Filesystems part 1

Taco Walstra

Learning objectives

- Disk structure terminology: sector, block, track, seek time, settling time, (S)ATA, SCSI, SAS, RAID (only general ideas)
- Memory mapped I/O, instruction based I/O
- Disk scheduling algorithms: FCFS, SFFT, Elevator, C-scan
- Device drivers and devices (block-character)
- I/O scheduling: Deadline, CFQ, NOOP
- Filesystems: FAT,
  (next lecture) EXT, inode concept, journaling filesystems, log structured filesystems, network filesystems, distributed filesystems.

Objectives

- How can we find our data, programs etc. and save them?
- How can we protect our data? (We cover this later)
- How can we retrieve data as efficient as possible?
- How can we manage numerous filesystem and device types using an uniform interface

Filesystems - 2 perspectives

- Filesystems user perspective: names, operations, internal structure of files.
- Filesystem / system perspective:
  hard disk drives organisation, RAID systems
  device drivers
  File system implementations
  Example filesystems: FAT, (NTFS), EXT, NFS, xFS
  Journaling Filesystems, log structured filesystems
  Distributed Filesystems
  I/O schedulers

What is a file?

- Confusing terminology:
- Disk - a named location to store data
- UNIX/LINUX - disk file or special file (e.g. block device)
- Application - external object to read from or write data to (disk file, pipe, communication channel, device..)
- C - FILE refers to a data structure which contains information about opened buffered files (indirectly referring to actual files on disk or comm. channel!)

Files - user perspective

- Organisation: Filenames, extensions, date/time, type, directory references, links
- Operations: delete, create, read/write, link to application, positioning for read/ write
- Protection: read/write/ execute. Who is allowed what?
- Structure: bytes, record size, physical size.

Files - system perspective

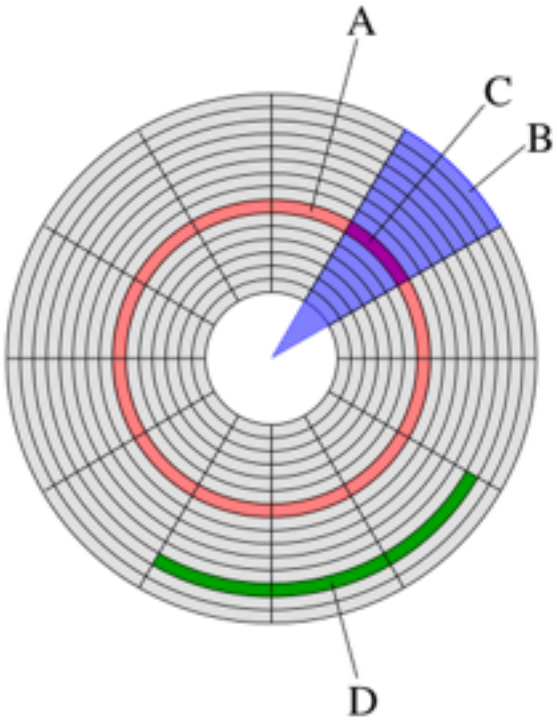
- Organisation: claim disk space, administration of blocks
  Create link to “i-nodes”
  Organizing filesystems on multiple disks, disk mounting (incl. network)
- Protection: users, groups, world. Linux views, Windows views
- Active filesystems: data structures in memory, caching, performance issues, I/O scheduling

IBM model 350 disk
5 MB (!) on 50 24” disks

Bottom-up: disk structures

- A hard disk has tracks (A)
- Sectors (B) of 512 byte (the number varies according to the track radius)
- Blocks of sectors (D). A block spans n sectors. Block size is determined during the format

  See it (Linux) with **fdisk -l** or
  blockdev --getbsz /dev/sda
  Can be set with **dd -bs=xxx** cmd
- Often 1 block = 8 sectors. Blocks are relevant in a filesystem context




```
Disk /dev/sda: 256.1 GB, 256060514304 bytes, 500118192 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x00073c5c

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *        2048   104859647    52428800   83  Linux
walstra@u031012:~$
```

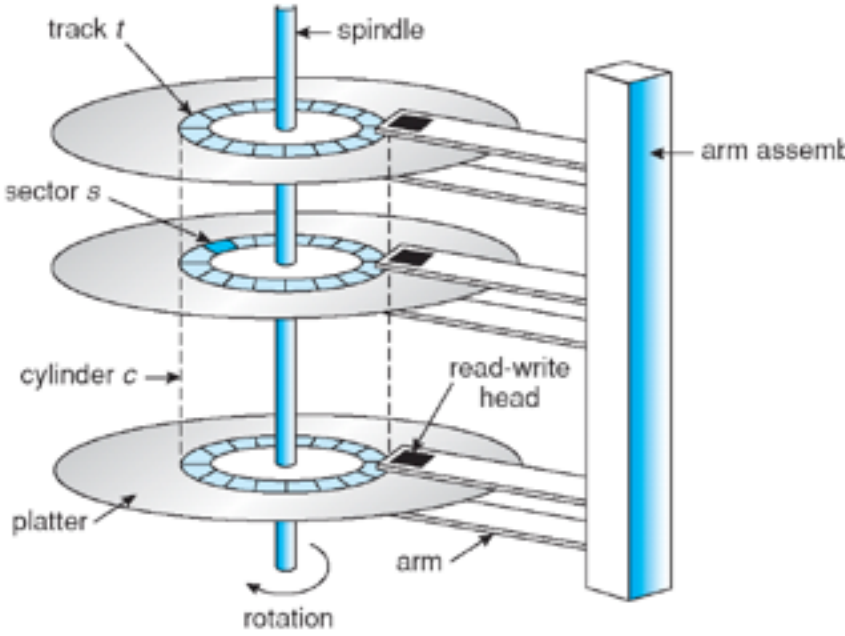
Choosing block size

- Block size often 4 kb (8 sectors).
- Large blocks:
  - less disk access, simpler administration
  - waste of disk space for small files
- 4.3 BSD UNIX and later use 1kb and 8 kb blocks

```
root@u031012:/usr/lib/modules$blockdev --getbsz  /dev/sda
4096
root@u031012:/usr/lib/modules$
```

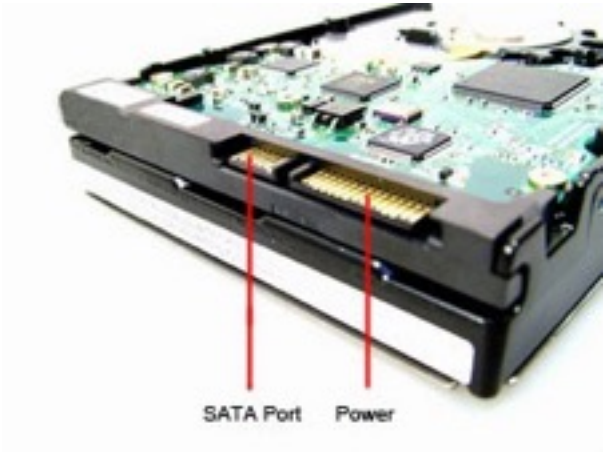
Hard disk parameters

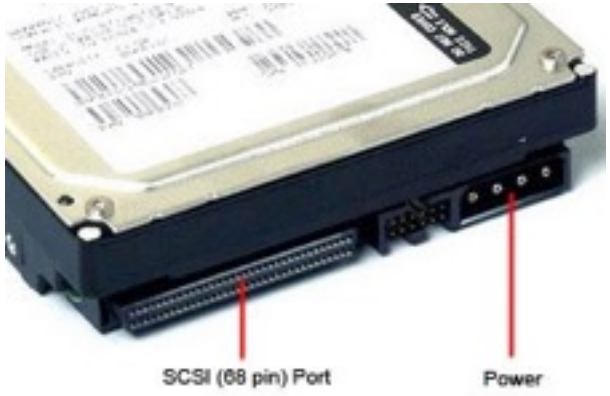
- Simple example: Track with sectors. Modern disks: millions of tracks
- Each sector contains 512 bytes
- Seek time: time to travel from one track to another and find a certain sector.
  4 phases: speedup-coast-slowdown-settle (accelerations of 40g!)
  A setting time determines the time to be certain to be in the correct track.
  Please note: this is a classical control system
  typical numbers: Barracuda 1 TB: 9 ms average seek
  7200 rotations per minute (rpm). 105 MB/s, 4 “platters”



