Input/Output

To do …

- Principles of I/O hardware & software
- I/O software layers
- Secondary storage
- Next: File systems

Operating systems and I/O

- Two key OS goals with respect to I/O
  - Control I/O devices
  - Provide a simple, easy-to-use, interface to devices
- Problem – large variety of devices
  - Data rates – From 10B/sec (keyboard) to 12,800MB/sec (100 Gigabit Ethernet)
  - Complexity of control – A printer (simple) or a disk
  - Units of transfer – Character or block devices
  - Data representation – Character codes, parity
  - Error condition – nature of errors, how they are reported, their consequences, …
- Difficult to get a uniform & consistent approach

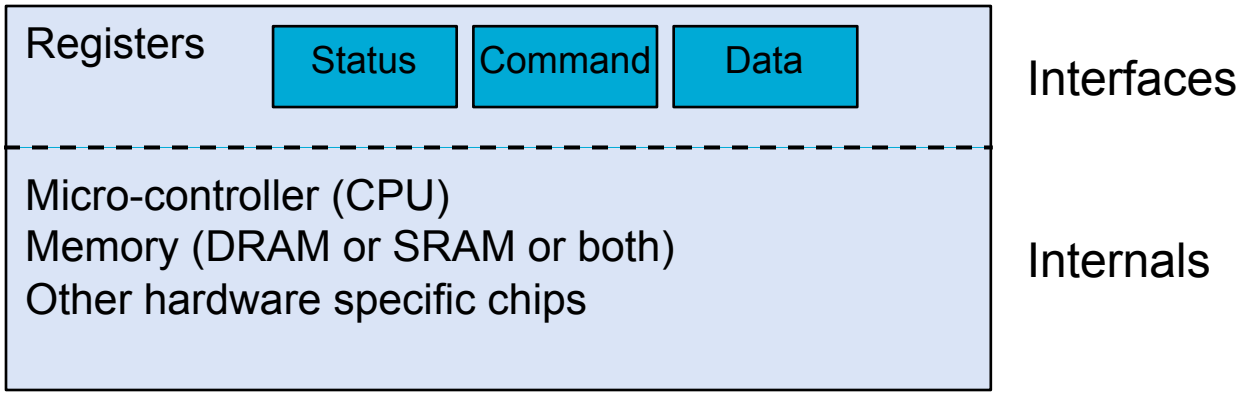
I/O devices

- I/O devices components
  - Device itself – mechanical component
  - Device controller or adapter – electronic component

- Device controller
  - Converts serial bit stream to block of bytes
  - Performs error correction as necessary
  - Makes data available in main memory
  - Maybe more than one device per controller
  - Some standard interface between controller and devices: IDE, ATA, SATA, SCSI, FireWire, Thunderbolt, …

I/O controller & CPU communication

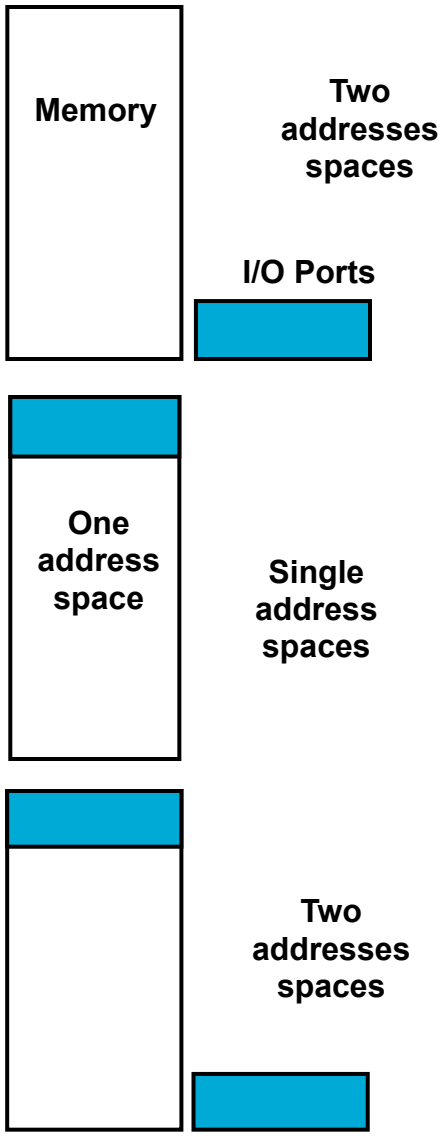
- Device controllers have
  - A few registers for communication with CPU
    - Data-in, data-out, status, control, command, …
  - A buffer that OS can read/write (e.g. video RAM)



A canonical device

I/O controller & CPU communication

- How does the CPU use that?
  a. Separate I/O and memory space, each control register assigned an I/O port – IBM 360 (`IN REG,PORT`)
  b. Memory-mapped I/O – Mapped all control registers into the memory space; first in PDP-11
  c. Hybrid – Pentium (e.g., graphic controller)

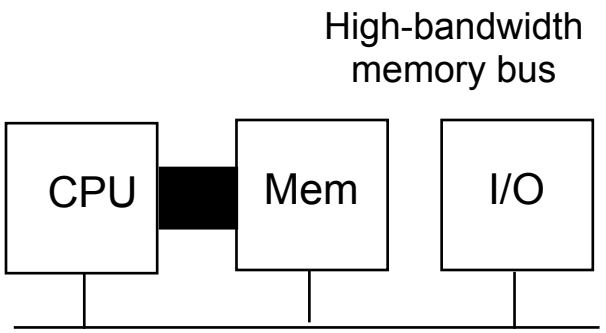


Memory-mapped I/O – pros and cons

- ✔ No special instructions or protection mechanism needed
  - Instruction that can reference memory can reference control registers
- ✔ Driver can be entirely written in C

