

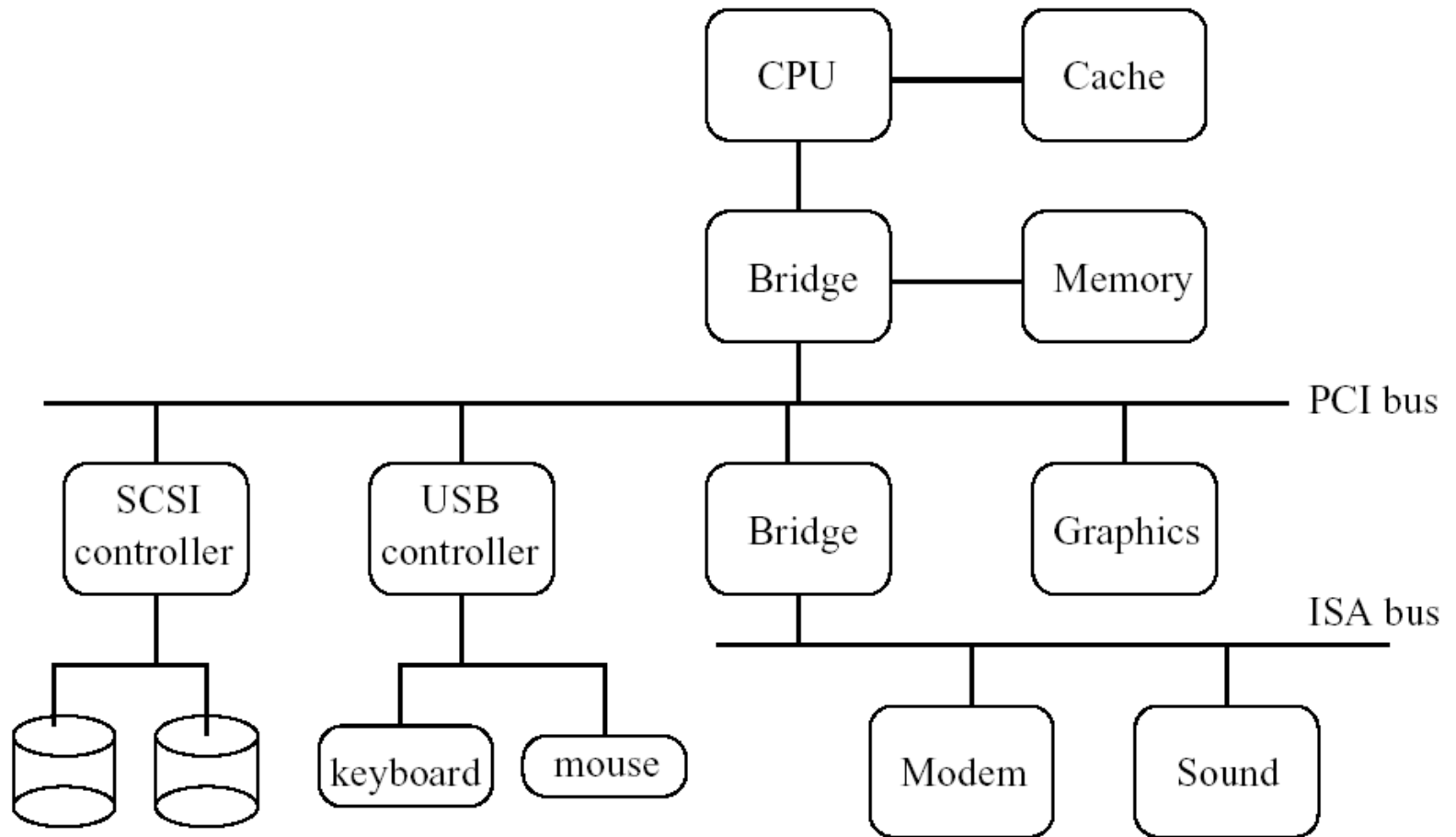
Devices and Device Controllers

A computer system contains a multitude of I/O devices and their respective controllers:

- network card
- graphics adapter
- disk controller
- DVD-ROM controller
- serial port
- USB
- sound card
- ...

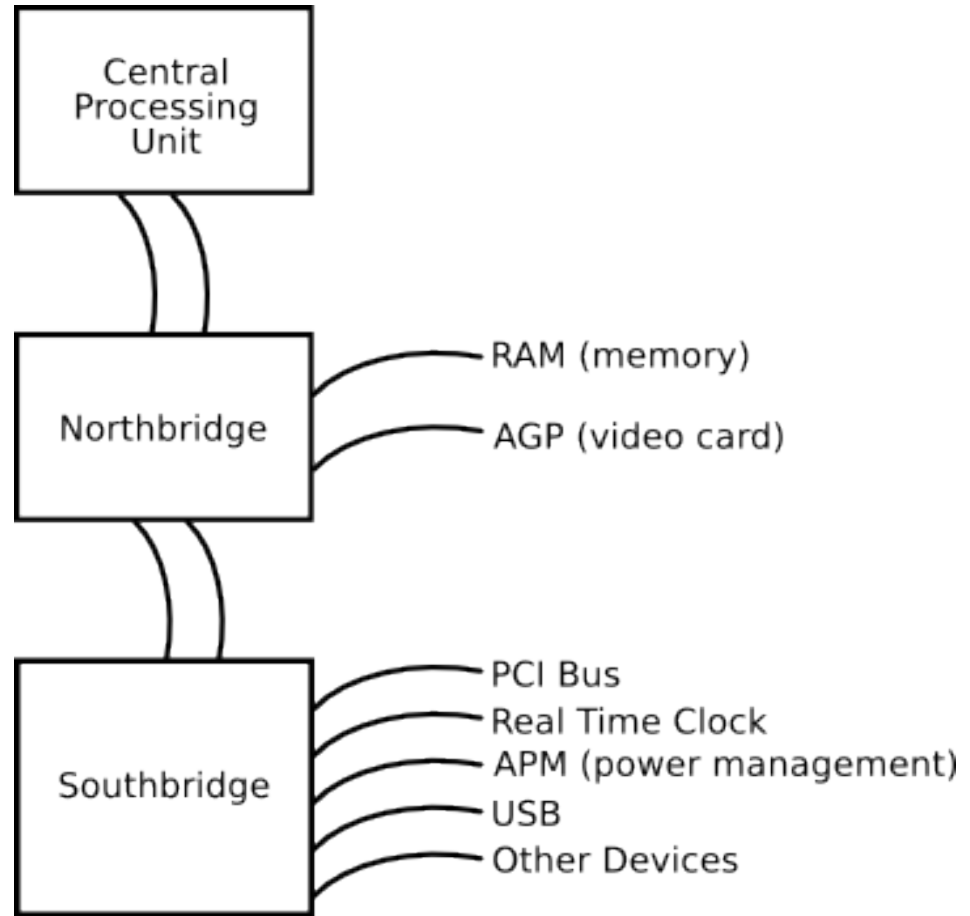
Bus Architecture

All devices in a computer are connected through a complicated bus system. Bus architecture 10 years ago:



