Input/Output

Today

- Principles of I/O hardware & software
- I/O software layers
- Disks

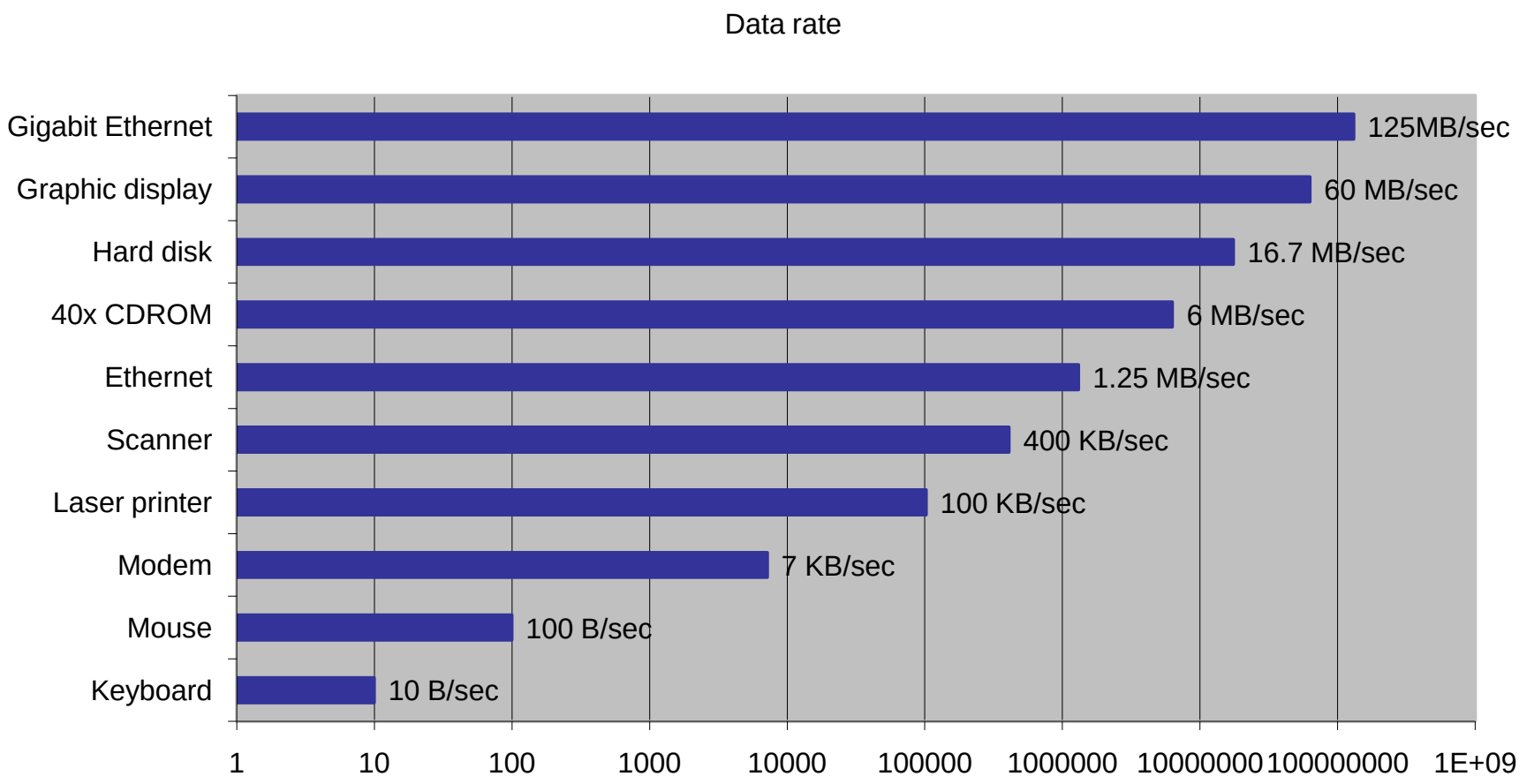
Next

- Protection & Security

Operating Systems and I/O

- Two key operating system goals
  - Control I/O devices
  - Provide a simple, easy-to-use, interface to devices
- Problem – large variety
  - Data rates
  - Applications – what the device is used for
  - Complexity of control – a printer (simple) or a disk
  - Units of transfer – streams of bytes or larger blocks
  - Data representation – character codes, parity
  - Error condition – nature of errors, how they are reported, their consequences, …
- Makes a uniform & consistent approach difficult to get

Typical I/O Devices Data Rates



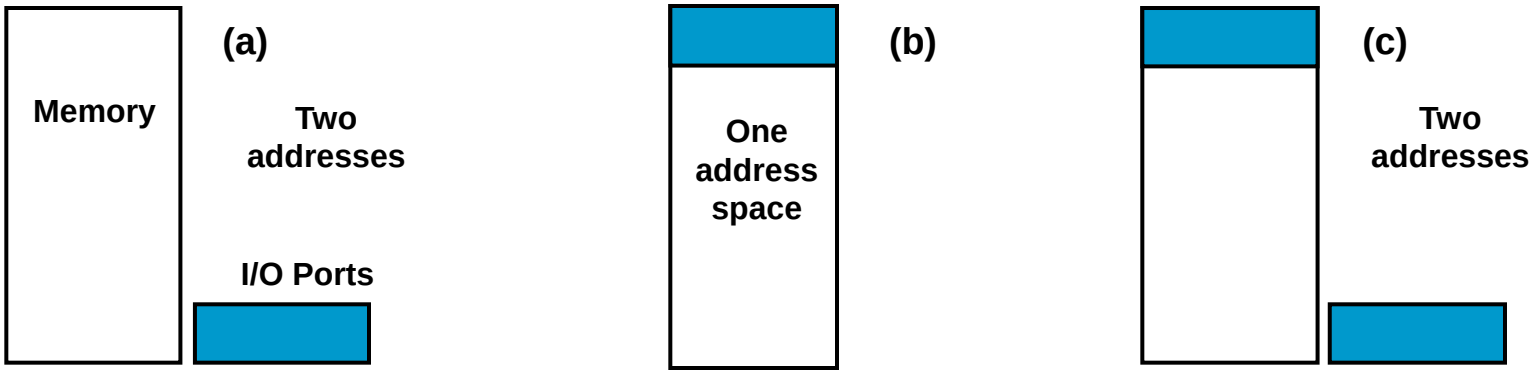
I/O Hardware - I/O devices

- I/O devices – roughly divided as
  - Block devices – stored info in fixed-size blocks; you can read/write each block independently (e.g. disk)
  - Character devices – I/O stream of characters (e.g. printers)
  - Of course, some devices don't fit in here (e.g. clocks)
- I/O devices components
  - Device itself - mechanical component
  - Device controller - electronic component
- Controller
  - Maybe more than one device per controller
  - Converts serial bit stream to block of bytes
  - Performs error correction as necessary
  - Makes data available in main memory