Disk interface

- disk interface to CPU by a bus: ATA, SATA, SCSI, SAS, USB
- possible features:
  - disconnect from bus during requests
  - command queue: give disk multiple requests (disk has it's own logic to schedule requests using rotational information)
  - disk cache for read-ahead
    - sequential reads would incur whole revolution
    - cross track boundaries
  - some disk support write caching





scsi - sata - sas

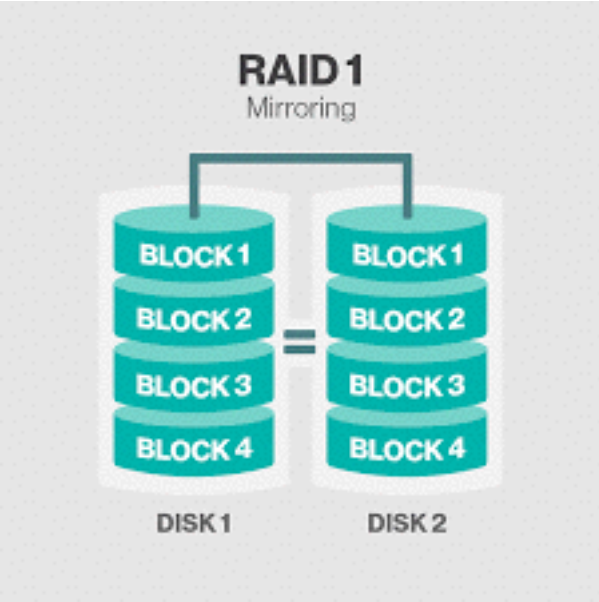
- SCSI (small computer system interface)
  - parallel interface (orig. apple macintosh, many unix systems (pdp-11)
  - high data transfer rates (up to 80 MB/s)
  - multiple devices can be attached to a single SCSI port (so it's more a bus than an interface)
- (S)ATA (Serial AT Attachment)
  - high data rates (150 MB/s), SATA 3: 6 GB/s
  - mainstream interface in desktop PC, cheap, MTBF 700k-1M hours
- SAS (serial attached SCSI)
  - evolution of scsi: up to 128 devices of different size/type to be connected simultaneously together with thin, long cables.
  - very high data rates SAS 1: 3 GB/s, SAS 4: 22.5 GB/s (2017)
  - hot pluggable
  - MTBF (mean time before failure) millions of hours

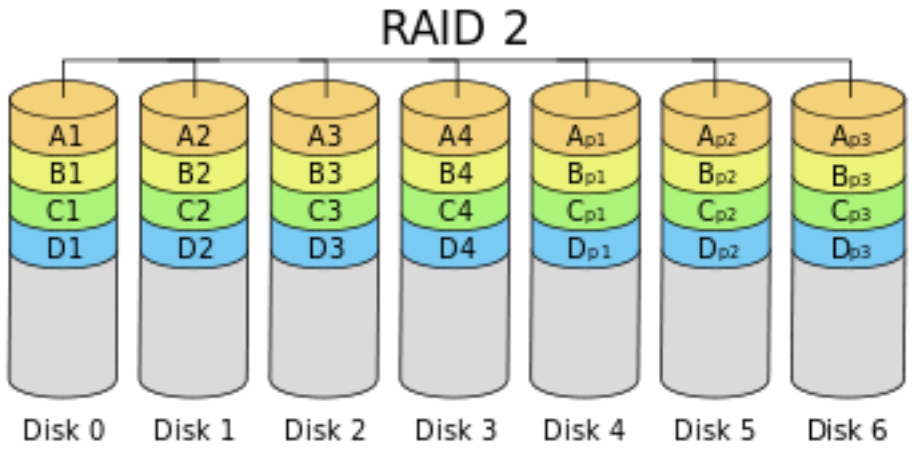
RAID structures

- RAID: Redundant Arrays of Independent Disks (in the past “Inexpensive”)
  - Performance
  - Reliability
- Performance improvement by parallelism
  - bit-striping: split the bits of each byte across multiple disks: 8 disks will lead to 8x access rate.
  - block-level striping: striping at the level of blocks across multiple disks: n disks: block i of a file goes to disk (i mod n) + 1
- Reliability can be increased using redundancy
- RAID types are numbered: 0-1-2 etc.
- You do NOT need to remember the differences. Raid concepts like striping, recovery etc. however you need to know.

RAID structures

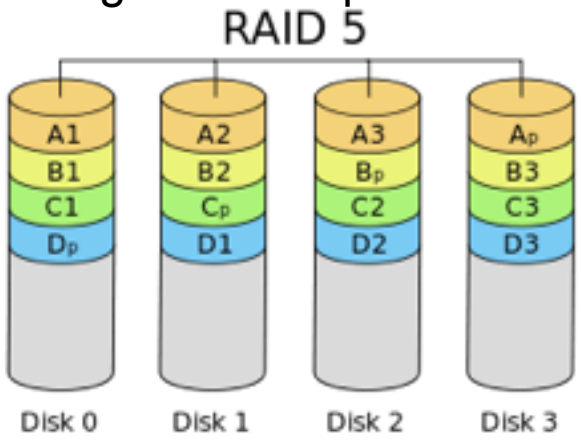
- RAID level 0: block level striping but NO redundancy
- RAID level 1: Mirror the disks to another set of disk
- RAID level 2: use parity. Error Correcting Organization (ECC)
  - parity=0 if the number of bits in a byte is even
  - parity=1 if the number of bits in a byte is odd
  - If one bit is damaged the parity does not match with stored parity.
  - Rarely used. bit striping across disks





RAID structures

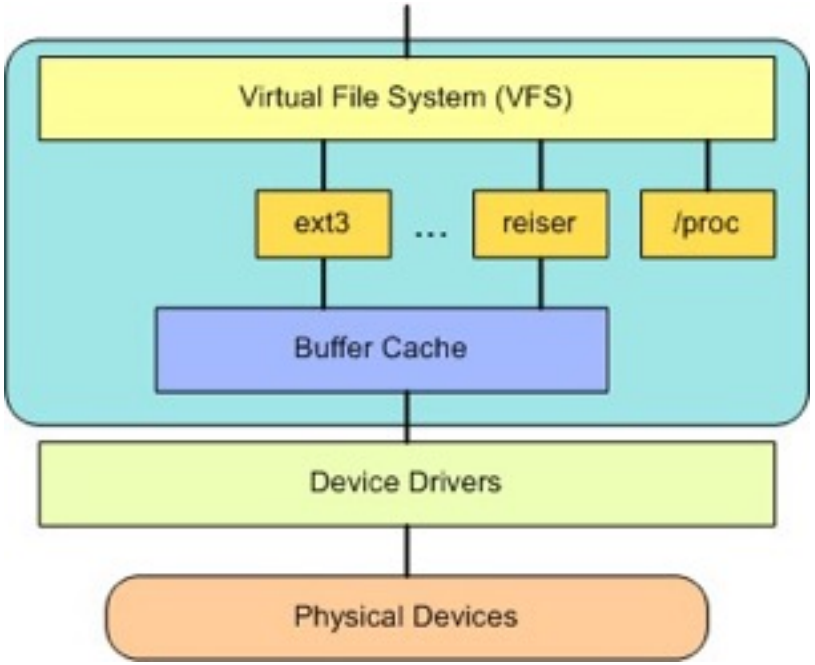
- RAID 3: builds on RAID 2: instead of bit striping it uses byte striping with a separate disk for the parity. In practice not used.
- RAID 4: the striping is on block-level with a separate parity disk
  - good performs for random reads. Not very good for writes because it needs to write the parity separately to a separate disk
- RAID 5: uses block level striping across disk with also the parity distributed over the disks.
  - when one disk has a failure the other disks are able to reconstruct the data from the distributed parity on the other disks.
  - Very common disk model
- RAID 6: like RAID5 but stores redundant information against multiple disk crashes.