Bus Architecture

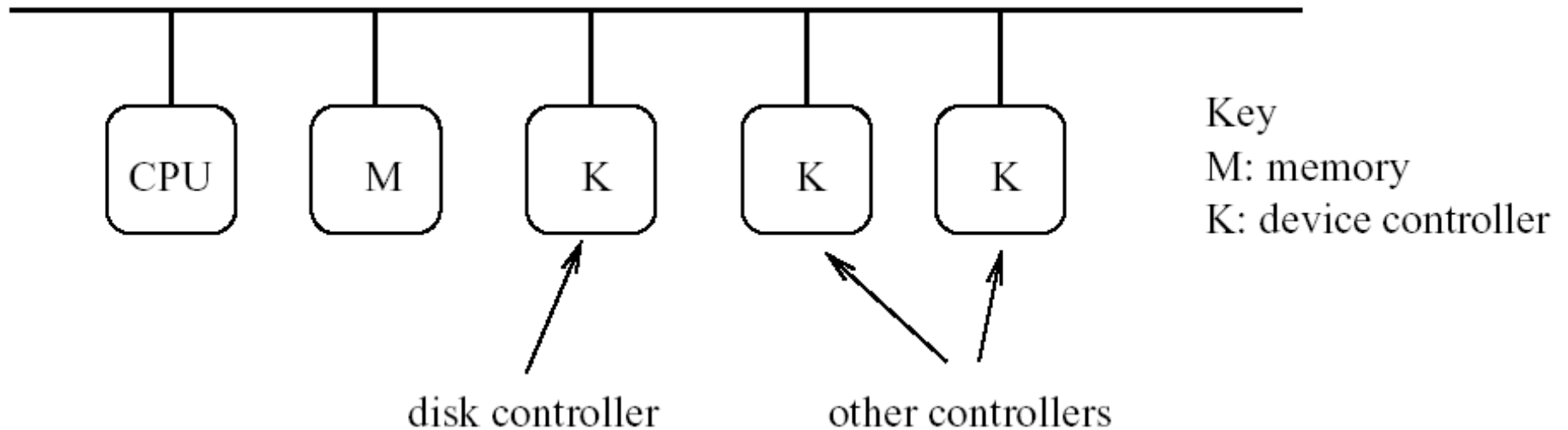
Bus architecture these days:



(<http://en.wikipedia.org/wiki/Image:Northsouthbridge.png>)

Simplified Bus Architecture

A common abstraction is to forget about the individual sub-systems in the bus architecture and to just assume that devices are connected with each other *somehow*.



Interacting with Device Controllers

Each device controller maintains a set of device registers:

- command registers (write-only);
- status registers (read-only);
- data registers (read/write).

The CPU can access those registers through special I/O instructions (`in/out` in Intel assembler) or via memory mapping. Memory mapping is handled by the memory controller and is usually faster than explicit I/O instructions.

When interacting with a device controller, the CPU can wait for a response by *polling* the status register(s), i.e., by periodically checking whether the status of the device has changed.

Interacting with Device Controllers

Problem with polling: The CPU is busy waiting for some event to happen. CPU utilization will be low.

Solution: **Interrupts.**

Used by devices for asynchronous event notification.

When an interrupt is fired, the CPU jumps to a predefined position in the kernel's address space and executes an interrupt handler.

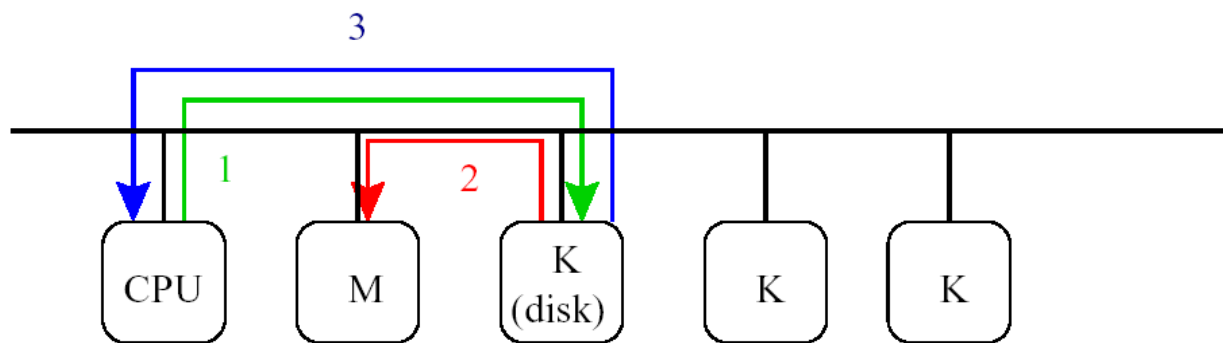
When an interrupt occurs, the CPU can start reading data from the device controller's data registers.

Direct Memory Access

CPU utilization can be improved even further by allowing a device controller to transfer data from/to main memory without any involvement of the CPU.

This is called *direct memory access* and is used by all modern hard drives (but also sound controllers, network cards, ...).

The CPU initiates the data transfer, but can do other things while the transfer is taking place. CPU utilization is increased.



1. CPU issues DMA request to controller
2. controller directs data transfer
3. controller interrupts CPU

Device Drivers

Interaction with a device controller is managed through a device driver.

Device drivers are part of the operating system, but not necessarily part of the OS kernel.

The operating system provides a simplified view of the device to user applications (e.g., character devices vs. block devices in UNIX).

In some operating systems (e.g., Linux), devices are also accessible through the `/dev` file system.