I/O Controller & CPU Communication

- Device controllers have
  - A few registers for communication with CPU
    - Data-in, data-out, status, control, …
  - A data buffer that OS can read/write (e.g. video RAM)
- How does the CPU use that?
  - Separate I/O and memory space, each control register assigned an I/O port (a) – IBM 360

    ```
    IN REG,PORT
    ```
  - Memory-mapped I/O – first in PDP-11 (b)
  - Hybrid – Pentium (c) (graphic controller is a good example)

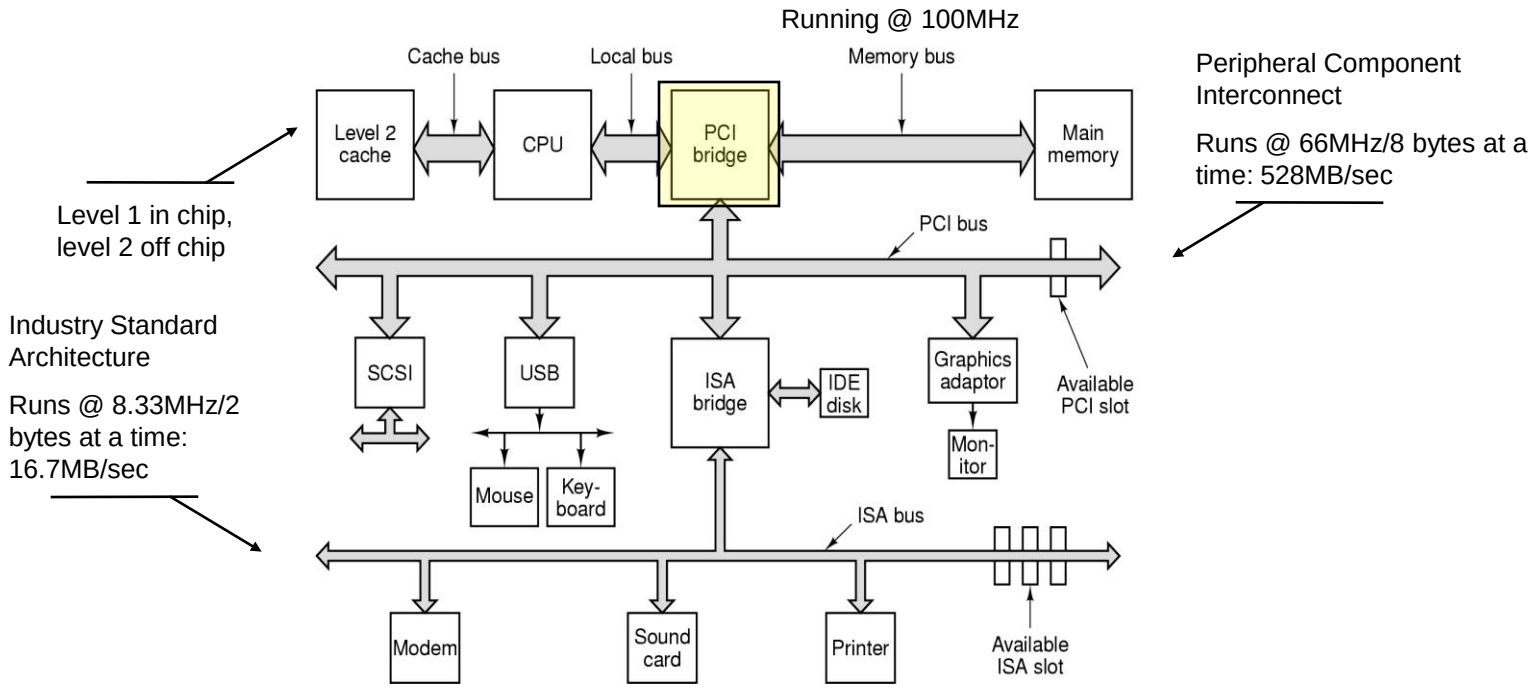


Memory-mapped I/O

- Pros:
  - No special instructions needed
  - No special protection mechanism needed
  - Driver can be entirely written in C (how do you do IN or OUT in C?)
- Cons:
  - What do you do with caching? Disable it in a per-page basis
  - Only one AS, so all must check all mem. references “is it for me?”
    - Easy with single bus (a) but harder with dual-bus (b) arch
    - Possible solutions
      - Send all references to memory first
      - Snoop in the memory bus
      - Filter addresses in the PCI bridge (preloaded with range registers at boot time)

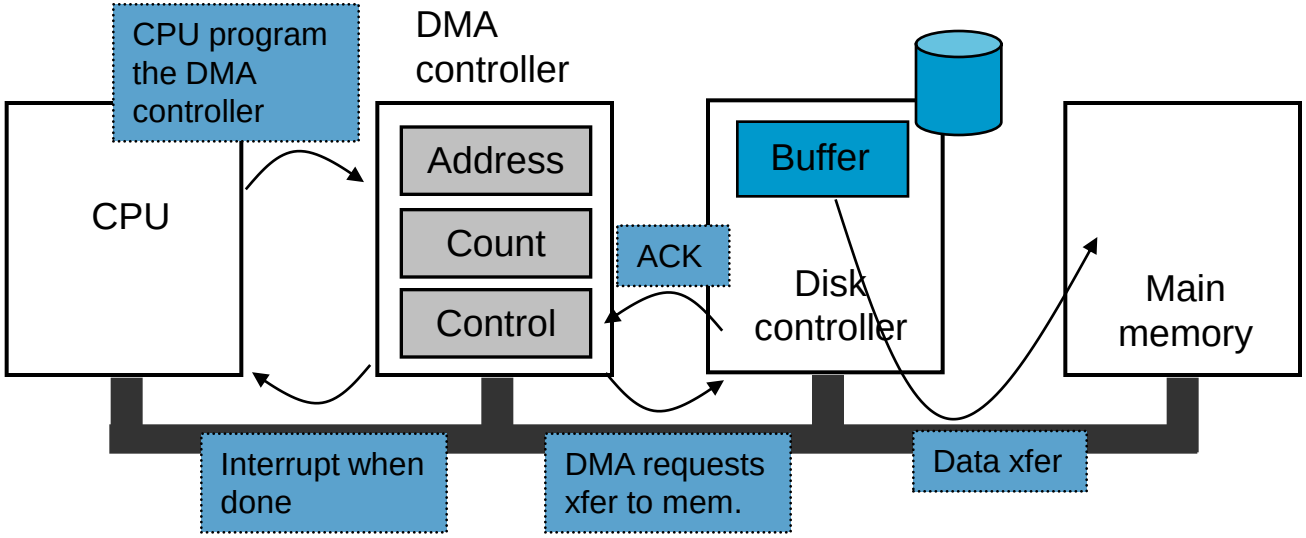
A more complex system

- Pentium system



Direct Memory Access (DMA)