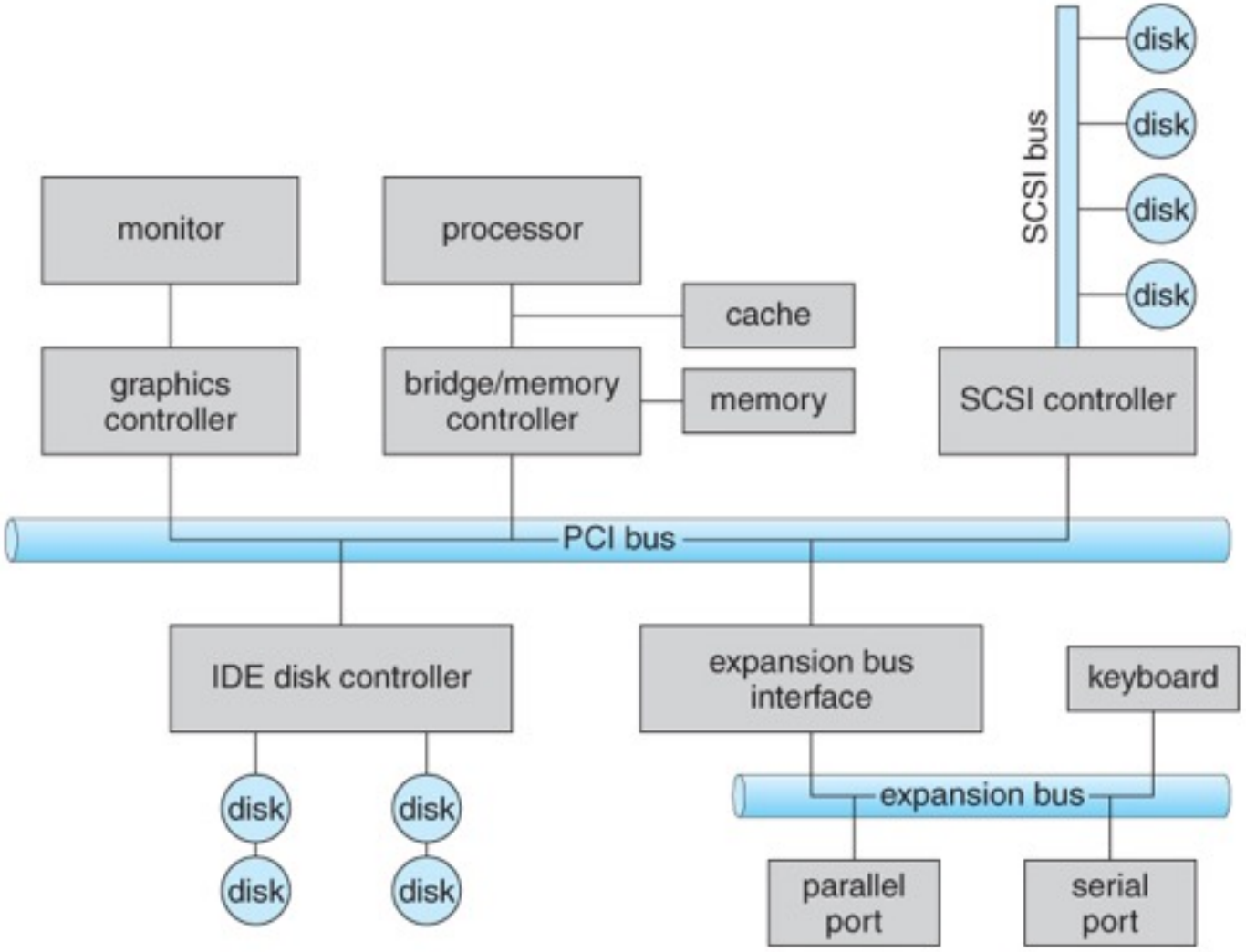
RAID recovery

- Example what happens with a disk crash:
- 4 drives. We store 3 datablocks of 3 bits in this case: 101, 010, 011
  Drive #1: 101
  Drive #2: 010
  Drive #3: 011
  Drive #4: (101 XOR 010 XOR 011) = 100 (parity)
- When disk #2 crashes:
  Data of disk #2: 101 XOR 011 XOR 100 = 010 e voila we are back

Where are we?


Virtual File System (VFS)
ext3
...
reiser
/proc
Buffer Cache
Device Drivers
Physical Devices

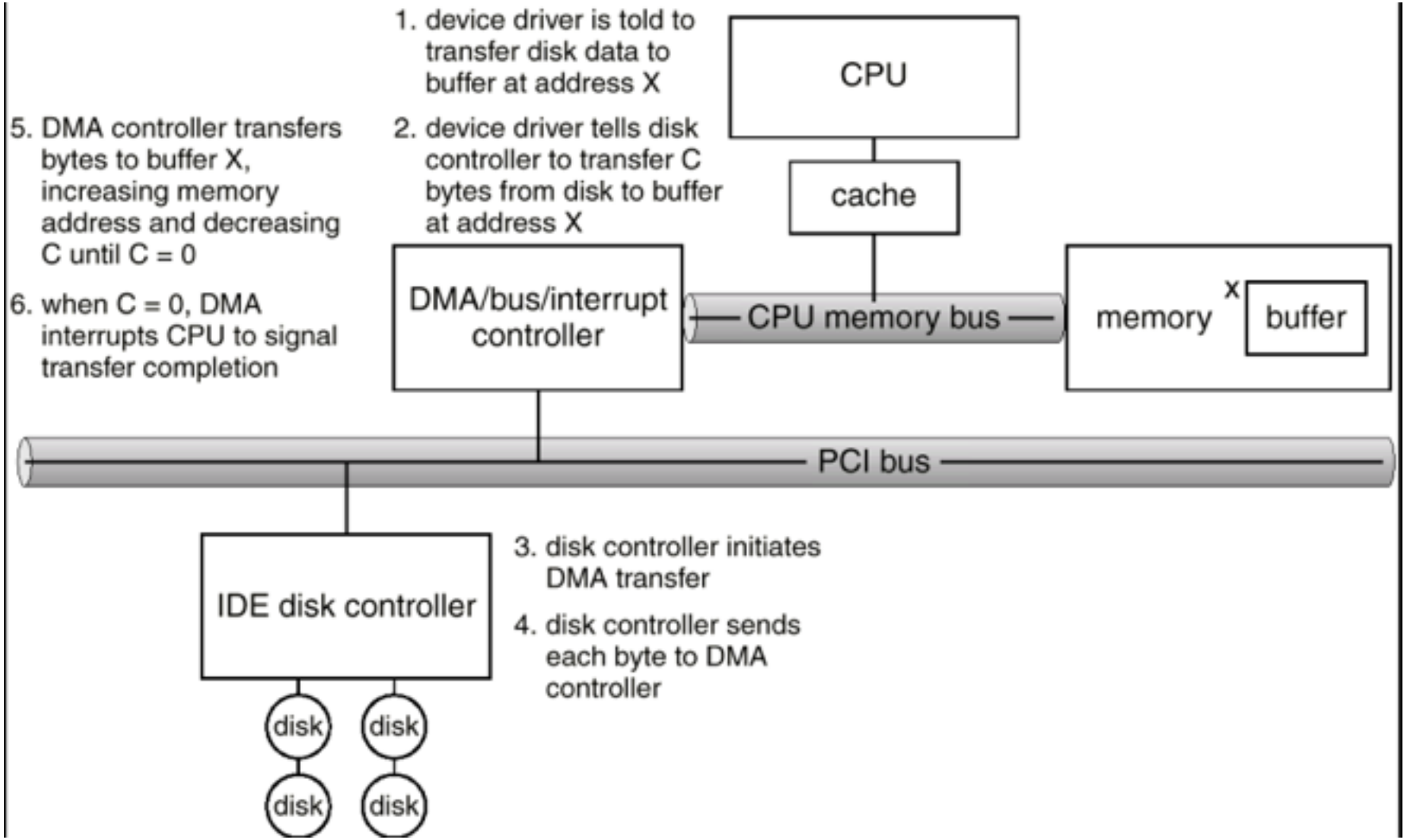
I/O devices


disk
disk
disk
disk
SCSI bus
monitor
processor
cache
graphics controller
bridge/memory controller
memory
SCSI controller
PCI bus
IDE disk controller
expansion bus interface
keyboard
disk
disk
disk
disk
expansion bus
parallel port
serial port

Evolution of I/O

- Polling an I/O device:
  While (status == BUSY);
  CPU sends data to data registers
  CPU sends command to command register
  while (status == BUSY)
- Used in PDP-1 (1959), PDP-8 (1965)
- Note: all the layers between are missing!
- Evolution:
  - Add a controller or I/O module which takes care of the low level I/O handling
  - Use interrupts to communicate with this controller
  - I/O module can access memory using DMA (direct memory access)
  - I/O module becomes a separate processor
  - I/O module has it's own memory and CPU creating a new computer system of it's own
  - I/O module can have a separate interface to CPU (non-pci for example)