In some cases, the operating system buffers data that are transferred between a device and a userspace program (disk cache, network buffer). This usually increases performance, but not always.

Hibernation

Hibernation (*Suspend-to-RAM* or *Suspend-to-Disk*) requires the operating system to save the current state of the computer and save it somewhere (RAM or disk).

This requires that

1. Each device controller provides a mechanism through which the entire current state of the device can be read by the CPU, stored somewhere, and restored when the system wakes up.

OR

2. The operating system needs to know about all the internal details of each hardware device and obtain all necessary information about the internal state piece-by-piece.

This is why hibernation usually does not work under Linux.

Monolithic Kernels vs. Micro-Kernels

In many operating systems (Linux, Windows, ...), device drivers are part of the operating system kernel with complete access to the entire system.

This can sometimes be convenient for the device driver developer, but may lead to severe security problems.

What if the network card device driver has a security hole that can be exploited by sending a carefully engineered network packet to the card? Then the whole kernel will be unsecure.

Micro-Kernel: MINIX (Andrew Tanenbaum et al.). A micro-kernel only contains the core components, such as memory management and CPU scheduler. All device drivers are realized through separate processes with normal userspace rights (and some additional task-specific privileges).

Logical View of a Disk Drive

A hard disk may be viewed as an array of numbered blocks (called *sectors*).

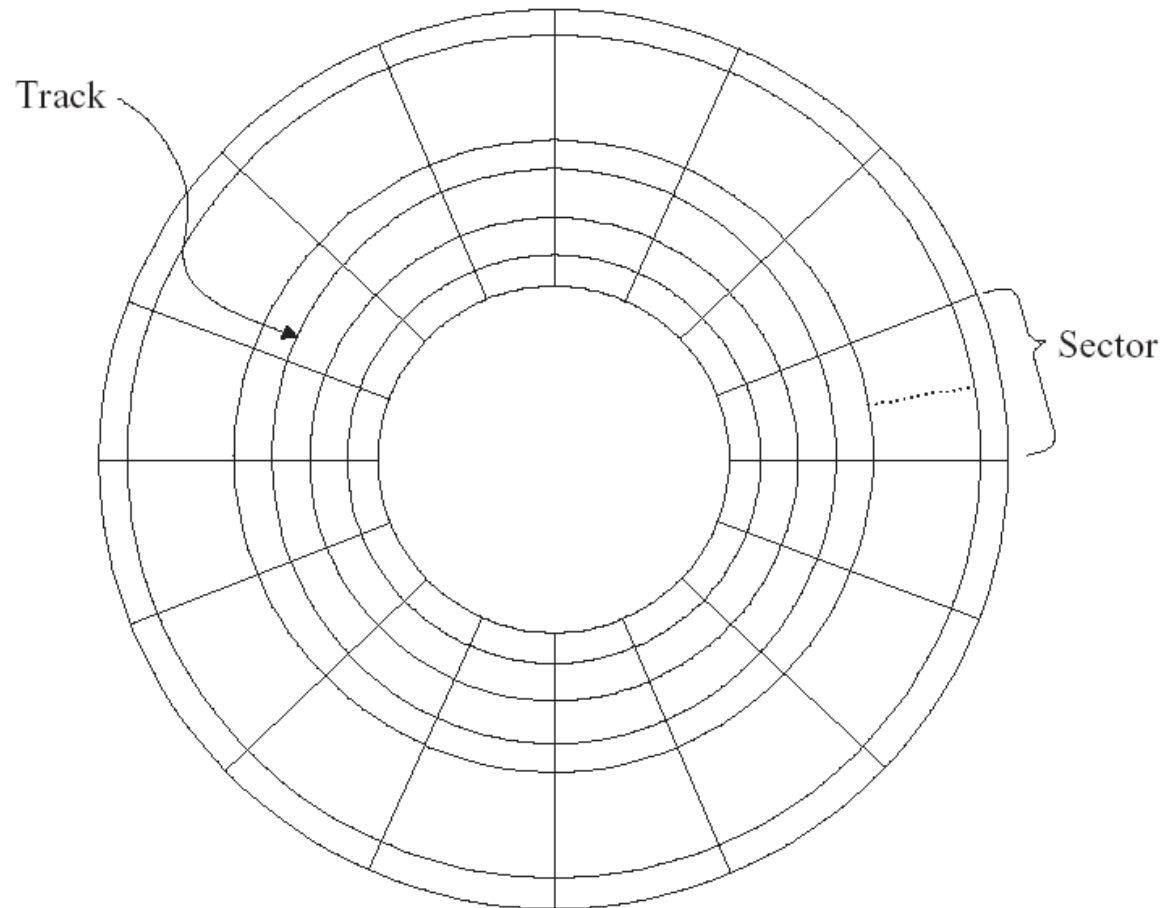
Each block has the same size, usually 512 bytes (CDROM: 2048 bytes; Nachos: 128 bytes, same as page size).

Blocks are the basic units of transfer between disk and main memory. Usually, multiple blocks can be transferred in a single operation, but data transfers of less than a single block are impossible.

Storage on hard disks is non-volatile. Data persists after the device is unplugged (as opposed to main memory).