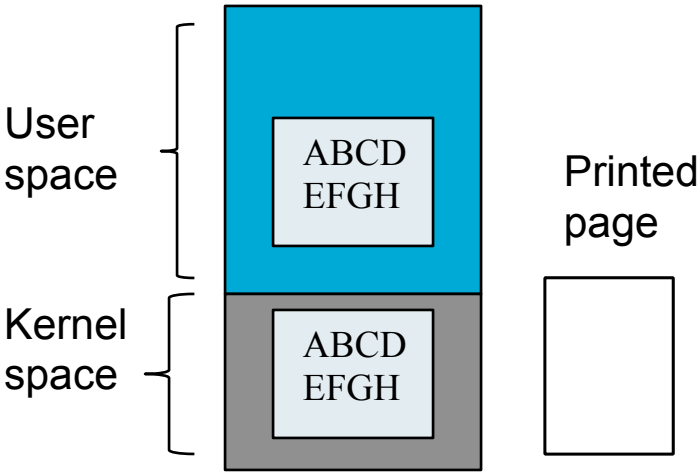
Memory-mapped I/O – pros and cons

- ✗ Caching? Disable it on a per-page basis
- ✗ One AS, so all memory modules and all I/O devices must check all references
  - Easy with single bus, harder with dual-bus arch
  - Possible solutions
    - Send all references to memory first, if fails try bus
    - Snoop in the memory bus
    - Filter addresses in the PCI bridge (preloaded with range registers at boot time)

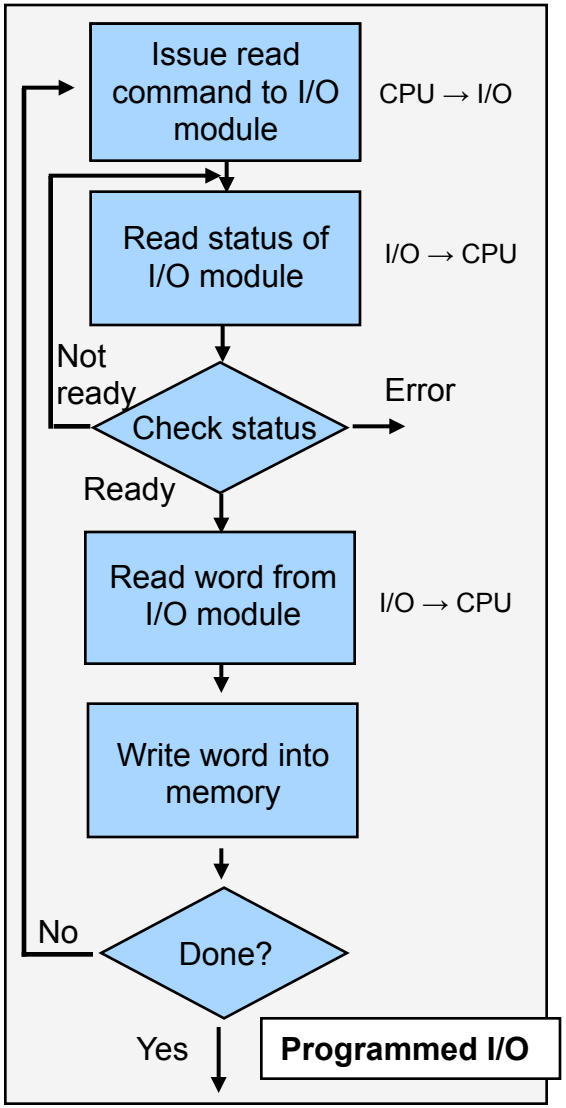


Ways I/O can be done (OS take)

- Programmed I/O
  - Simplest – CPU does all the work
  - CPU polls the device
  - … it is tied up until I/O completes




```
copy_from_user((buffer_p, count);
for (i = 0; i < count; i++) {
   while(STATUS == BUSY);
   Write p[i] to DATA
   Write command to COMMAND
}
return_to_user();
```



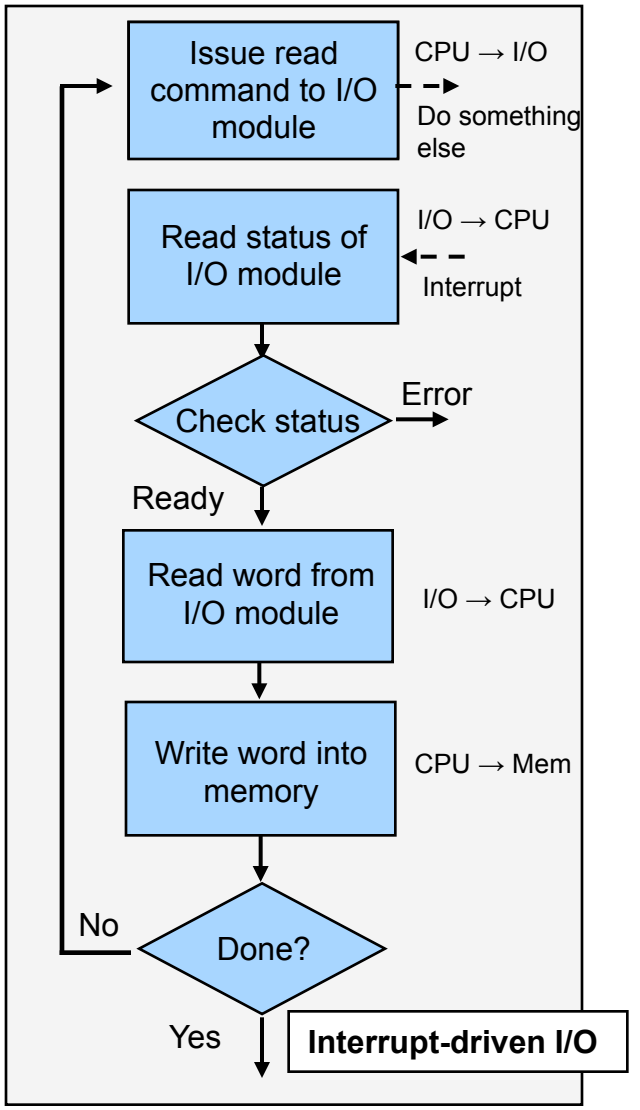
Ways I/O can be done

- Interrupt-driven I/O
  - Instead of waiting for I/O, context switch to another process & use interrupts

```
copy_from_user((buffer_p, count);
enable_interrupts();
While(STATUS == BUSY);
Write p[0] to DATA
Write command to COMMAND
scheduler();
```