- With or w/o memory-mapped I/O – CPU has to address the device controllers to exchange data
  - By itself, one byte at a time
  - Somebody else doing it instead – DMA
- Clearly OS can use it only if HW has DMA controller
- DMA operation



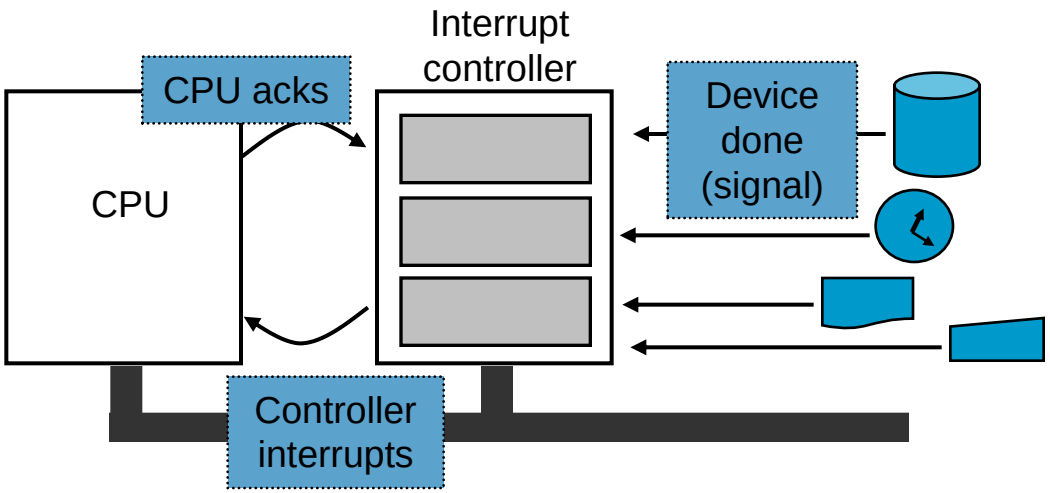
Some details on DMA

- One or more transfers at a time
  - Need multiple set of registers for the multiple channels
  - DMA has to schedule itself over devices served
- Buses and DMA can operate on one of two modes
  - Cycle stealing – as described
  - Burst mode (block) – DMA tells the device to take the bus for a while
- Two approaches to data transfer
  - Fly-by mode – just discussed, direct transfer to memory
  - Two steps – transfer via DMA; it requires extra bus cycle, but now you can do device-to-device transfers
- Physical (common) or virtual address for DMA transfer
- *Why you may not want a DMA?*

  If the CPU is fast and there's not much else to do anyway

Interrupts revisited

- When I/O is done – interrupt by asserting a signal on a bus line
- Interrupt controller puts a # on address lines – index into interrupt vector (PC to interrupt service procedure)
- Interrupt service procedure ACK the controller
- Before serving interrupt, save context …



Interrupts revisited

Not that simple …

- Where do you save the state?
  - Internal registers? Hold your ACK (to protect you from another interrupt overwriting the internal registers)
  - In stack? You can get a page fault … pinned page?
  - In kernel stack? change to kernel mode ($$$ - change MMU context, invalid cache and TLB,…)
- Besides: pipelining, superscalar architectures, …

Ideally - a precise interrupt

- PC is saved in a known place
- All previous instructions have been fully executed
- All following ones have not
- The execution state of the instruction pointed by PC is known

The tradeoff – complex OS or really complex interrupt logic within the CPU (design complexity & chip area)

I/O software – goals & issues

- Device independence
  - Programs can access any I/O device w/o specifying it in advance
- Uniform naming, closely related
  - Name independent of device
- Error handling
  - As close to the hardware as possible (first the controller should try, then the device driver, …)
- Buffering for better performance
  - Check what to do with packets, for example
  - Decouple production/consumption
- Deal with dedicated (tape drives) & shared devices (disks)