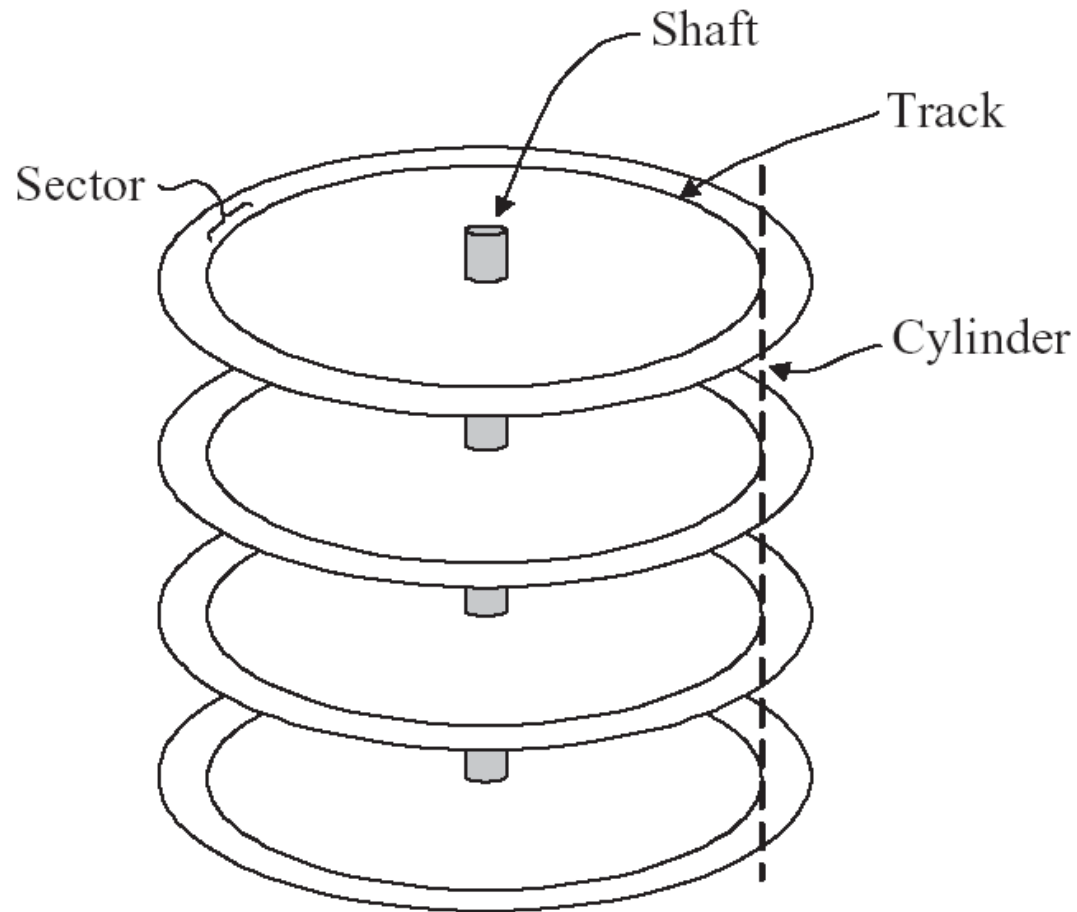
A Disk Platter's Surface

A hard disk contains one or more *platters*, rotating discs that have a read/write head that can be used to read/write data from/to the disk.