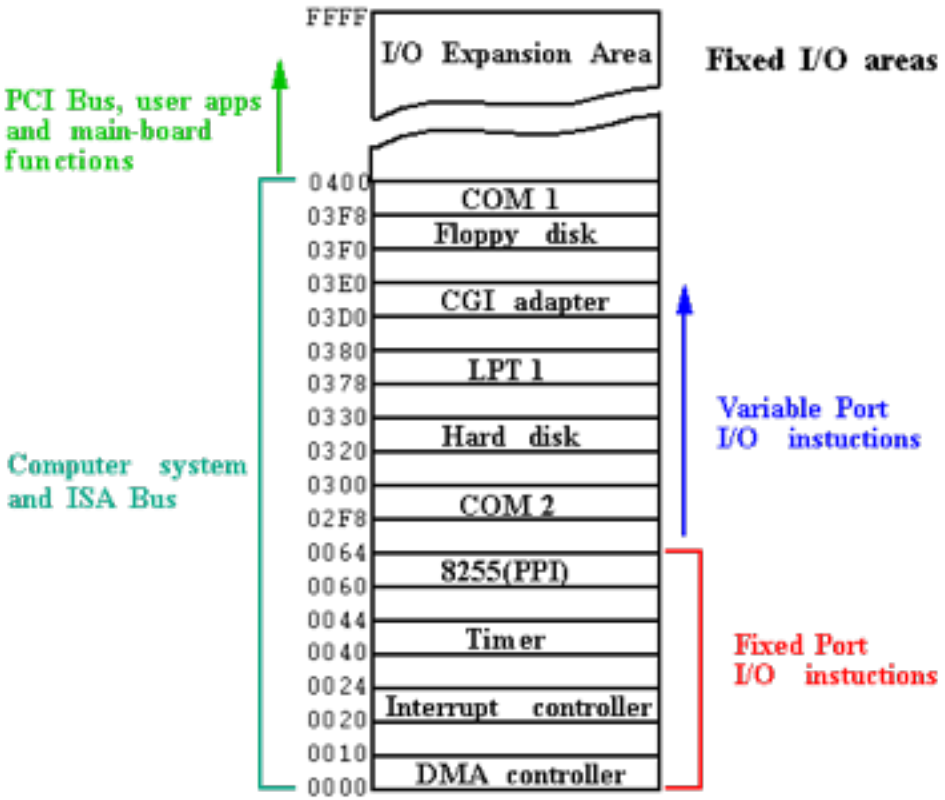
Direct Memory Access (DMA)



Bottom-up: I/O Communication

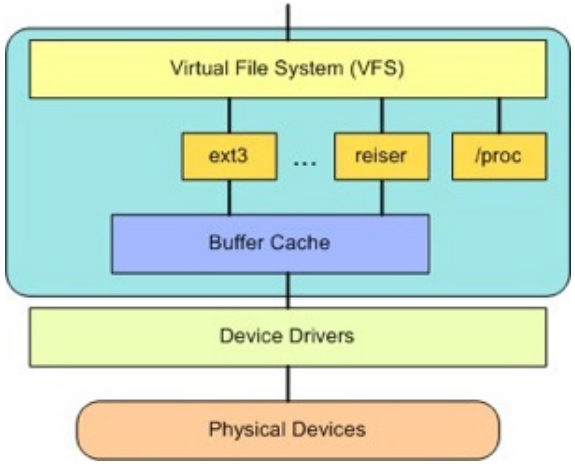
- Dedicated instructions:
  in AL, 1Ah (read 8 bits from IO port 0x1A into reg AL
  out DX, EAX (write contents of reg EAX to IO dx .
  Intel uses this model.
- Memory mapped I/O: reserve part of memory for I/O:

  in C:
  int *pmem = 0x00AA12AE;
  *pmem = 0x345; /*value*/



Bottom-up: Device Drivers

- OS goal: provide a generic, consistent and convenient way to access I/O devices. As device independent as possible.
- Lowest level software layer; Initiates I/O operations
- Interrupt handler takes care of most important actions
- Device driver takes care of the rest (scheduling after RTI)
- Device drivers can be problematic in monolithic kernels like windows (the blue screen of death), linux too. (Alternative: message passing microkernels/nanokernels)
- Kernel module commands “lsmod”, “insmod” / “rmmod”
- Kernel modules can be statically loaded in the kernel (compiled into kernel) or dynamically loaded (with insmod)
- Linux modules live happy in /lib/modules/3.10.xxx/kernel/drivers/



Example device driver

```
int init_module(void)
{
    void *ptr_error;
    struct cdev* c_dev;
    int result=0;

    /* register_chrdev */
    result=register_chrdev(my_major, "gpio_device", &fops);
    if (result < 0)
    {
    printk(KERN_INFO "Registering device failed\n");
    return result;
    }

    DEV_T = MKDEV(my_major, my_minor);

    /* class_create */
    device_class = class_create(THIS_MODULE, "gpio_device");
    if (IS_ERR(device_class))
    {
    unregister_chrdev(my_major, "gpio_device");
    printk(KERN_INFO "Class creation failed\n");
    return PTR_ERR(device_class);
    }

    /* device_create */
    ptr_error = device_create(device_class, NULL, DEV_T, NULL, "gpio_device");
    if (IS_ERR(ptr_error))
    {
    class_destroy(device_class);
    unregister_chrdev(my_major, "gpio_device");
    printk(KERN_INFO "Device creation failed\n");
    return PTR_ERR(ptr_error);
    }

    printk(KERN_INFO "Joepie, GPIO driver initialized\n");
    return SUCCESS;
}

static void __exit mod_exit(void{
    ...
}

module_init(init_module);
module_exit(mod_exit);
MODULE_LICENSE("GPL");
```

Disk I/O scheduling

- Scheduling of blocks can be done in the I/O module, but….
- The disk controller has no knowledge about certain constraints
  - application I/O constraints
  - write when cache is full
  - write ordering (taking into account that systems can crash)
- So…
- Scheduling is done in the OS as part of a kernel driver AND in the I/O module (which has a CPU and memory too…)
- Models of disk scheduling:
  - FCFS, SSTF, Elevator, Scan and C-Look are examples

Disk I/O scheduling parameters

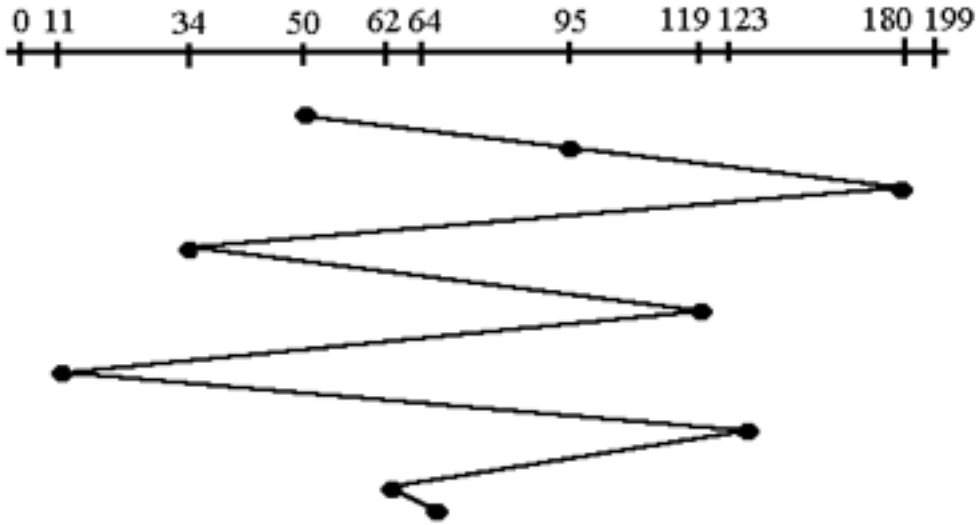
- Seek time: time to locate disk arm to a specified track where the data is to be read or write.
- Rotational latency: time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads.
- Transfer time: time to transfer the data. Depends on rotating speed of the disk and the number of bytes to be transferred.
- Disk access time

  disk access time = seek time + rotational latency + transfer time

- Disk response time: time spent by a request waiting to perform its I/O operation. Average response time is the response time of all requests