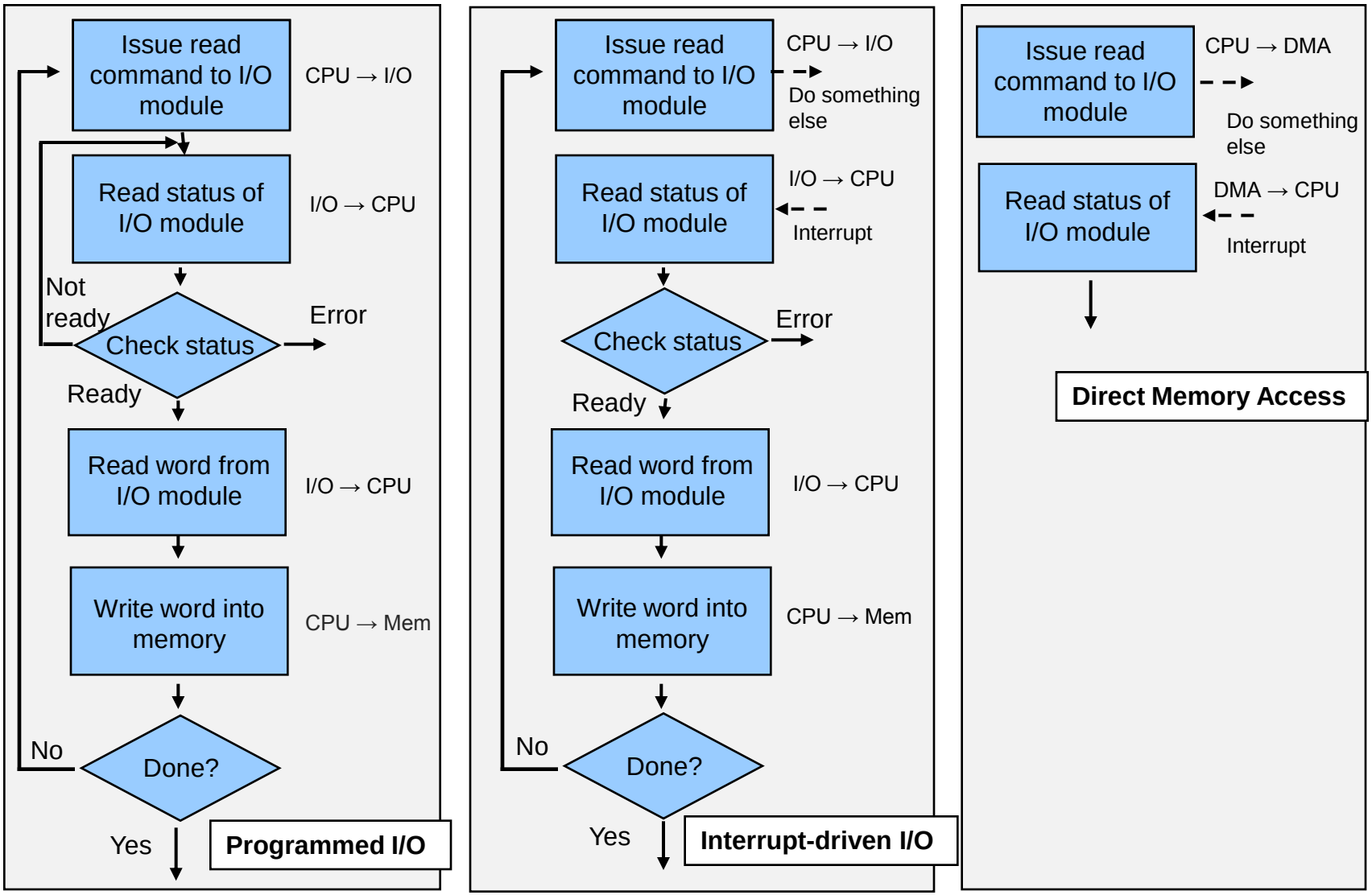
  Dedicated dev. bring their own problems – deadlock?

Ways I/O can be done (OS take)

- Programmed I/O
  - Simplest – CPU does all the work
  - CPU basically pools the device
  - … and it is tied up until I/O completes
- Interrupt-driven I/O
  - Instead of waiting for I/O, context switch to another process & use interrupts
- Direct Memory Access
  - Obvious disadvantage of interrupt-driven I/O?

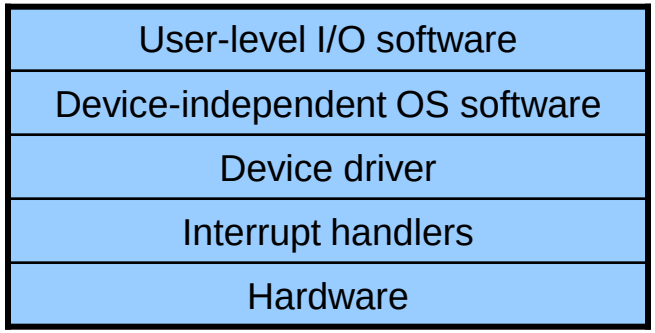
    An interrupt for every character
  - Solution: DMA - Basically programmed I/O done by somebody else

Three techniques for I/O



I/O software layers

- I/O normally implemented in layers



- Interrupt handlers
  - Interrupts – an unpleasant fact of life – hide them!
  - Best way
    - Driver blocks (semaphores?) until I/O completes
    - Upon an interrupt, interrupt procedure handles it before unblocking driver

Layers - Device drivers

- Different device controllers – different registers, commands, etc → each I/O device needs a device driver
- Device driver – device specific code
  - Written by device manufacturer
  - Better if we have specs
  - Clearly, it needs to be reentrant
  - Must be included in the kernel (as it needs to access the device's hardware) - How do you include it?
    - *Is there another option?*
  - Problem with plug & play

Layers - Device-independent SW

Some part of the I/O SW can be device independent

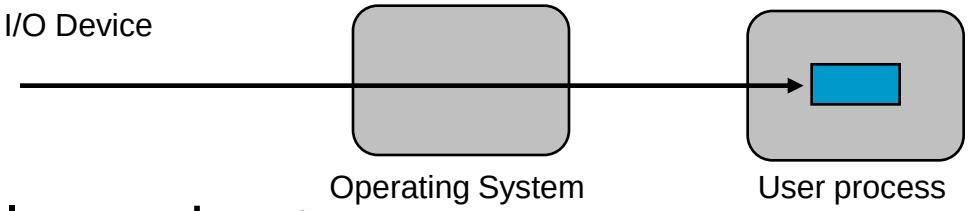
- Uniform interfacing with drivers
  - Fewer modifications to the OS with each new device
  - Easier naming (`/dev/disk0`) – major & minor device #s in UNIX (kept by the i-node of the device's file)
  - Device driver writers know what's expected of them
- Error reporting
  - Some errors are transient – keep them low
  - Actual I/O errors – reporting up when in doubt
- Allocating & releasing dedicated devices
- Providing a device-independent block size
- Buffering

Buffering

A process reading data from a modem

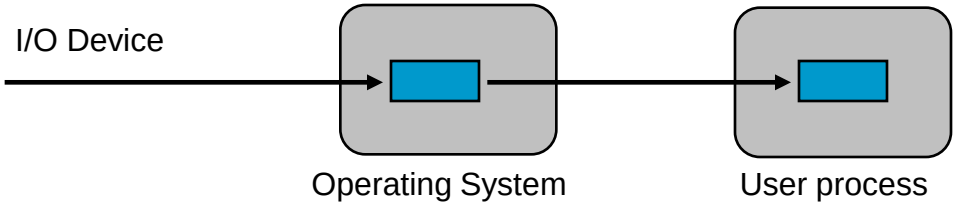
- Unbuffered input

  Wake up the process when the buffer is full



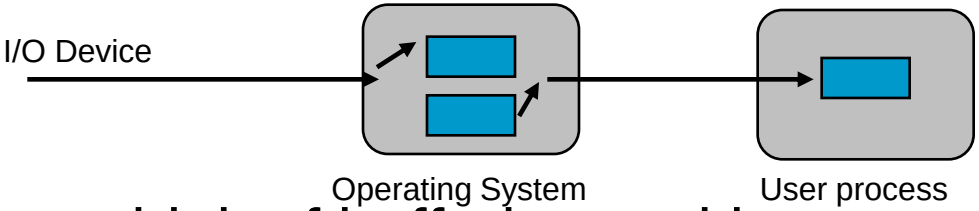
- Buffering in the kernel, copying to user space