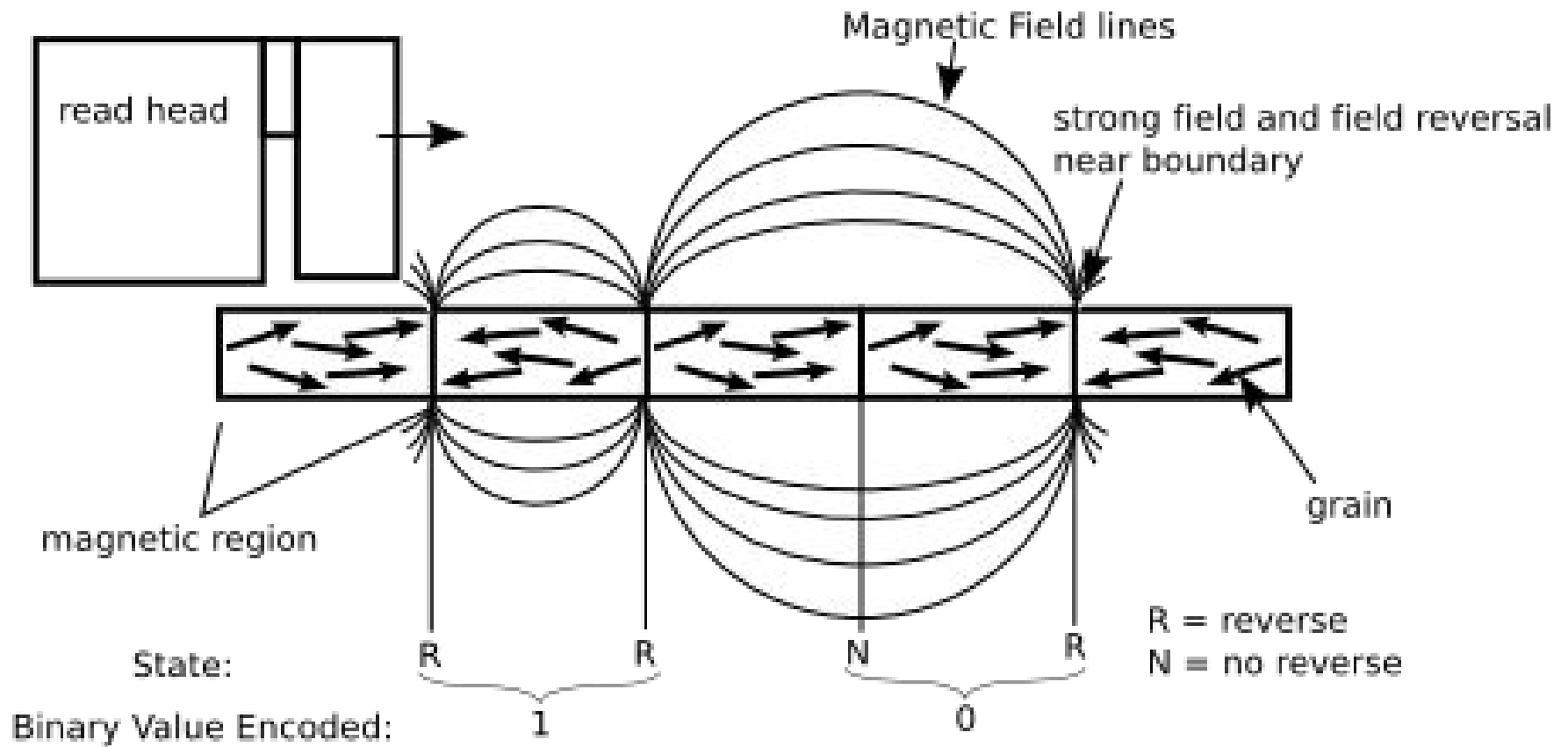


Multiple Platters

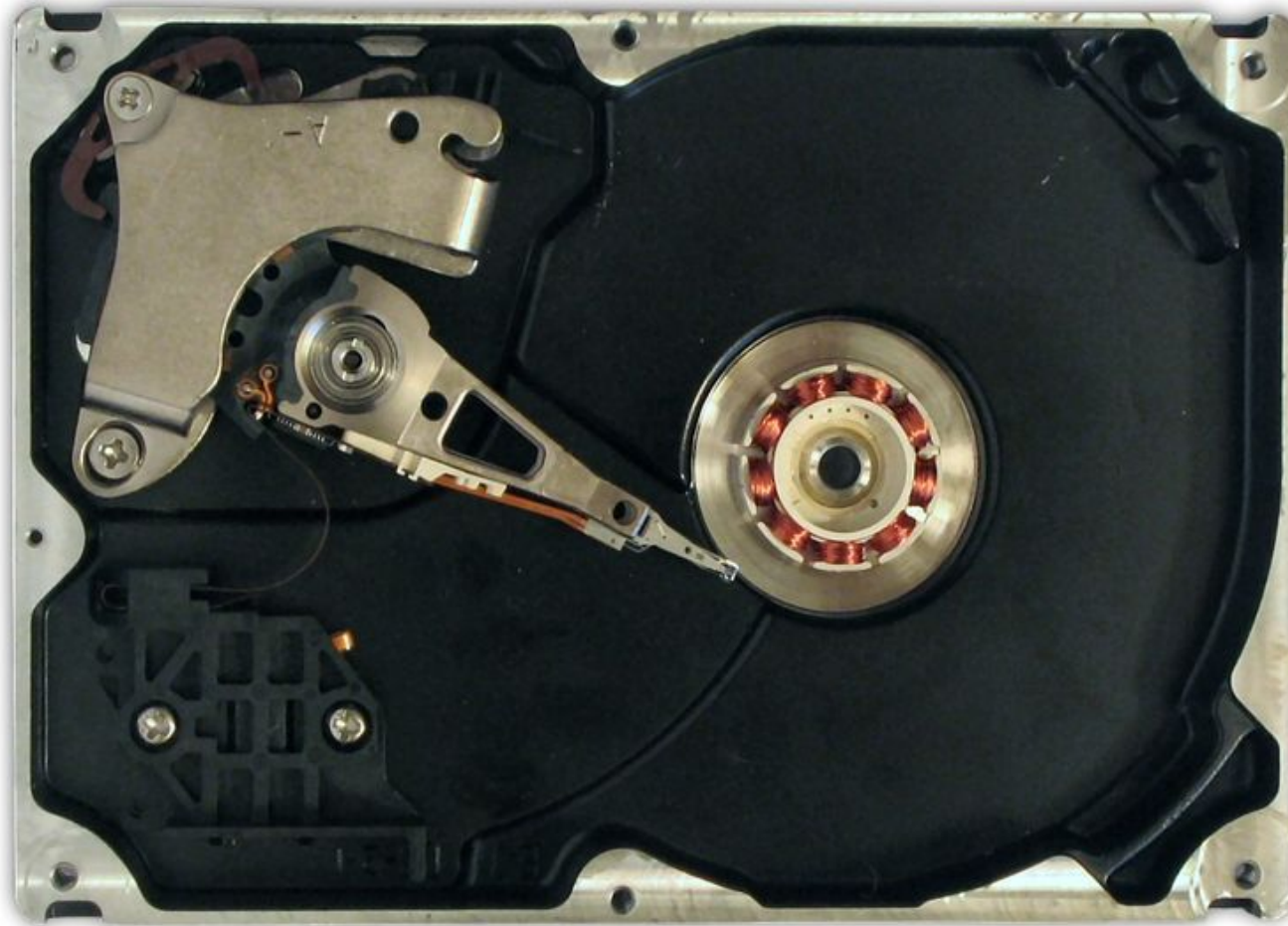
A hard disk consists of multiple platters. Platters are connected by a shaft. Tracks on different platters form *cylinders*.



Data Encoding



Something Visual



http://en.wikipedia.org/wiki/Image:Hard_disk_dismantled.jpg

Something Visual



<http://en.wikipedia.org/wiki/Image:HDDspin.JPG>

Something Visual



http://en.wikipedia.org/wiki/Image:IBM_old_hdd_corrected.jpg

Disk Transfer Costs

Hard disks are not true random-access storage devices.
Transferring data between main memory and disk involves:

Seek time – Wait until the hard drive has positioned the read/write head(s) above the appropriate track or cylinder.

Rotational latency – Wait until the sector that we want to read from is directly below the read/write head.

Transfer time – The time it takes to perform the actual operation (wait while the requested sectors spin past the r/w head).

The total *request service time* is the sum of all three components listed above:

$$t_{\text{service}} = t_{\text{seek}} + t_{\text{rotation}} + t_{\text{transfer}}$$

Some Examples

Seagate Barracuda 7200.10:

750 GB storage; 16 MB cache; 7200 rpm; 8 heads; 4 platters;
512 bytes per sector; read seek:

Weight: 720g

Western Digital Caviar SE16:

500 GB storage; 16 MB cache; 7200 rpm; read seek: 8.9 ms;
write seek: 10.9 ms; track-to-track: 2 ms; full stroke: 21 ms

Western Digital Raptor X:

150 GB storage; 16 MB cache; 10000 rpm; read seek: 4.6 ms;
write seek: 5.2 ms; full stroke: 10.2 ms

Some Examples

Seagate Barracuda 7200.10:

750 GB storage; 16 MB cache; 7200 rpm; 8 heads; 4 platters;
512 bytes per sector; read seek:

Weight: 720g

Western Digital Caviar SE16:

500 GB storage; 16 MB cache; 7200 rpm; read seek: 8.9 ms;
write seek: 10.9 ms; track-to-track: 2 ms; full stroke: 21 ms

Western Digital Raptor X:

150 GB storage; 16 MB cache; 10000 rpm; read seek: 4.6 ms;
write seek: 5.2 ms; full stroke: 10.2 ms

Some Numbers...

Rotational Latency

Transfer time

Seek time

Disk Head Scheduling

Regardless of the actual formula used to compute the seek latency, it is clear that disk seeks take a long time compared to sequential reads.