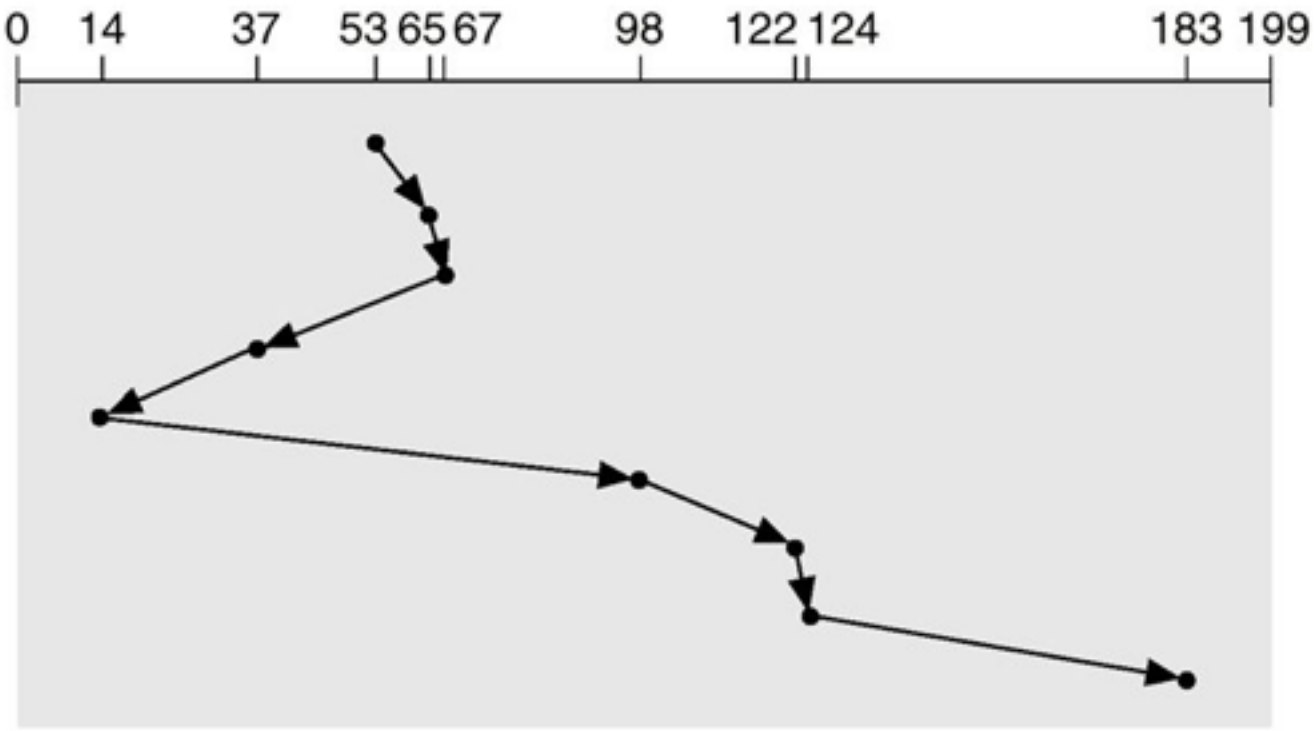
FCFS disk scheduler

- First come First served
- Track queue: 98 - 183 - 37 - 122 - 14 - 124 - 65 - 67 (start at 53)



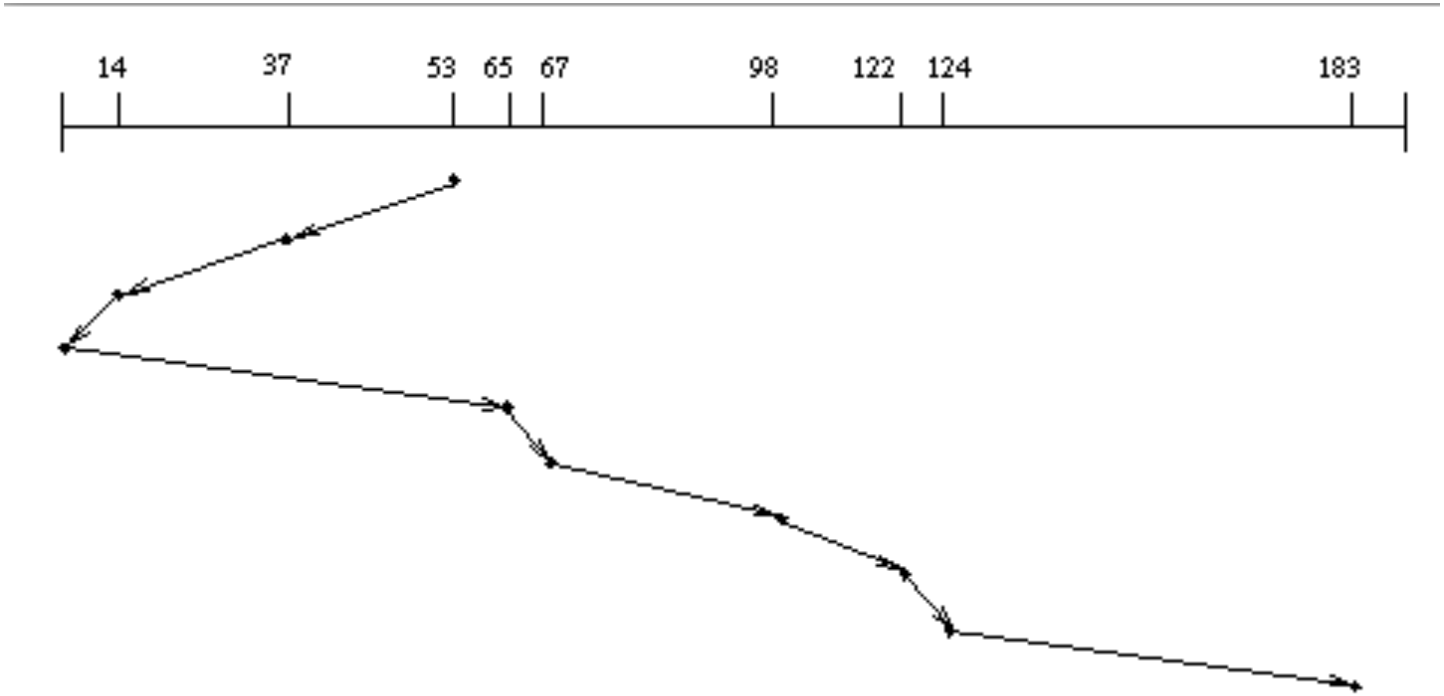
SSFT disk scheduler

- Shortest Seek Time First algorithm
- Queue: 98 - 183 - 37 - 122 - 14 - 124 - 65 - 67 (start at 53)



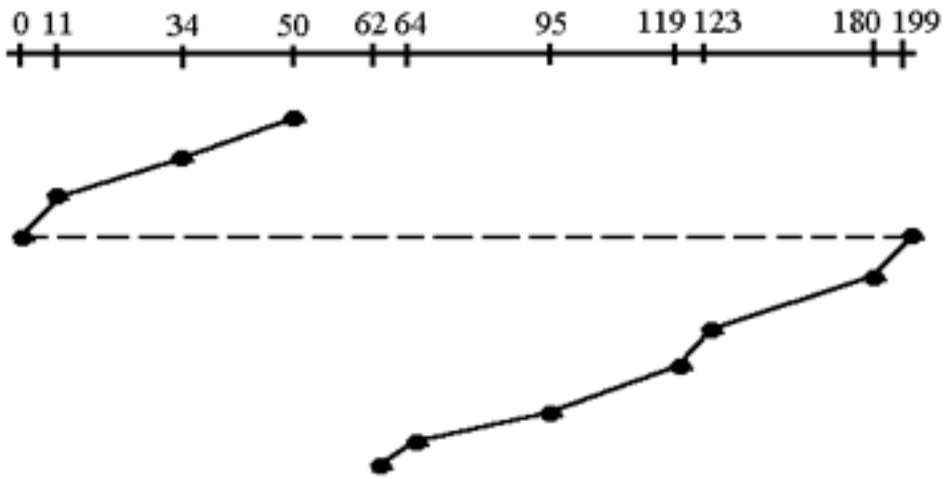
Elevator disk scheduler

- Queue: 98 - 183 - 37 - 122 - 14 - 124 - 65 - 67 (start at 53)