Interrupt service procedure

```
if(count == 0) {
   unblock_user();
} else {
   Write p[i] to DATA
   --count; ++i;
   Write command to COMMAND
}
acknowledge_interrupt();
return_from_interrupt();
```



Ways I/O can be done

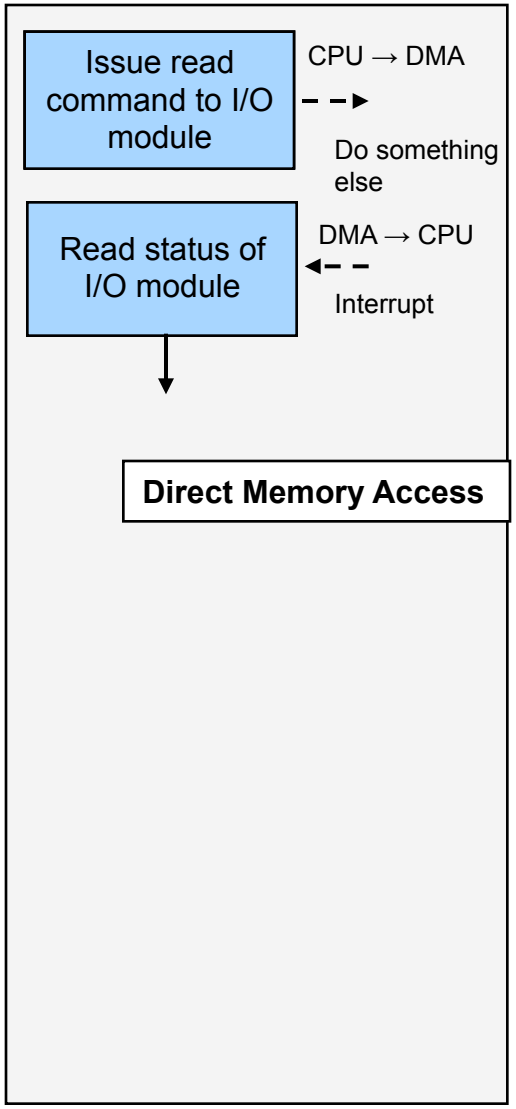
- Direct Memory Access
  - Obvious disadvantage of interrupt-driven I/O?

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

```
copy_from_user((buffer_p, count);
set_up_DMA_controller();
scheduler();
```

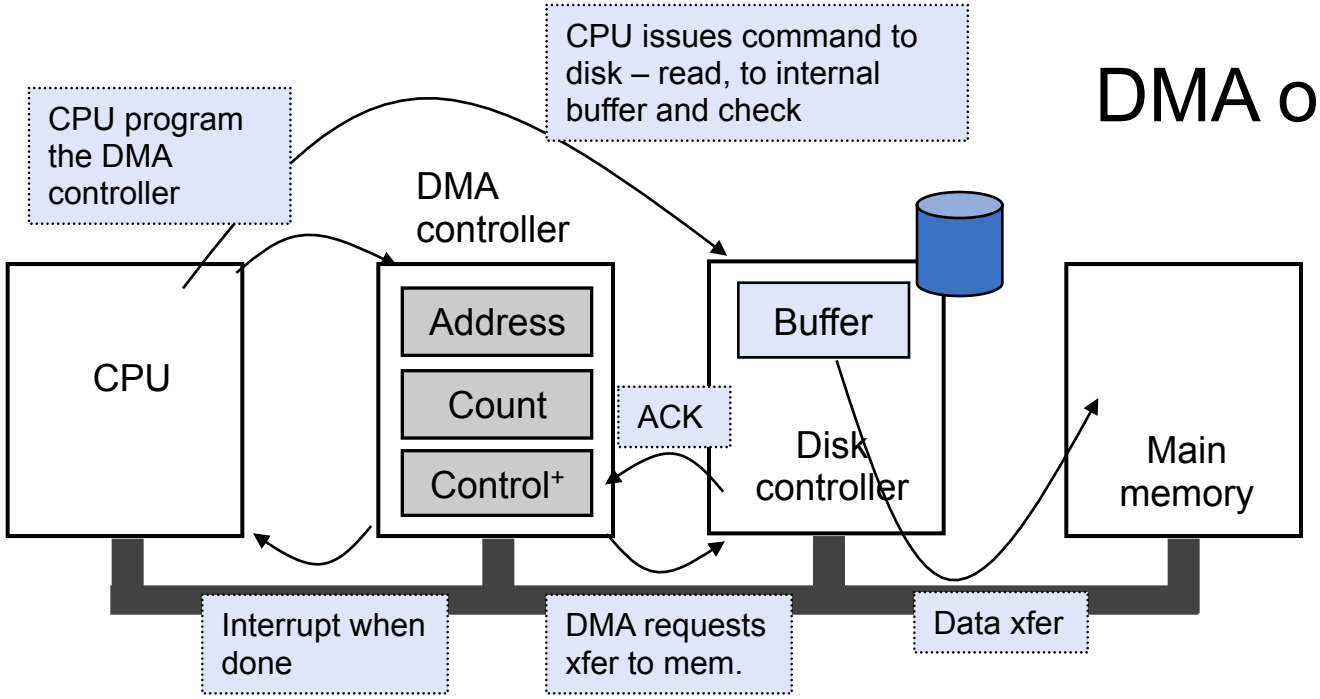
Interrupt service procedure

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



Direct Memory Access

- OS can use it only if HW has DMA controller
  - Either on the devices (controller) or the parentboard
- DMA controller has access to the system bus, independent of CPU
  - Various registers accessible by the CPU



DMA operation

Some details on DMA ...