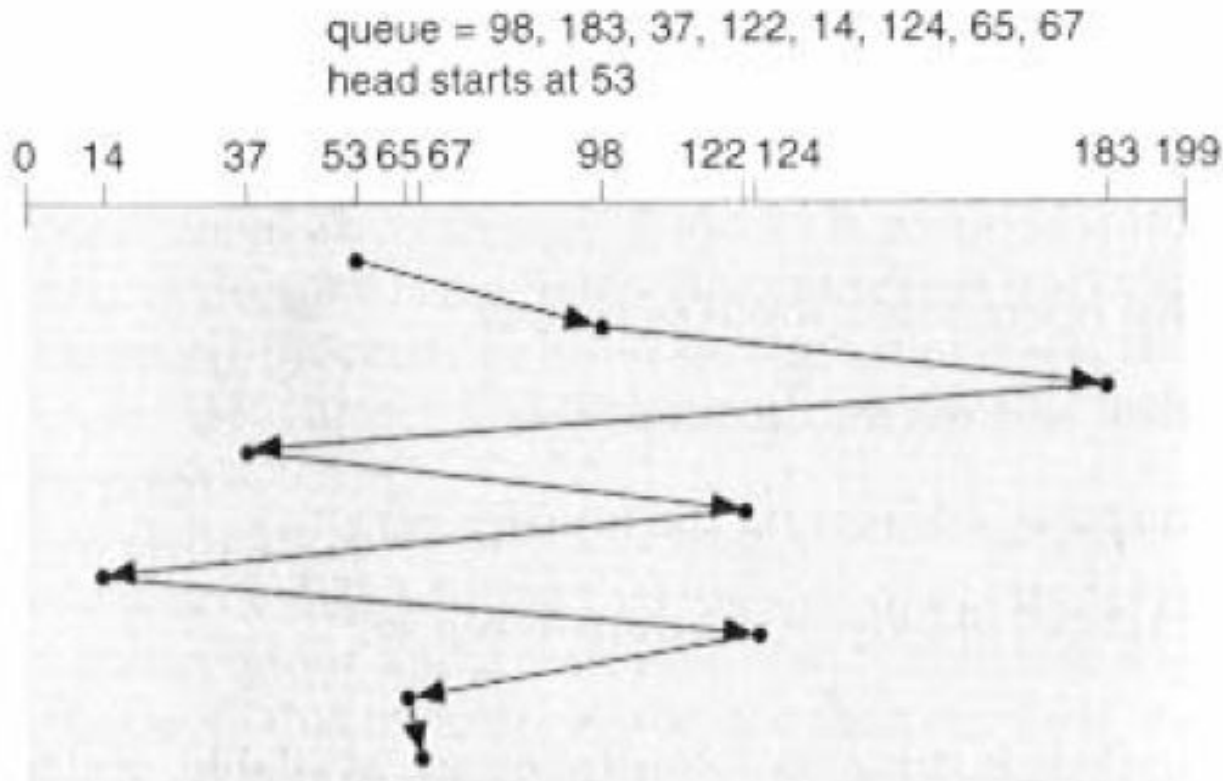
The goal of disk head scheduling: Reduce seek times by controlling the order in which requests are served.

Disk head scheduling may be performed by the OS, by the disk controller (e.g., NCQ), or both.

Disk head scheduling takes a list of outstanding requests (read or write) and re-orders them.

FCFS Disk Head Scheduling

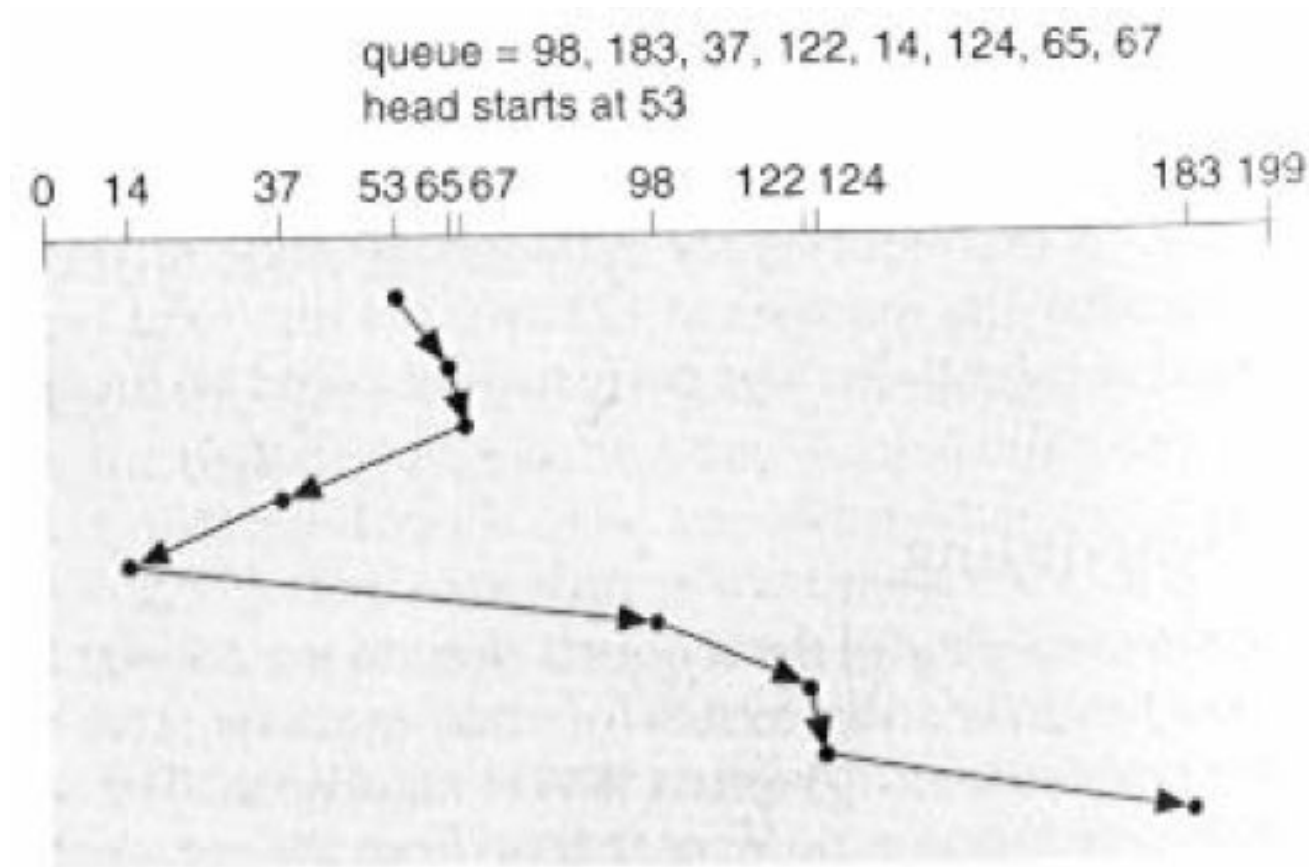
FCFS: Handle requests in the order in which they arrive.