Circular scan /C-Look disk scheduler

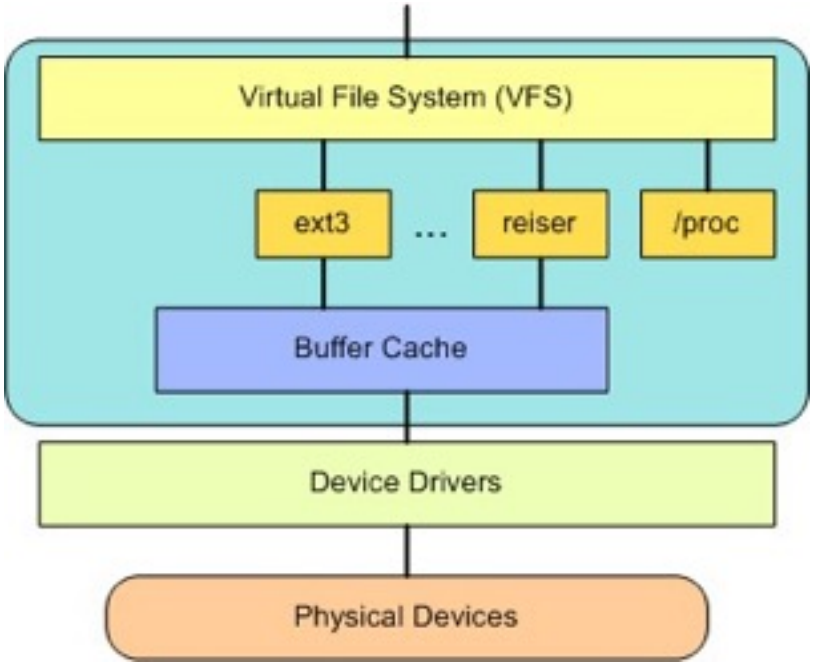
- Queue: 98 - 183 - 37 - 122 - 14 - 124 - 65 - 67 (start at 53)
- C-scan moves to 0. C-Look is only an enhanced version and does not move past the last requested track move (in this case it will move to 11 and proceeds with 183)



Devices

- Two flavors
  - block
    - data organized in fixed size blocks
    - blocks have addresses and data is therefore addressable
    - e.g. harddisks (/dev/sda)
  - character
    - deliver or accept streams of characters, no block structure
    - no “seek”, not addressable
    - read or write from/to stream
    - e.g. printers, network (/dev/eth0, /dev/ttyUSB0)

Where were we?



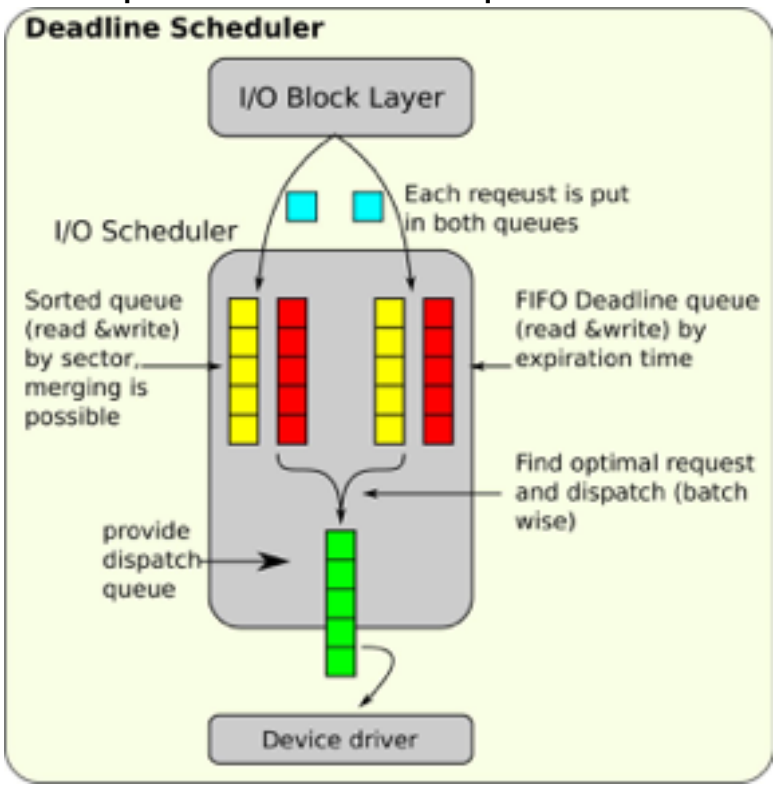
Linux I/O schedulers

- NOTE: we have a process scheduler for CPU processes AND an I/O scheduler
- I/O Scheduling in the OS (Linux):
  - Noop, CFQ (yes, counterpart of CFS), Deadline (yes, counterpart of Earliest Deadline First in RTOS)
- Kernel 2.4: “Linus elevator” (removed from 2.6)
- Kernel 2.6: earliest deadline first, “noop”, anticipatory (removed from kernel after 2.6.33), “Completely Fair Queue” (CFQ) (after 2.6.6, 2004)
- Kernel 3.x: deadline, noop and CFQ (CFQ default)
- cat /sys/block/sda/queue/scheduler

- Switch to different algorithm is possible!:
  echo “cfq”>/sys/block/sda/queue/scheduler

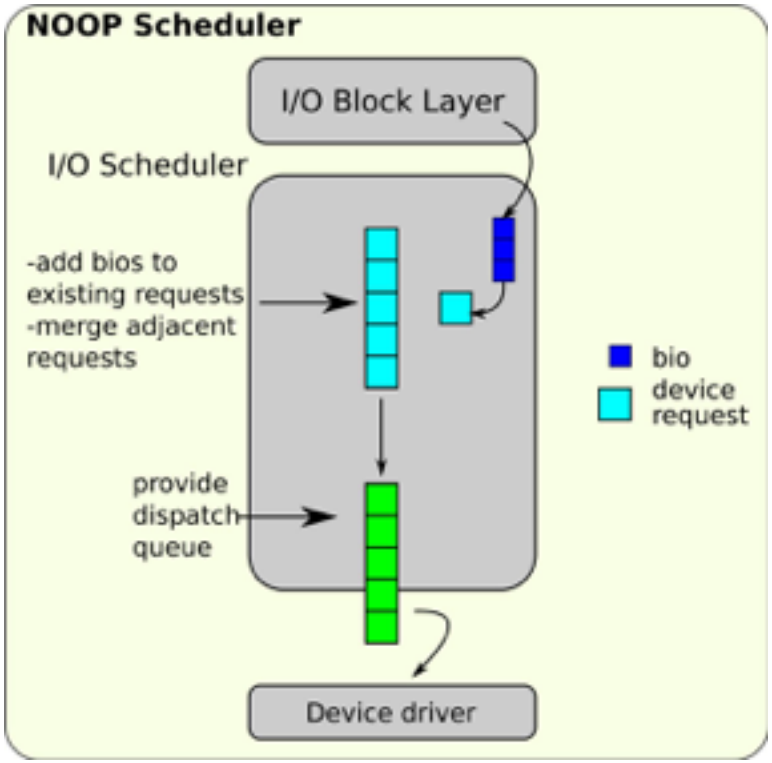
Linux “deadline” I/O scheduler

- Goal was: fix the starvation problems of the elevator
- Each request an expiration time (500 ms for reads, 5s for writes)
- 3 queues:
  - sorted queue a la the elevator
  - read/write FIFO: requests sorted chronologically
  - submit from sorted queue unless a request from head r/w FIFO’s expires



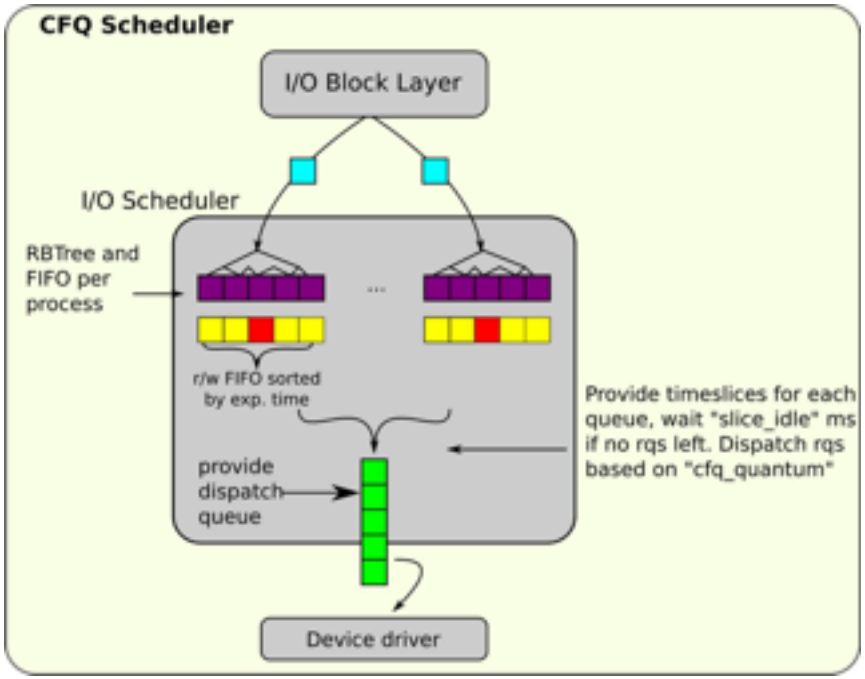
Linux “NOOP” I/O scheduler

- NOOP
  - Pronounce as No Operation
  - Simple FIFO queue. Merge with adjacent requests
  - Ok for devices who do the sorting themselves (RAID storage controllers, Network Attached Storage)
  - Ok for random access disks like flash