- One or more transfers at a time
  - Need multiple set of registers for multiple channels
    - Often a different ack line on the bus for each DMA channel
  - DMA has to schedule itself over devices served
- Buses and DMA can operate on two modes
  - Cycle stealing – Device controller occasionally steals the bus, delaying the CPU a bit
  - Burst mode (block) – DMA tells the device to take the bus for a while, more efficient transfer, longer wait for others

Some details on DMA

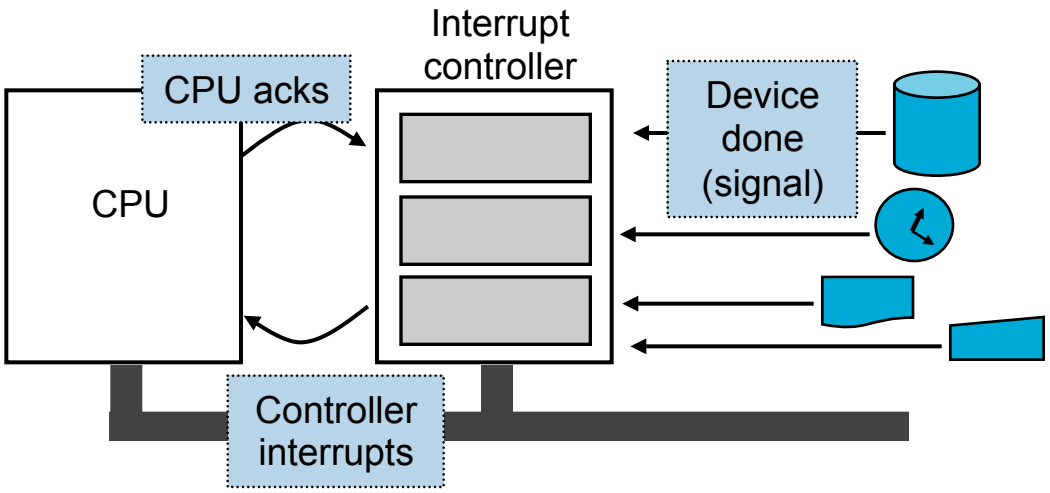
- Two approaches to data transfer
  - Fly-by mode – As discussed, direct xfer to memory
  - Two steps – xfer via DMA; it requires extra bus cycle, but you can do device-to-device or memory-to-memory transfers
- Physical (common) or virtual address for DMA transfer

- *Why you may not want a DMA?*

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

Interrupts revisited

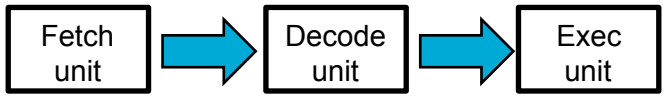
- I/O is done – interrupt asserting a signal on a bus line
- Interrupt controller detects it and puts a # on address lines – index into interrupt vector
- Signal makes CPU stop what is doing and change PC to interrupt service procedure (found in vector)
- Interrupt service procedure ACK the controller
- Before serving interrupt, save context …

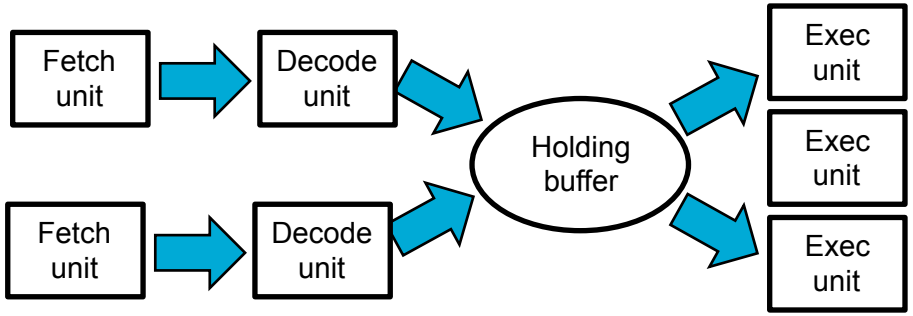


Interrupts revisited

Not that simple …

- Where do you save the state?
  - Internal registers? Hold your ACK until all potentially relevant info is read (to avoid overwriting internal regs)
  - On the current stack? Could be user's and get a page fault … pinned page?
  - On the kernel stack? Change to kernel mode $$$
- Besides: pipelining, superscalar architectures, …
  - When interrupt is signaled there may be instructions in various state of execution
  - PC may not reflect boundary between executed/not executed





Interrupts revisited

- Ideally – a *precise* interrupt, leaves machine in a well-defined state
  1. PC is saved in a known place
  2. All previous instructions have been fully executed
  3. All following ones have not (partially perhaps but should be undone before the interrupt happens)
  4. The exec state of the instruction pointed by PC is known

- No prohibition on what instructions to start
  - but all changes to register or mem must be undone before interrupt happens
- Tradeoff – complex OS or really complex interrupt logic within the CPU (design complexity & chip area)

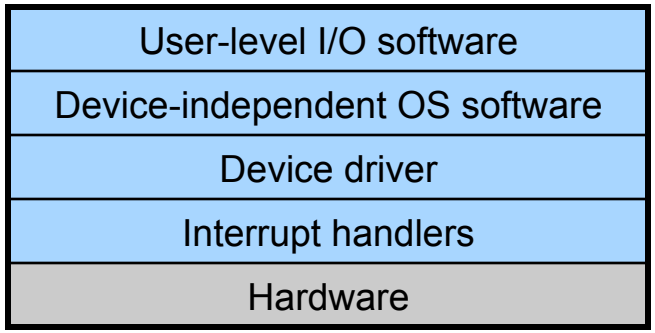
I/O software – Goals & issues

- Device independence
  - Programs can access any I/O dev without having to be modified
- Uniform naming, closely related
  - Name independent of device
- Error handling
  - As close to hardware as possible
    - Many errors are transient or can be dealt with, transparently at a low lever
- Buffering for better performance
  - Check what to do with packets, for example
  - Decouple production/consumption
- Deal with dedicated (tape) & shared devices (disks)

  Dedicated devices bring their own problems – deadlock?

