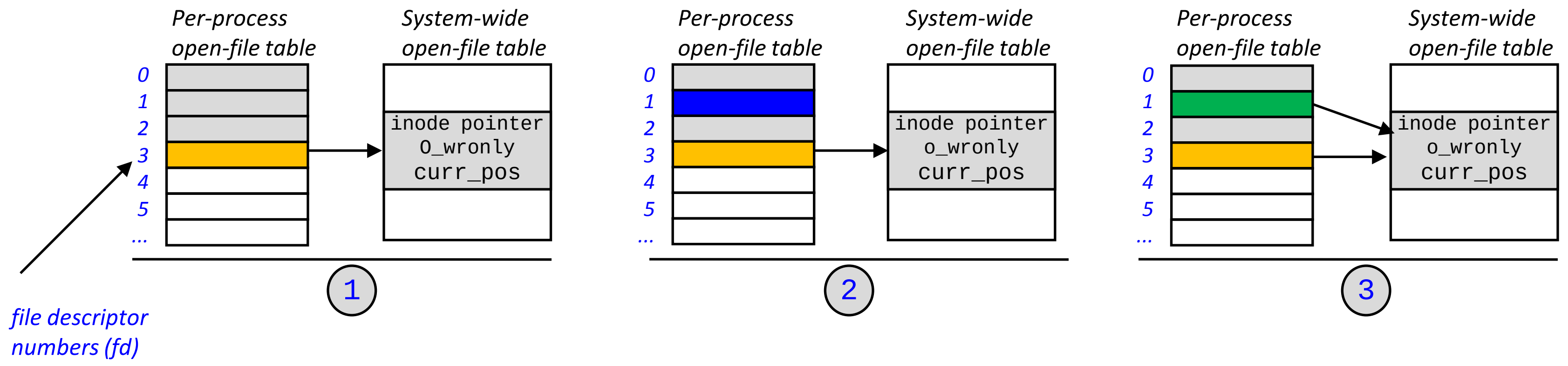
- Examples:
  1. Redirect the list of all the files in directory /usr to file "listing.txt"
     ```
     $ ls –l /usr > listing.txt
     ```
  2. Redirect the errors that arise when compiling program p1.c to file "errs.txt"
     ```
     $ gcc p1.c &> errs.txt
     ```
  3. Run grep to search for word "undefined" within file errs.txt
     ```
     $ grep undefined errs.txt  ## grep reads lines from file (argv[2]='errs.txt')
     $ grep undefined <errs.txt ## grep reads lines from std input (argv[2]=NULL)
     ```

I/O in Unix

Redirection of input, output, or error

- System calls ***dup***, ***dup2***, and ***fcntl*** with command F_DUPFD allow us to do the redirection
- **dup** duplicates the file descriptor, and uses the smallest available number (entry in the open-file table).
  - The following code shows how to redirect the standard output to a file 'out.txt' (no errors-control is included)

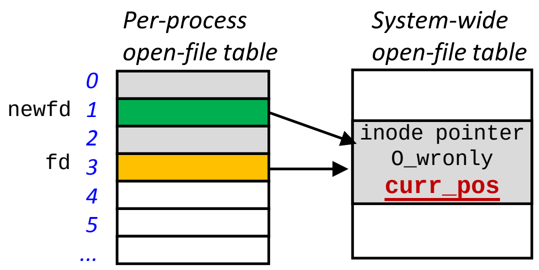
```
int fd= open ("out.txt", O_WRONLY);   1
close (STDOUT_FILENO);                2
dup (fd);                             3
... // from here on, standard output is redirected
```


Per-process open-file table
System-wide open-file table
0
1
2
3
4
5
...
inode pointer
O_wronly
curr_pos
1
2
3
file descriptor numbers (fd)

I/O in Unix

Redirection of input, output, or error

- **int dup(int fd)**
  - **int newfd= dup(fd):** both the *newfd* and the old *fd* refer to the same open-file, and consequently they share status flags such as the *curr_pos* (offset within the file)➔ if we call *lseek* over *fd* the *curr_pos* is also modified for *newfd* (and vice versa).