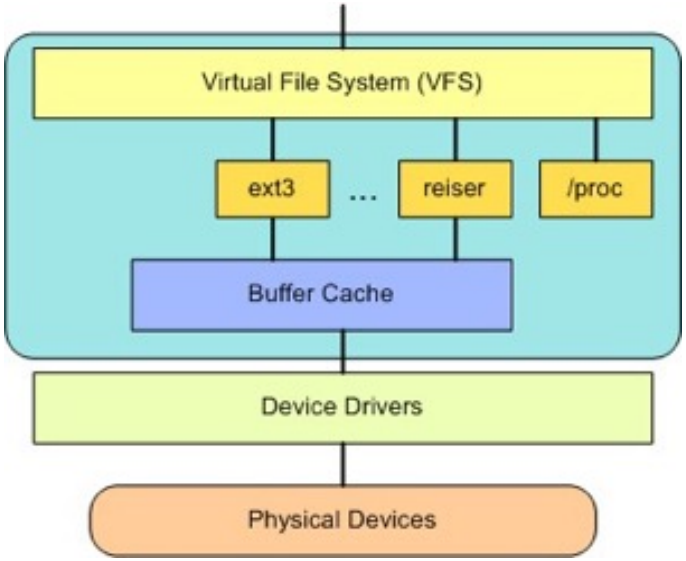
CFQ: Complete Fairness Queueing

- Current default I/O scheduler (note the default Linux process scheduler is named CFS: Completely Fair Scheduler)
- per process sorted (elevator type) queue for synchronous requests
- Fewer queues for asynchronous requests
  - Time slice for each queue. Scheduler visits each queue in a round-robin fashion.
- Priorities, backward-seek possible (with penalty calculation)



Above kernel modules: basic file system

- The basic file system takes care of I/O between multiple sources.
- Buffering and cache of data in memory
- Do nothing with contents of data blocks or file structure
- Examples: FAT, ext3, ntfs, xfs, Reiserfs…

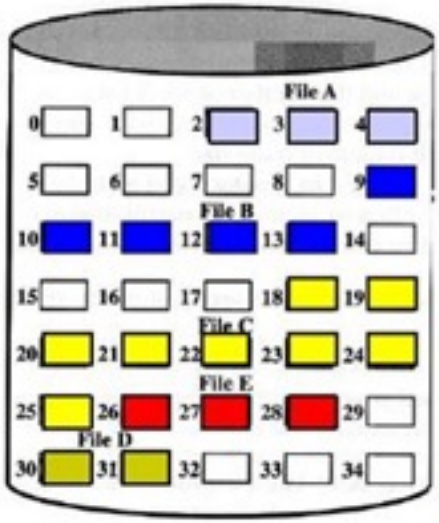


File allocation

- Each file occupies a set of adjacent blocks
- We need to know where it starts + file length
- 3 basic methods:
  - contiguous
  - linked: FAT table
  - indexed: unix FS

Contiguous allocation

- Suppose we have complete contiguous files…=> Yippy: only administration of start + length and fast localisation. Minimal seek time too.
- Uh?: what to do with file growth?
  What to do with allocation for a certain file size?
- External fragmentation: scattered blocks of free space
- Internal fragmentation: unused space in a block
- Do periodic de-fragmentation.



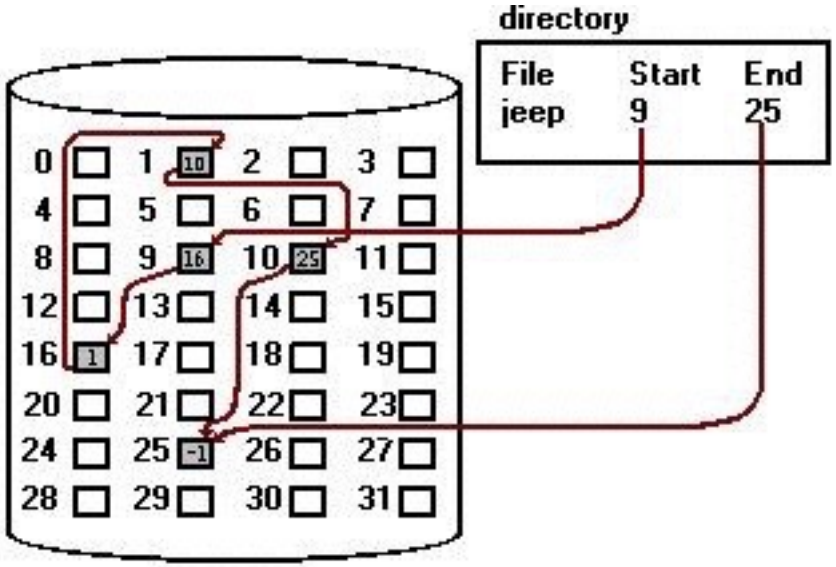
directory

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Since blocks are allocated contiguously, external fragmentation may occur. Thus, compaction may be needed.

Block allocation: linking

- Link the file's data as a linked list of disk blocks
  Directory contains pointer to first block of the file
  Each block contains a pointer to the next block
- Pro: no external fragmentation
  Pro: directory entry just stores the disk address of the first block
- Cons: nice for sequential access but not for non-sequential/random access.
  Cons: Each block uses space for the "next" pointer.

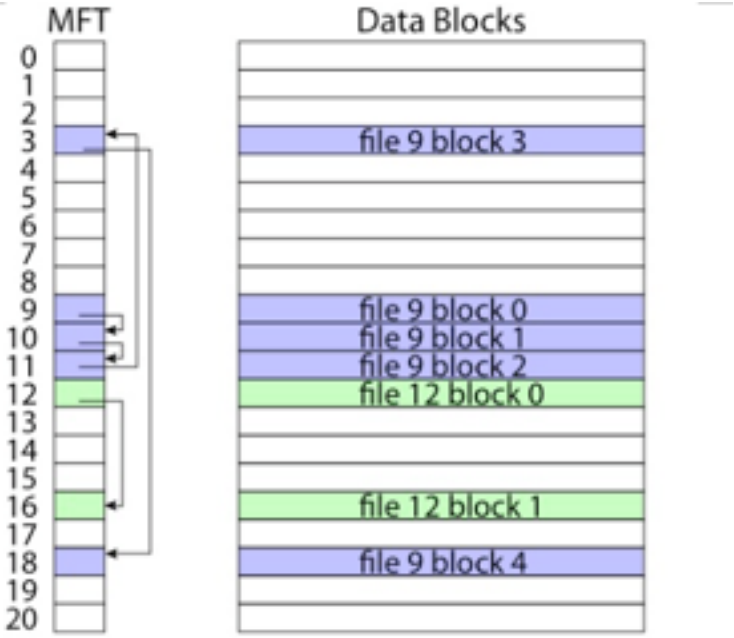


File Allocation Table (FAT)