I/O software layers

- I/O normally implemented in layers



- Interrupt handlers
  - Interrupts – unpleasant fact of life, hide them!
  - Best way
    - Driver blocks (semaphores, condition signal, waiting on a msg?) 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 device needs a driver
- Device driver – device specific code
  - Written by device manufacturer
  - Better if we have specs
  - Clearly, it needs to be reentrant (I/O device may complete while the driver is running, interrupting the driver and maybe making it run …)
  - 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

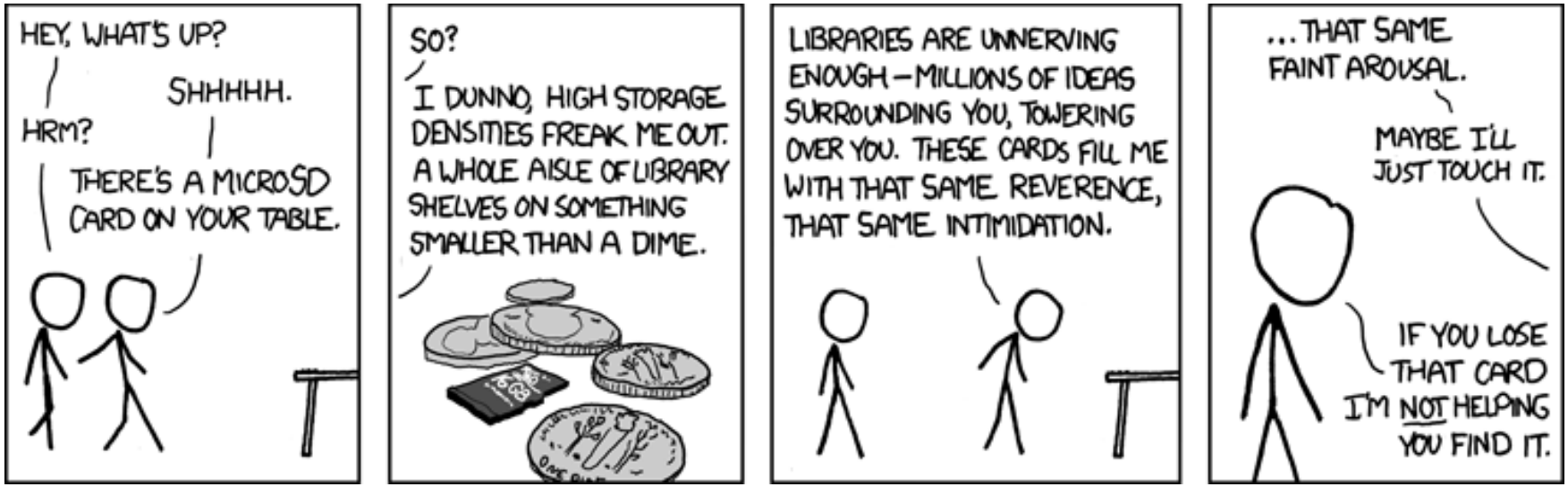
Layers - Device-independent SW

- Some 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, driver + unit, (kept by i-node of device's file)
  - Device driver writers know what's expected of them
- Buffering
  - Unbuffered, user space, kernel, …
- 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

User-space I/O software

- Small portion of I/O software runs in user-space
- Libraries that linked together with user programs
  - E.g., stdio in C
  - Mostly parameter checking and some formatting (printf)
- Beyond libraries, e.g. spooling
  - Handling dedicated devices (printers) in a multiprogramming system
  - Daemon plus spooling directory

Now a short break ...



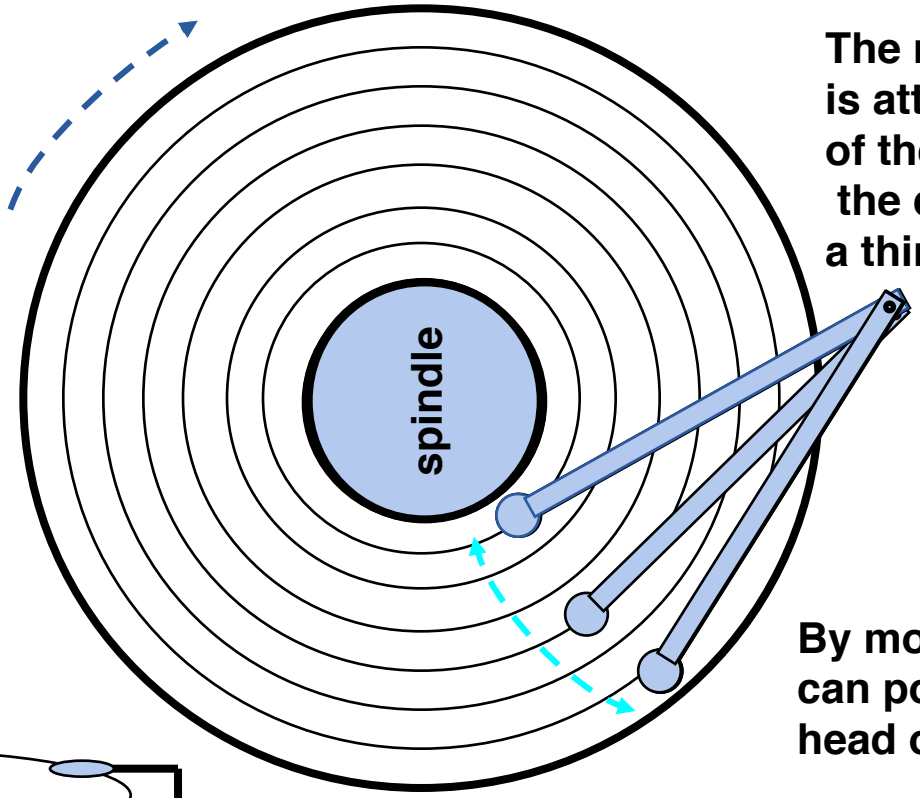
xkcd

Disk – a concrete I/O device

- Interface
  - A large number of sectors that can be read/written; each numbered from 0 to n – 1
    - Multi-sector operations are possible but only a single block write is atomic (so a portion of a larger write can fail)
- Magnetic disk hardware - organization
  - Platter with two surfaces
  - Tracks – divided in sectors
  - Cylinders – made of vertical tracks
  - 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