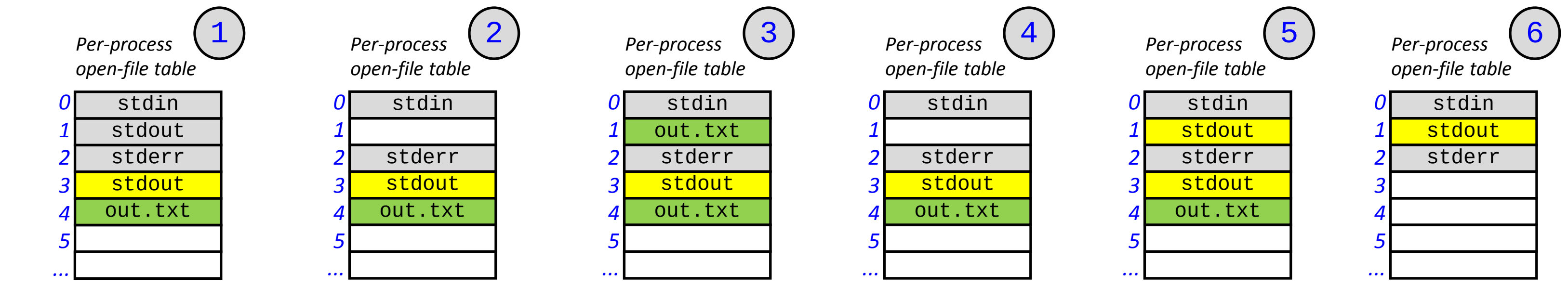


I/O in Unix

Redirection of input, output, or error

- If we want to **undo** the redirection, we have to keep a duplicate of the original file descriptor and then redirect back to that duplicate

```
    int backup, df;
 1  backup = dup(STDOUT_FILENO);
    fd = open(“out.txt”, O_WRONLY,...);
 2  close (STDOUT_FILENO);
 3  dup (fd);
    // from here on, standard output is redirected
    ...
    // now we undo redirection
 4  close(STDOUT_FILENO);
 5  dup(backup);
    close(fd);    /* if they are no longer used */
 6  close(backup);
```



```
backup = dup(STDOUT_FILENO);
fd = open(“out.txt”,O_WRONLY);
```
```
close(STDOUT_FILENO);
```
```
dup(fd)
```
```
close(STDOUT_FILENO);
```
```
dup(backup)
```
```
close(fd);
close(backup);
```

- It can also be done with dup2() or with fcntl()

I/O in Unix

Redirection of input, output, or error

- The following code executes a program (with arguments) with its standard output redirected to a file. The file, the program, and args are received as command-line parameters

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
void main (int arcc, char *argv[])
{
  int fd, backup, status;
  pid_t pid;
  if (argv[1]==NULL || argv[2]==NULL){
    printf ("use %s filename prog arg...\n",argv[0]);
    printf ("\t executes prog with its args. Output is redirected to filename\n");
    exit(0);
  }
  if ((fd=open(argv[1], O_CREAT|O_EXCL|O_WRONLY ,0777))==-1){        //creates output file
    perror("Unable to open target file for redirection");
    exit(0);
  }

  backup=dup(STDOUT_FILENO);                                         //copy of stdout
  close(STDOUT_FILENO); /*redirection*/  //file descriptor “1” now points to the output file “argv[1]”
  dup(fd);
  if ((pid=fork())==0){
    execvp(argv[2],argv+2);                     //child process receives a copy of the file descriptors where
    perror ("Unable to execute program");       //the output is already redirected.
    exit(255);
  }
  waitpid(pid,&status);
  close(STDOUT_FILENO); /*undo redirection*/
  dup(backup);
  if (WIFEXITED(status) && WEXITSTATUS(status)!=255)
    printf("program %s redirected to %s ended successfully(%d)\n",
                          argv[2],argv[1],WEXITSTATUS(status));
  else unlink(argv[1]);   //deletes output file
}
```