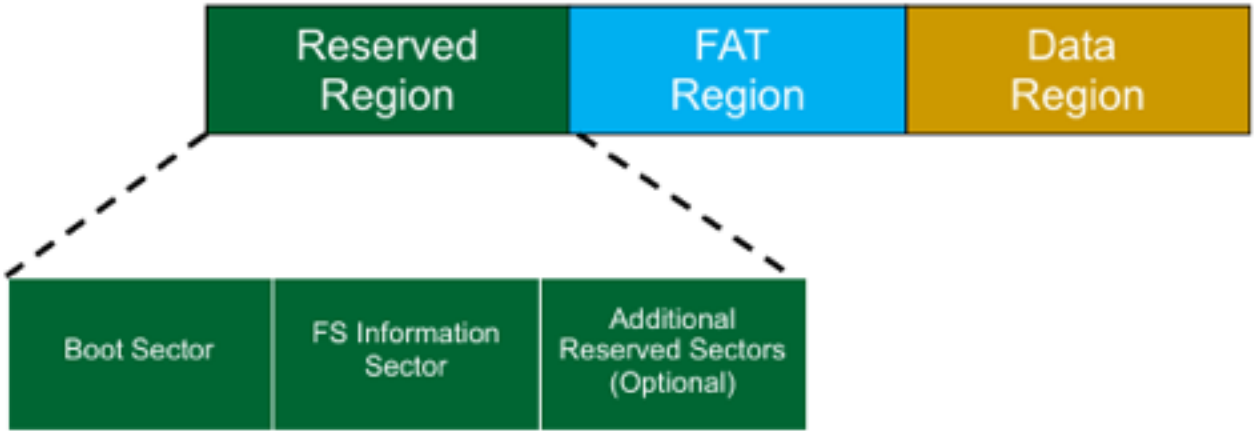
- Still in use in usb keys. Historical: FAT12 (early DOS), FAT16 (win 3.1)
  FAT32 (win95-win ME, win server2003). FAT32 has a 32 bit file alloc. table entry. Even FAT64 exists…(God, will it ever be blown to hell?)
- Table:start indicates start of file cluster (group of adjacent disk sectors). Contains an array of "next" block numbers or end of file (0xFFFF).
- FAT32 uses 4 kb blocks for volume sizes -8 GB, 16 kb for volumes - 32 GB (max)

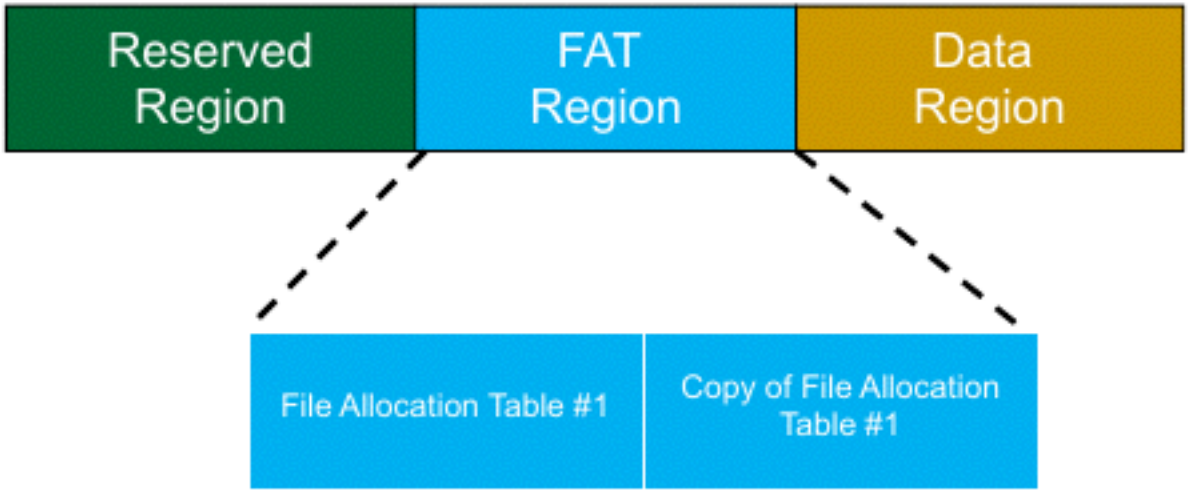
FAT-16

8 bytes	3	1	10	4	2	4
Name	Extension	Attrs	Reserved	Date and time	1st block	File size



File Allocation Table (FAT)





File Allocation Table (FAT)

- In the old days of FAT bootable disks the boot sector was located in the first 512 bytes. This is an useless section on most usb keys from boot point of view
- However it contains some important information:
  - offset 11 gives a value for the bytes / sector
  - the location of our root directory cluster at offset 44.
  - The root directory can span multiple clusters
- FAT32:
  - root directory is part of the data area
  - FAT32 keeps copy of 3 boot sectors
  - FAT32 will keep 2 copies of the FAT table

File Allocation Table (FAT)

FAT-16

8 bytes	3	1	10	4	2	4
Name	Extension	Attrs	Reserved	Date and time	1st block	File size

- Directory attributes:
  - 0x1: file is read only
  - 0x2: hidden file
  - 0x4: system file (and hidden)
  - 0x8: special: entry contains disk label
  - 0x10: entry is a subdirectory
  - 0x20:archive flag, used by back up systems. set when file modified
  - 0x40 / 0x80: not used (0)

```
Block 21 (0x0015)
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
000  43 4f 38 38 33 2d 41 32 20 20 20 28 00 00 00 00 CO883-A2   (....
010  00 00 00 00 00 00 91 9e 65 39 00 00 00 00 00 00 ........e9......
020  46 4f 4f 42 41 52 20 20 54 58 54 21 00 a3 91 9e FOOBAR  TXT!....
030  65 39 65 39 00 00 91 9e 65 39 c6 10 1a 00 00 00 e9e9....e9......
040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Example FAT

- File1: foo1.jpg 1400 bytes and File2: foo2.png 980 bytes
- File1 will occupy 3 sectors, for example 2, 3, 4.
  Root directory will contain name “foo1” (8 bytes), extension “jpg” (3 bytes),
- File2 will occupy 2 sectors, for example 5, 6. 44 bytes slack space in sector 6.
- FAT will contain:
  2: 3
  3: 4
  4: 0xFFFF (end of file)
  5: 6
  6: 0xFFFF (end of file)
  7: 0x0
  …
  Directory will contain:
  foo1 | jpg | 0 | reserved | 171231 | 2 | 1400

FAT disadvantages and advantages

- Extensive defragmentation temporarily necessary (-)
- Large volumes will have large/huge FAT tables (-)
  - 16 GB disk with 4 kb blocksize: 4 10^6 entries of 32 bits = 128 MB !
  - should be cached to get a bit performance (windows did this….)
- Information of file distributed over whole FAT table (-)
  - slow administration/access
  - Add to file not possible without the last pointer, i.e. walk down the whole tree.
  - write errors result in useless disk. Copy of FAT is stored on volume too, but reduces again access time
  - performance slows as more files are stored
- No support for links, file protection attributes etc. (-)
- Efficient use of disk space (+)
- easy to recover deleted files

FAT question

- (part of)Directory of a FAT:

filename	First block	size
fileA	5	1350
fileB	7	2200

State of the FAT is:

0 — FREE
1 — FREE
2 — -1
3 — 4
4 — -1
5 — 2
6 — FREE
7 — 3
8 — FREE

block size = 1 kb. What is the last block of fileB? How many bytes are used in this last block?

Answer FAT question

- fileB has size 2200 bytes. It will need 3 blocks of 1k (3x 1024 bytes)
- You can follow how they proceed: The directory contains a point to the first entry in the FAT no 7. The order is 7 - 3 - 4. The last entry contains -1 (0xFFFF) which means the last block.
- The blocks 7 and 3 are fully occupied: 2048 bytes. So, the last block contains 2200-2048 = 152 bytes.

FAT question

- A filesystem for a 128 GB (2^37 bytes) disk. blocksize 8 kb (2^13 bytes). Suppose we choose a FAT system, what is the smallest amount of memory that could be used for the FAT, i.e. size of the FAT?

Answer FAT question

- A disk of 128 GB with blocks of 8kb needs: 2^37/2^13 entries = 2^24
- To refer each entry we need a pointer space of 24 bits = 3 bytes.
- So the FAT size is 2^24 *3 bytes = 48 MByte

Disk scheduler question

- FCFS disk scheduler. Arm starts at location 100. Max. position=200
- We request locations 55 - 58 - 39 - 18 - 90 - 160 - 150 - 38 - 184
- How many moves are made?
- Compare with an elevator scheduler.

Answer disk scheduler

- The FCFS is a simple one: we move all over the place without any ordering, so we move from 100 to 55 = 45 positions etc. 498 positions in total.
- For the SSTF scheduler we need to sort the requests:
  100-90-58-55-39-18-150-160-184. Total movements = 248
- For the elevator (assume we move up first)
  100-150-160-184-(and down) 90-58-55-39-38-18. Total moves = 250