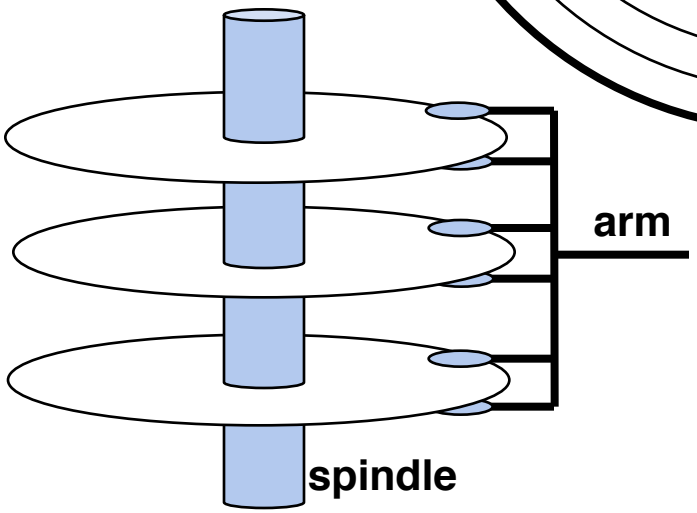
Disk operation (Single-platter view)

The disk surface spins at a fixed rotational rate (5400-15000 RPM)



The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.



read/write heads move in unison from cylinder to cylinder

Disk access time

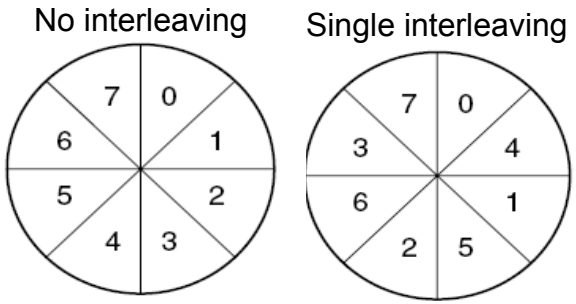
- Avg time to access target sector approximated by
  - $T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$
- Seek time ($T_{avg\ seek}$)
  - Time to position heads over cylinder containing target sector
  - Typical $T_{avg\ seek}$ = 9 ms
- Rotational latency ($T_{avg\ rotation}$)
  - Time waiting for first bit of sector to pass under r/w head
  - $T_{avg\ rotation} = T_{max\ rotation}/2 =$ 1/RPMs x 60 sec/1 min x 1/2
- Transfer time ($T_{avg\ transfer}$)
  - Time to read the bits in the target sector
  - $T_{avg\ transfer}$ = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min

Disk access time example

- Given (~Seagate Barracuda)
  - Rotational rate = 7,200 RPM
  - Avg seek = 9 ms
  - Avg # sectors/track = 400
- Derived:
  - $T_{avg\ rotation}$ = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms
  - $T_{avg\ transfer}$ = 60/7200 RPM x 1/400 sec/track x 1000 ms/sec = 0.02 ms
  - $T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$ = 9 ms + 4 ms + 0.02 ms
- Important points:
  - *Access time dominated by seek time and rotational latency*
  - First bit in a sector is the most expensive, the rest are free
  - SRAM access time ~4 ns/doubleword, DRAM ~60 ns
    - 2,500 times slower than DRAM
    - Disk is about 40,000 times slower than SRAM,

Disk formatting

- Low-level formatting ~20% capacity goes with it
  - Set of concentric tracks of sectors with short gaps in between
  - Sectors – [preamble, to recognize the start + data + ecc]
  - Spare sectors for replacements
  - Sectors and head skews (bet/ tracks) to deal with moving head
  - Interleaving to deal with transfer time (space bet/ consecutive sectors)



- After formatting, partitioning – multiple logical disks – sector 0 holds master boot record (boot code + partition table)

Disks over time

- 20 years trends

Parameter	IBM 360KB floppy	WD 18300 HD
Capacity	360KB	18.3GB
Seek time (avg)	77msec	6.9msec
Rotation time	200msec	8.33msec
Motor stop/start	250msec	20msec
Time to transfer 1 sector	22msec	17μsec

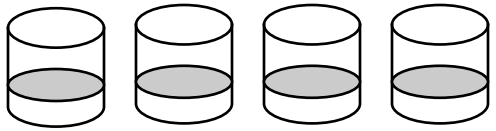
- Note different rates of improvements on seek time, transfer rate and capacity

RAIDs

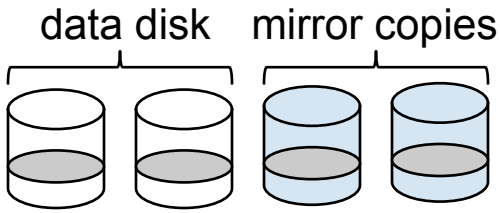
- Redundant Array of ~~Inexpensive~~ Independent Disks
- Disk transfer rates are improving, but slower than CPU performance
- Use multiple disks to improve performance
  - Strip content across multiple disks
  - Use parallel I/O to improve performance
- To the file system, a RAID looks like any other disk – a linear array of blocks
  - Only bigger, potentially faster and potentially more reliable
- Evaluating RAID designs
  - Capacity – with *N* disks, what is the useful capacity available?
  - Reliability – how many disk faults can we tolerate?
  - Performance

RAIDs

- RAID 0 – non-redundant disk array
  - Files (range of sectors) are striped across, non redundant info
  - High read throughput
  - Best write throughput (nothing extra to write)