Properties?

Shortest Seek Time First

SSTF: Choose closest request (greedy).



Properties?

SCAN and LOOK

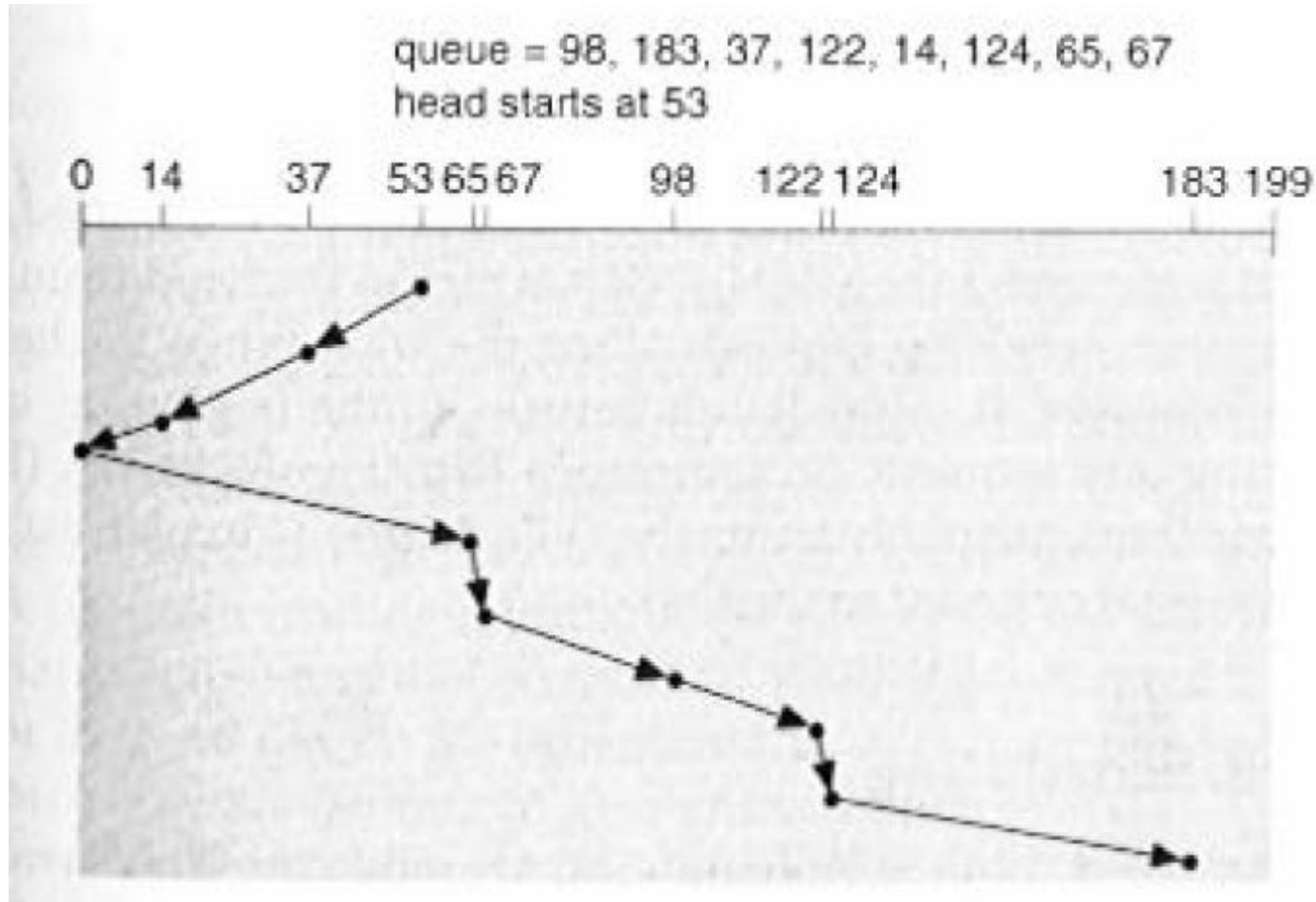
In the SCAN algorithm, the disk head moves from one end of the disk to the other end, serving all requests it encounters on the way.

When it reaches the end of the disk, it reverses its direction, and the whole process is repeated.

Reduces average seek time without causing starvation.

The LOOK strategy is a variant of SCAN. With LOOK, the disk head reverses as soon as there are no more requests in front of it.