  When kernel-buffer is full page with user buffer is brought in & copy done at once



- Double buffering in the kernel

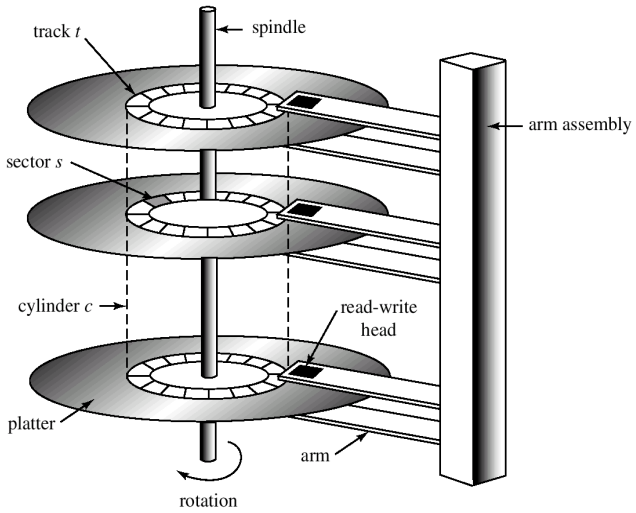
  What happen to characters arriving while the user page is brought in?



- Careful – nothing is free, think of buffering and host-to-host communication

Disk hardware

- Disk organization
  - Cylinders – made of vertical tracks
  - Tracks – divided into sectors
  - Sectors – minimum transfer unit



- Simplified model - careful with specs
  - Sectors per track are not always the same
  - Zoning – zone, a set of tracks with equal sec/track
- Hide this with a logical disk w/ constant sec/track

IBM 2314

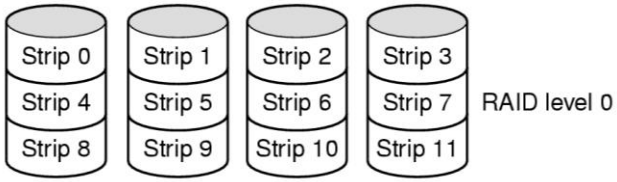
- Announced April 1965
- Eight disk drives plus a spare one and a control unit
- Capacity: 8x29MB
- Average access time: 60msec
- Data rate: 312KB/sec

Disk hardware nowadays

Characteristics (All 512B/sector)	**Seagate Cheetah 15-36LP**	**Seagate Barracuda 36ES**	**Toshiba HDD1242**	**IBM Microdrive**
Application	High-performance server	Entry-level desktop	Portable	Handheld
Capacity	36.7GB	18.4GB	5GB	1GB
Minimum seek time	0.3ms	1.0ms	-	1.0ms
Average seek time	3.6ms	9.5ms	15ms	12ms
Spindle speed	15K rpm	7.2K rpm	4.2K rpm	3.6K rpm
Average rotational delay	2ms	4.17ms	7.14ms	8.33ms
Max. transfer rate	522-709MB/s	25MB/s	66MB/s	13.3MB/s
Sector per track	485	600	63	-
Tracks per cylinder	8	2	2	2
Cylinders	18,479	29,851	10,350	-

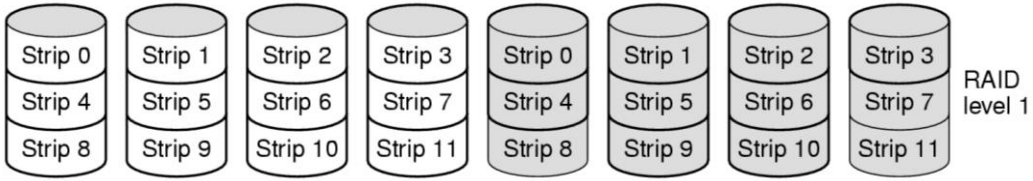
RAIDs

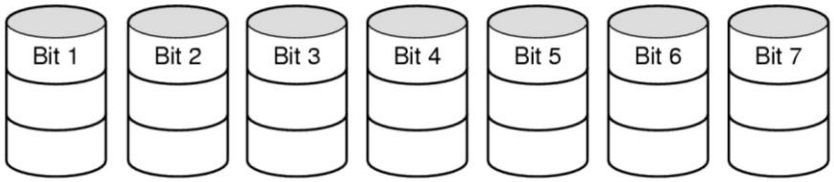
- Processing and I/O - parallelism
- Redundant Array of Inexpensive Disks & SLED Single Large Expensive Disks



RAID 0: Striping w/o redundancy

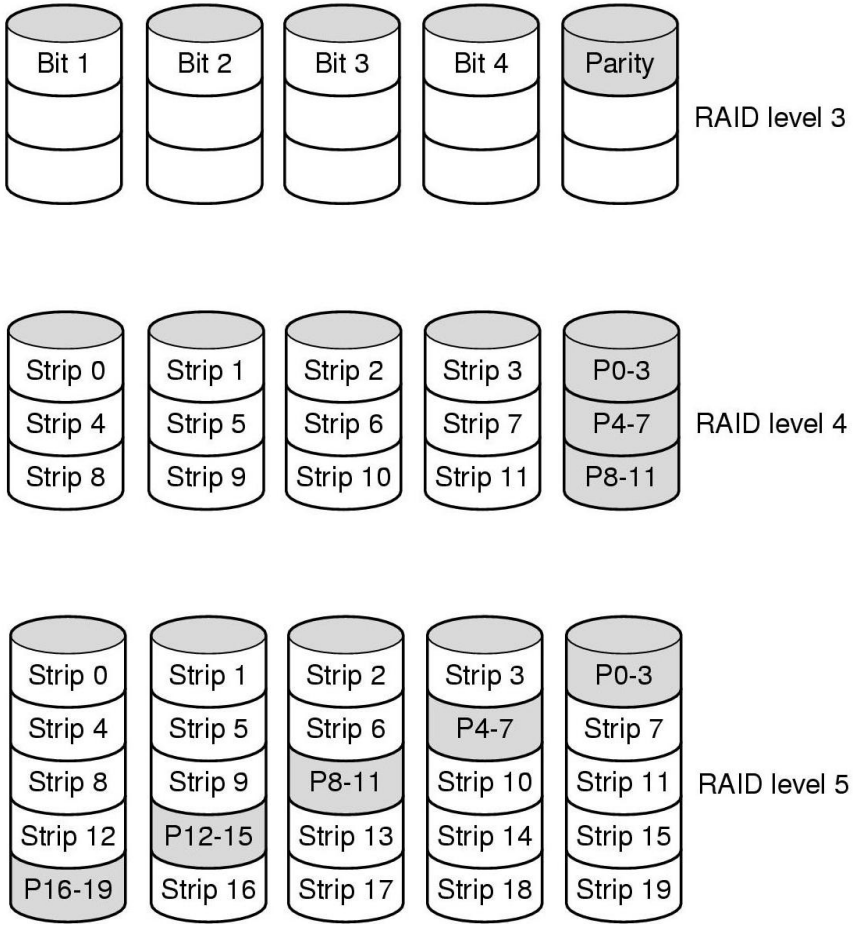
RAID 1: Disk mirroring





RAID 2: Memory-style error-correcting-code organization; striping bits across disk & add ECC