- But striping reduces reliability (n*MTBF)
  - Add redundancy for reliability
- RAID 1 – mirrored disk
  - Files are striped across half the disks
  - Data is written in two places
  - Read from either copy
  - On failure, just use the surviving one
  - Of course you need 2x space



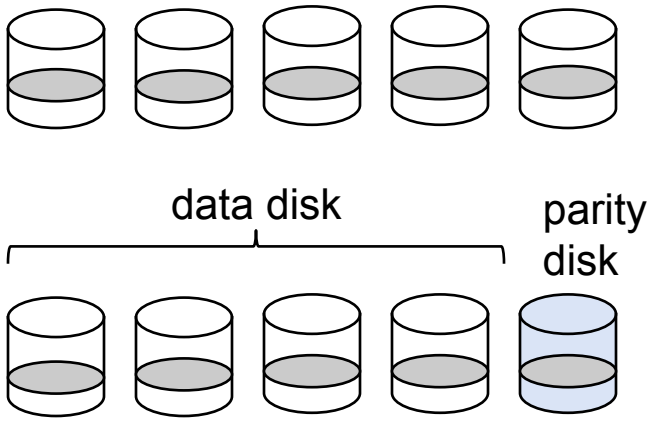
RAIDs

- Another form of redundancy
  - Parity – add a bit to get even number of 1's

1	0	1	1	0	1	1	0	1
0	0	1	1	0	1	1	0	0

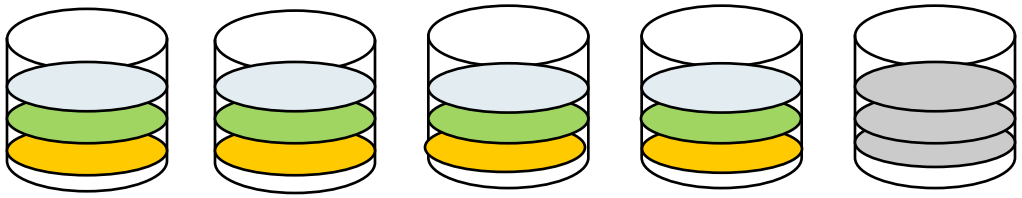
  - Any single missing bit can be reconstructed
    - More complex schemes can detect/correct multiple bit errors

- RAID 2, 3 work on word (or byte) basis
  - Extra EEC bits added to parts of a word and distributed over all disks – RAID 2
  - A single parity bit computed per word and written to the parity disk – RAID 3
  - A read can access all data disks
  - A write updates 1+ data disks and parity disk

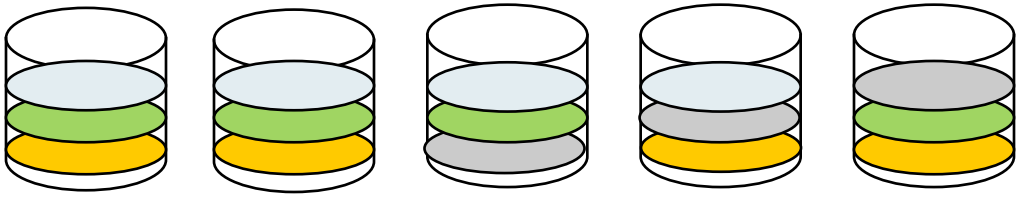


RAIDs

- RAID 4 and 5
  - Work with strips again (not individual words as 2 and 3) and do not require synchronized drivers
  - RAID 4 is like RAID 0 with a strip-for-strip parity written onto an extra drive



- RAID 5 – block interleaved distributed parity
  - Distribute parity info over all disks
  - Much better performance (no hot spot)

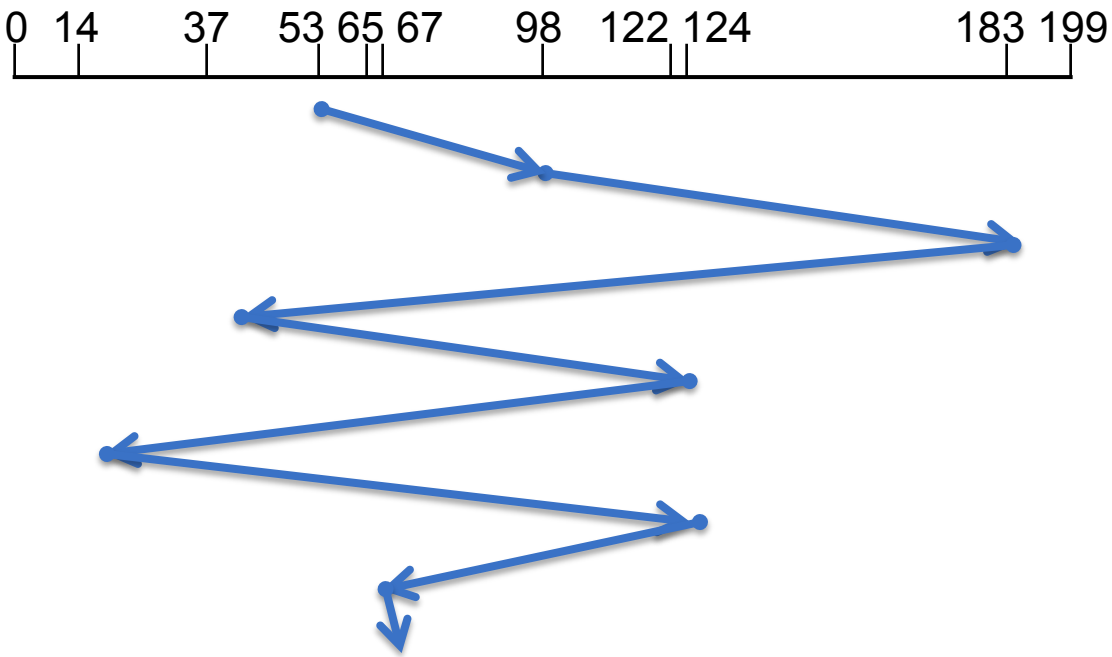


RAIDs tradeoffs

- Granularity
  - Fine-grained – stripe each file over all disks
    - High throughput for the file
    - Limits transfer to one file at a time
  - Course-grained – stripe each file over only a few disks
    - Limit throughput for one file
    - Allows concurrent access to multiple files
- Redundancy
  - Uniformly distribute redundancy information on disks
    - Avoid load-balancing problems
  - Concentrate redundancy information on a small # of disks
    - Partition the disk into data disks and redundancy disks
    - Simpler