SCAN Example



C-SCAN and C-LOOK

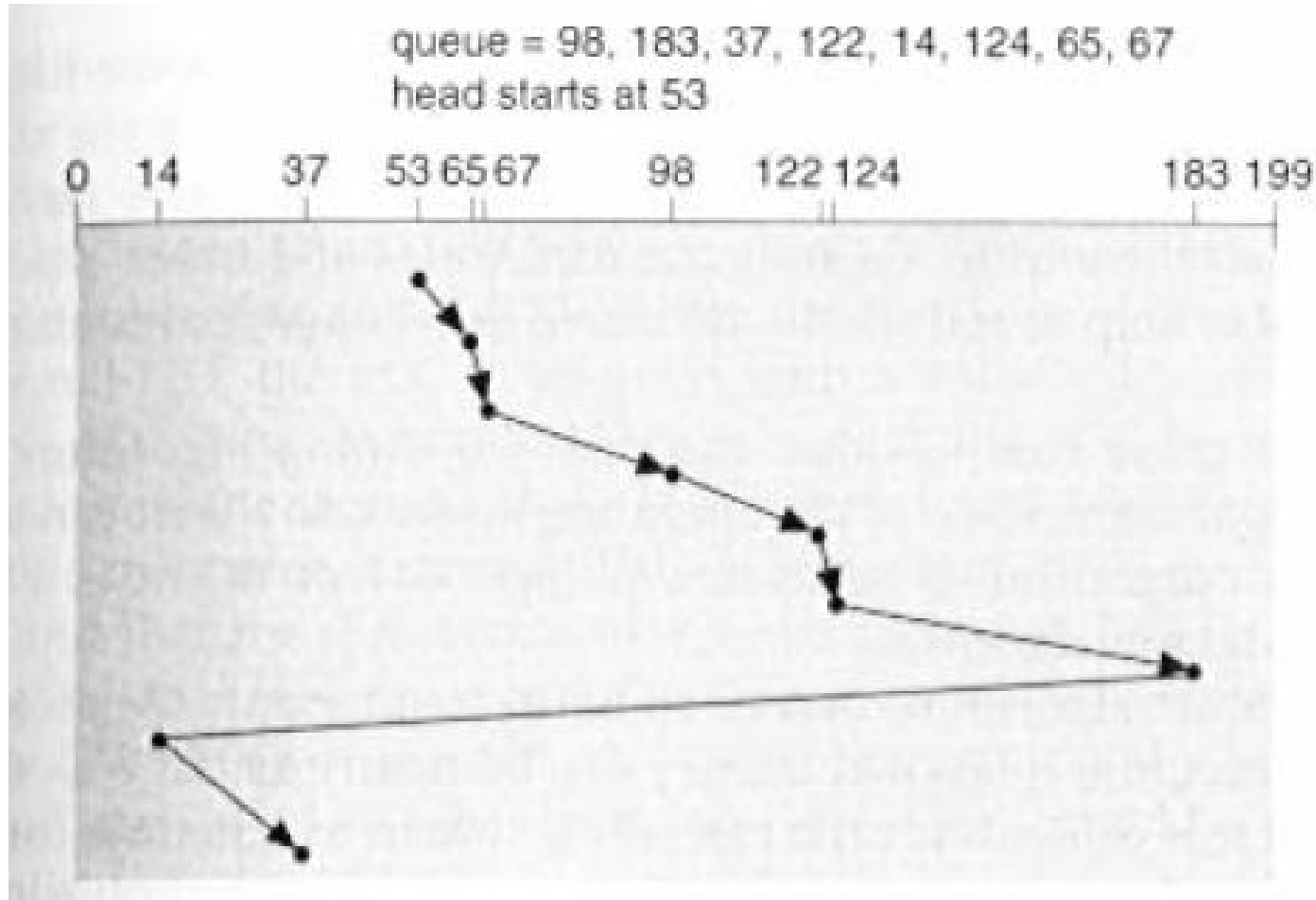
SCAN and LOOK are unfair against tracks that are close to the edge of the hard drive (inner or outer).

C-SCAN is a variant of SCAN that always processes requests in the same direction and seeks back to the very beginning when it reaches the end of the disk.

C-SCAN = Circular SCAN. Think of the tracks on a disk as a circular list ($N \bmod \text{TRACK_COUNT}$).

C-LOOK is the circular variant of the LOOK strategy.

C-LOOK Example



RAID: redundant array of independent disks (originally: redundant array of inexpensive disks).

Basic idea: Combine multiple hard drives into bigger units. Use them together and hope that makes them faster or more reliable.

RAID can be implemented in the device driver (i.e., OS kernel) or in the hard drive controller.

Different RAID levels lead to different levels of performance or reliability.

RAID-1: Mirroring

In RAID-1, a set of N disks is used to store N exact copies of the same data.

Advantages:

- Sequential read operations can be performed N times as fast as on a single disk.
- If $N-1$ disks fail, the data are still accessible.

Disadvantages:

- Disk seeks take a little longer (max. of all individual seeks).
- Write operations take a little longer.

RAID-0: Striping

In RAID-0, a set of N disks is used to store N times as many data as on a single disk.

Data are arranged in a striped fashion:

	Block 1	Block 2	Block 3	Block 4	Block 5	...
Disk 1	X			X		
Disk 2		X			X	
Disk 3			X			

Advantages: N-fold storage; N-fold transfer speed.

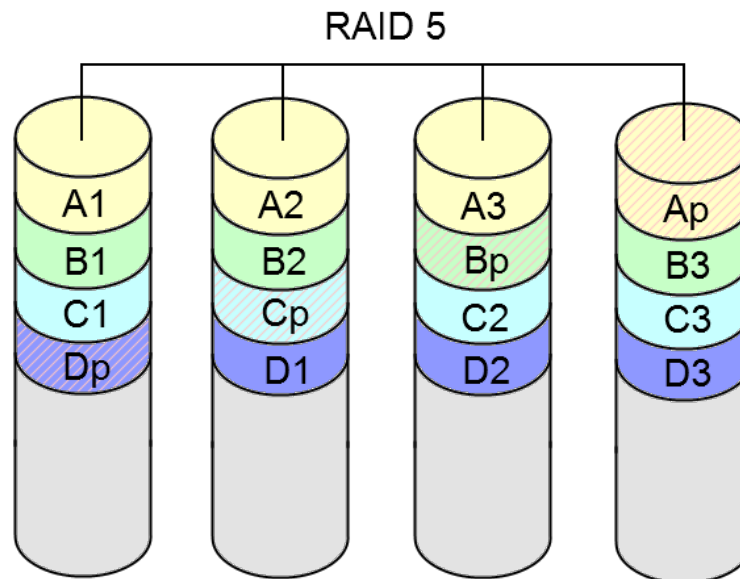
Disadvantages: Less reliable. If one disk fails, all data are useless.

RAID-5: Striping with Parity

RAID-5 is similar to RAID-0. However, N disks only store N-1 times the data that fits onto a single disk.

For every N-1 blocks (spread across N-1 disks), there is a parity block on the N-th disk (XOR of the data on disks 1..N-1).

Advantage: No data loss with 1 disk failure; N-fold transfer speed.



<http://en.wikipedia.org/wiki/Image:RAID5.png>

Nested RAID Levels

RAIDs may be nested:

- RAID 0+1 (“RAID 01”): A RAID-1, where each component is in fact a RAID-0.
- RAID 1+0 (“RAID 10”): A RAID-0, where each component is a RAID-1.
- RAID 1+0+0 (“RAID 100”): A RAID-0, where each component is a RAID-0, where each component is a RAID-1.
- RAID 5+0 (“RAID 50”): A RAID-0 on top of a bunch of RAID-5's.

The limit is your imagination...