RAIDs



RAID 3: Bit-interleaved parity organization. If one sector is damage you know which one; used the parity to figure out what the lost bit is.

RAID 4: Block-interleaved parity organization. Like 3 but at block level.

RAID 5: Block-interleaved distributed parity organization. Like 4 but distributed the parity block.

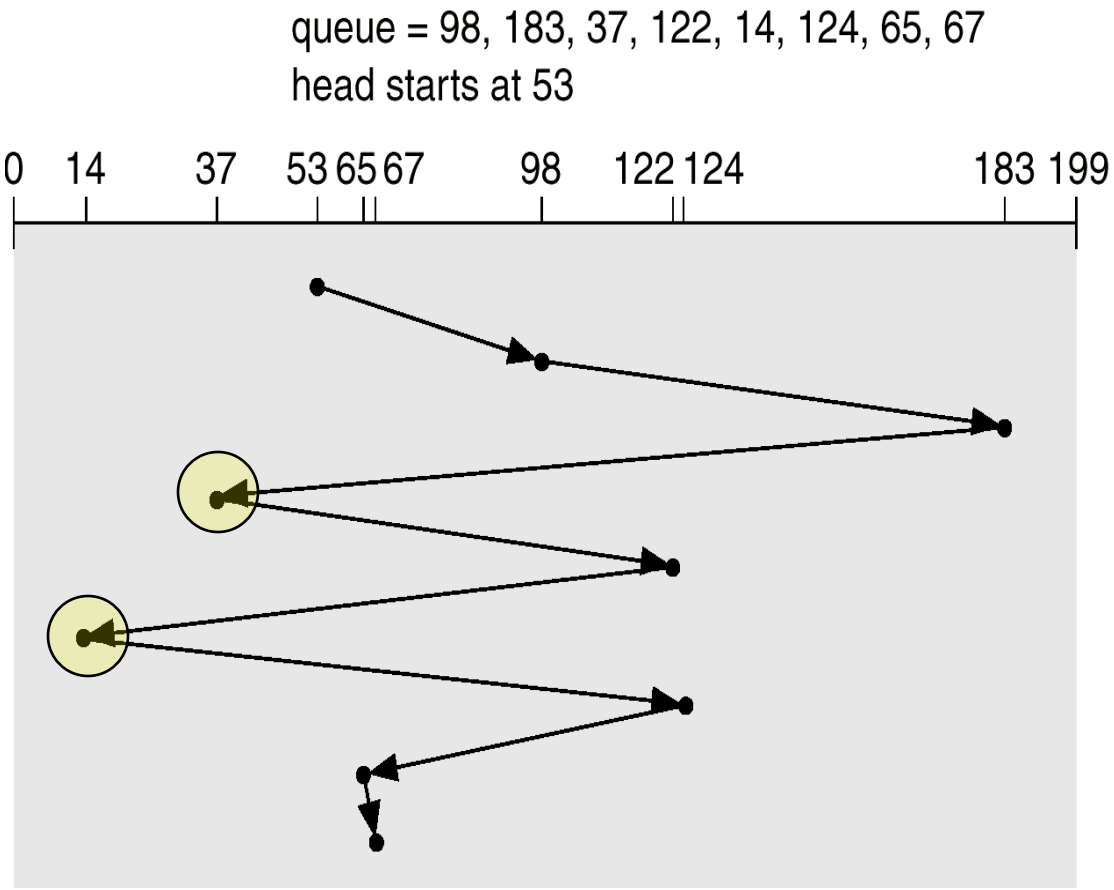
Disk formatting

- Low-level formatting
  - Sectors – [preamble, to recognize the start + data + ecc]
  - Spare sectors for replacements
  - ~20% capacity goes with it
    - Sectors and head skews to deal with moving head
    - Interleaving to deal with transfer time
- Partitioning – multiple logical disks – sector 0 holds master boot record (boot code + partition table)
- High-level formatting
  - Boot block, free storage admin, root dir, empty file system