Disk arm scheduling

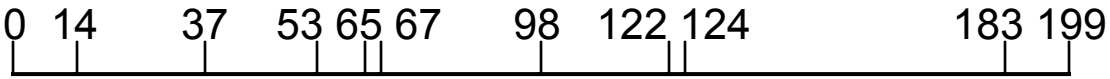
- Time to read/write a disk block determined by
  - Seek time – dominates!
  - Rotational delay
  - Actual transfer time
- If request come one at a time, little you can do - FCFS

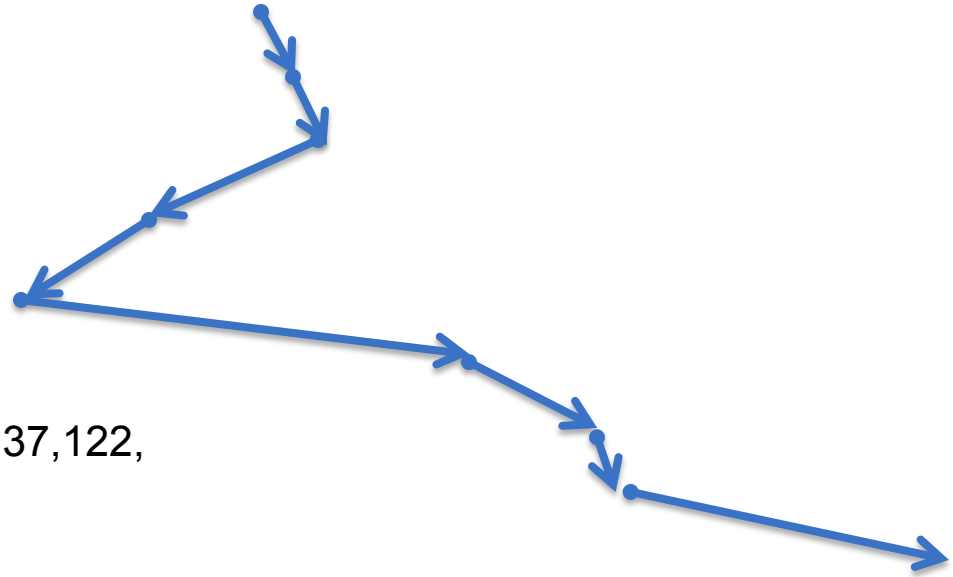


Starting at 53
Requests: 98,183,37,122,
14,124,65,67

SSTF

- Given a queue of request for blocks → scheduling to reduce head movement

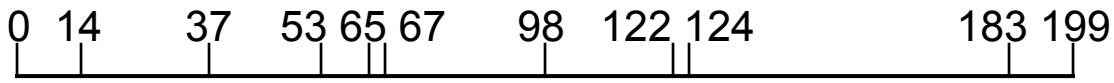


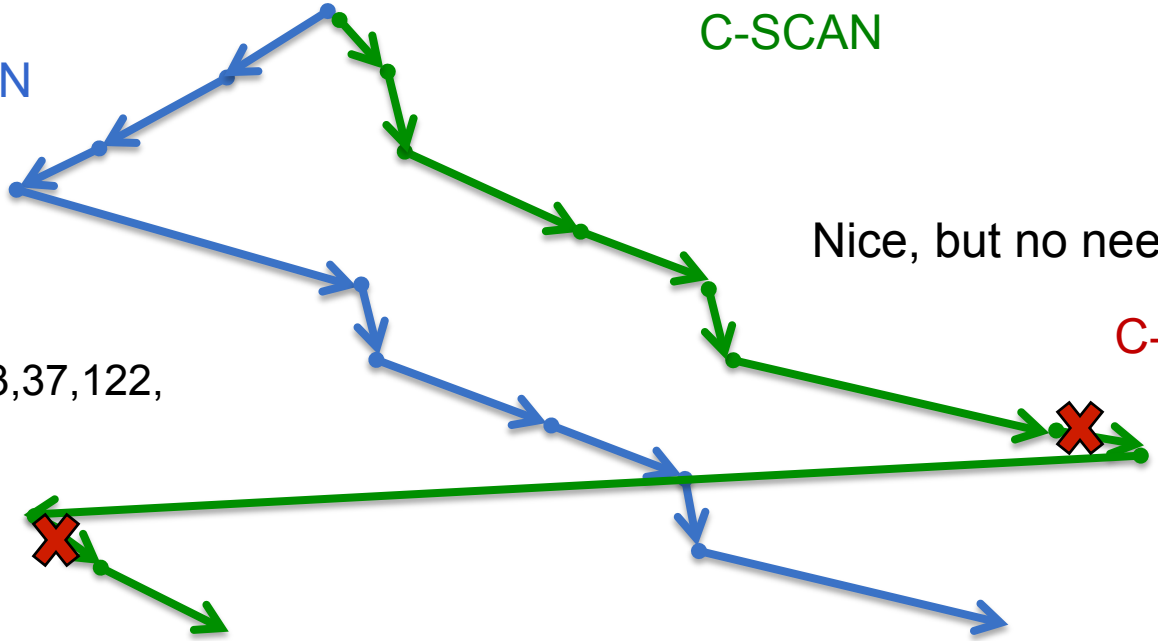


Starting at 53
Requests: 98,183,37,122,
14,124,65,67

- As SJF, possible starvation

SCAN, C-SCAN and C-LOOK





Nice, but no need to be blind

C-LOOK

Starting at 53
Requests: 98,183,37,122,
14,124,65,67

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

Next time

- File systems
  - Interface
  - Implementation
  - And a number of good examples