Disk arm scheduling

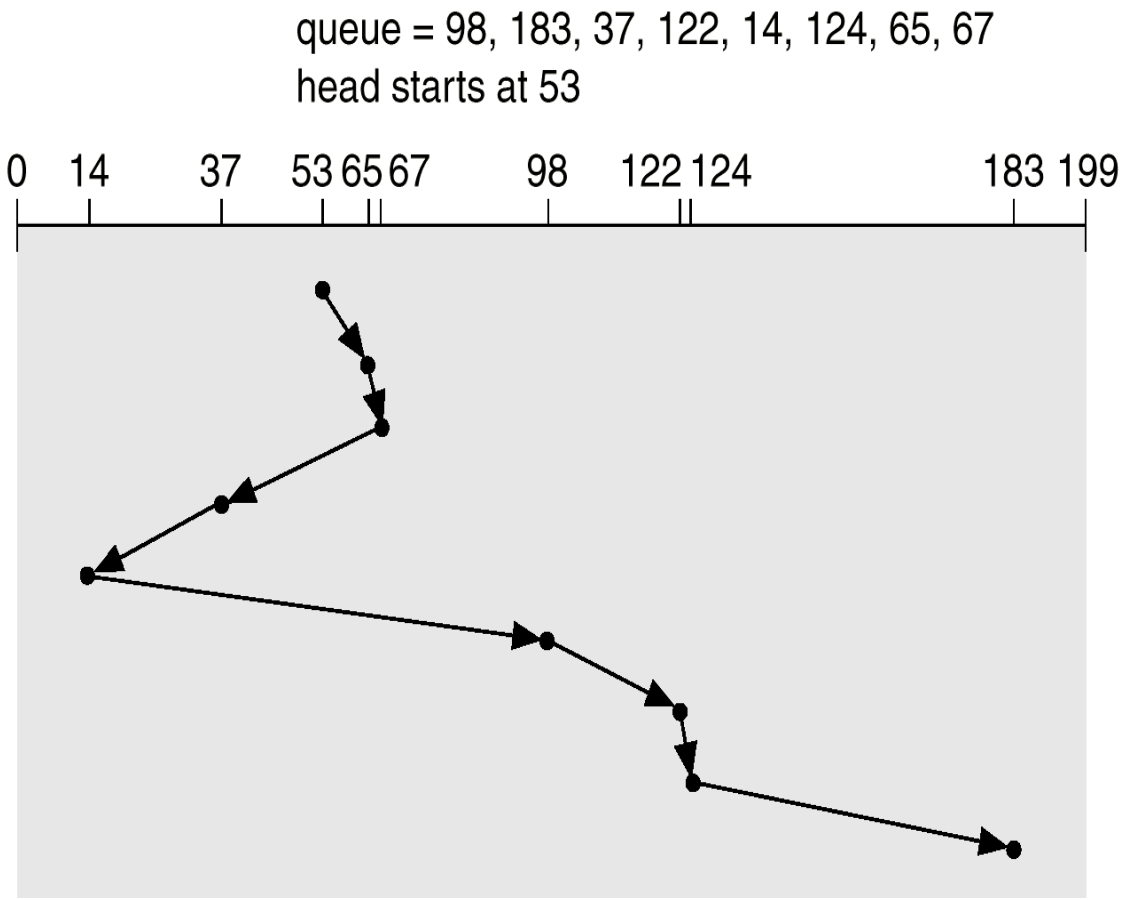
- Time to read/write a disk block determined by
  - Seek time – dominates!
  - Rotational delay
  - Actual transfer time
- Sequence of request for blocks → scheduling
- Some algorithms
  - FCFS (FIFO) – not much you can do
  - Shortest Seek Time First (SSTF)
  - Elevator algorithm or SCAN
  - C-SCAN - circular scan
  - LOOK variation
- Reading beyond your needs

FCFS



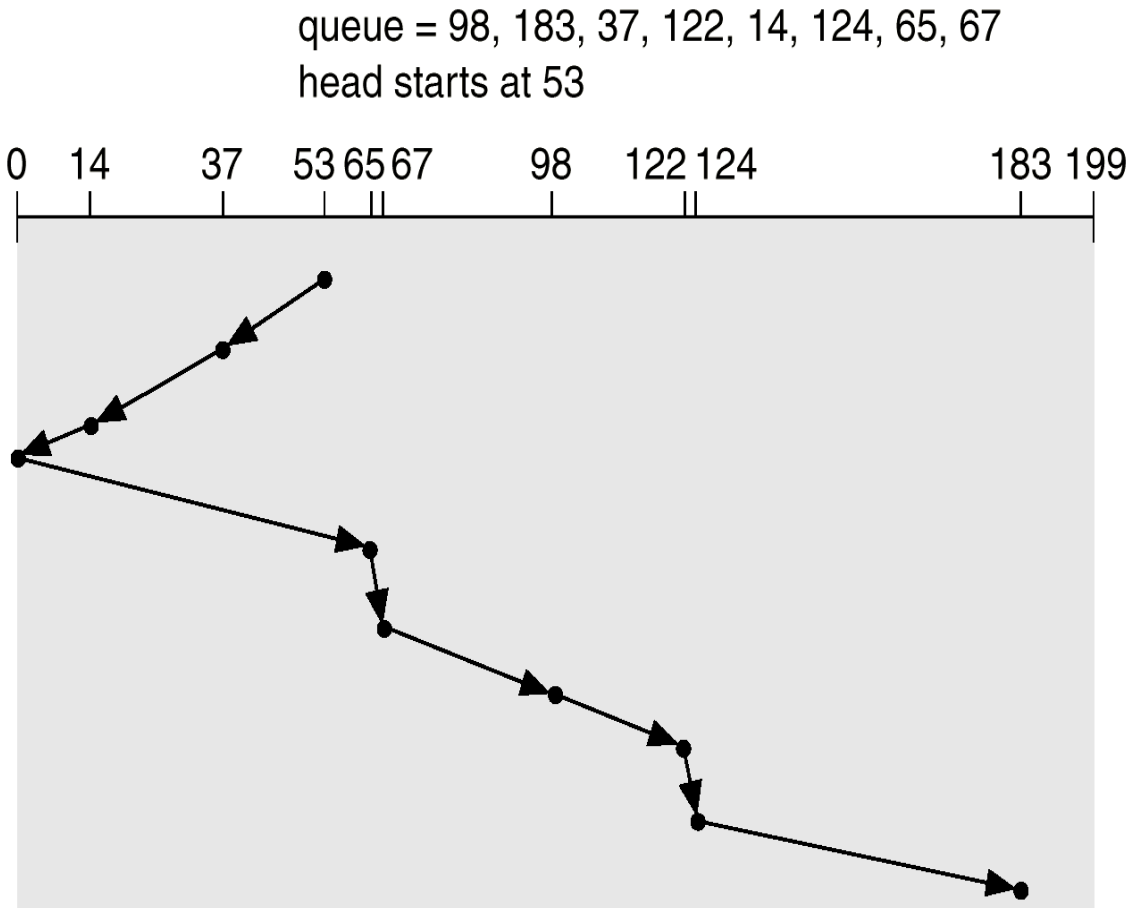
- and the obvious problem is …

SSTF



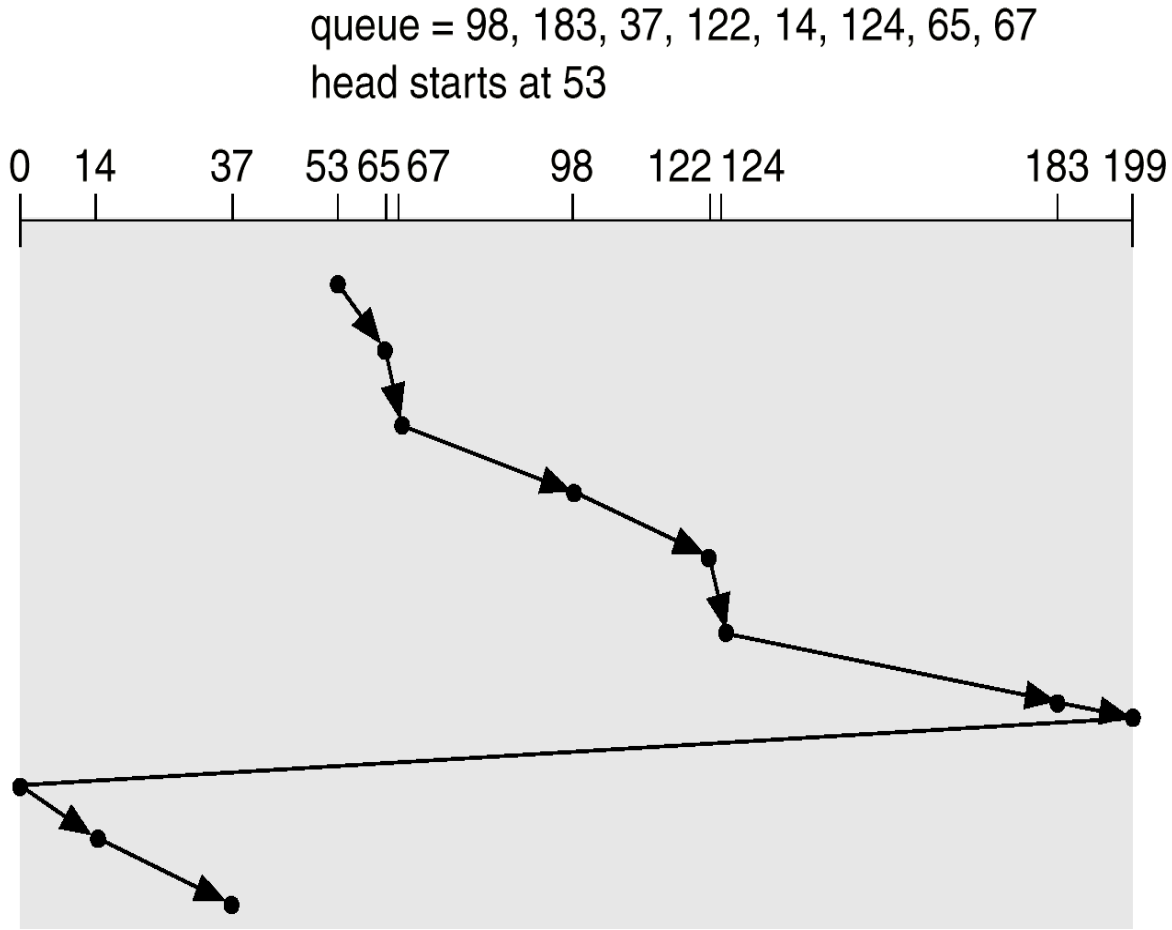
- As SJF, possible starvation

SCAN



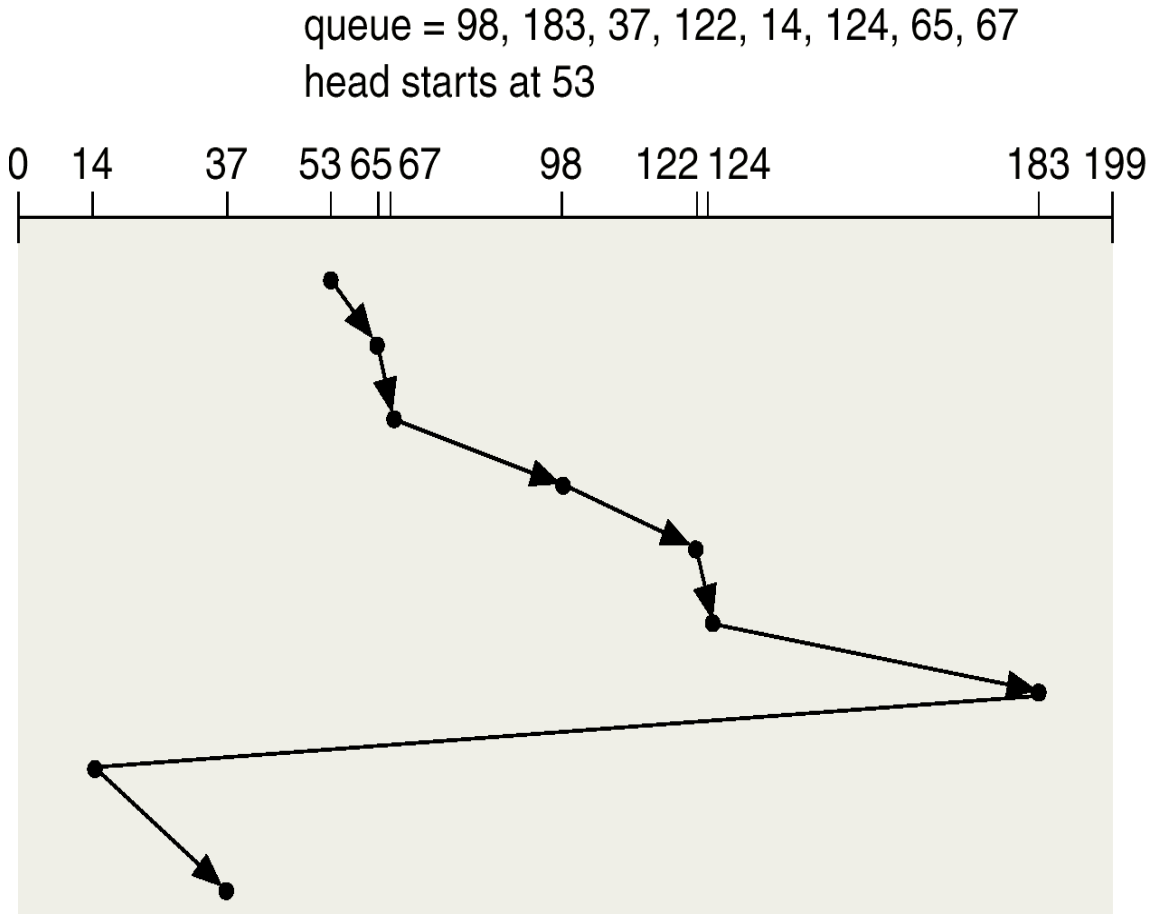
- Assuming a uniform distribution of requests, where's the highest density when head is on the left?

C-SCAN



- Cool, but no need to be blind

C-LOOK



- Look for a request before moving in that direction

Next time

- Protecting access to your system and paying attention to the system's environment
- Final review and a taste of systems research