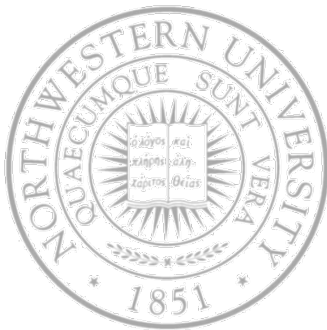


# Input/Output

---



## Today

- Principles of I/O hardware & software
- I/O software layers
- Secondary storage

## Next

- File systems

# 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 – from 10B/sec (keyboard) 125MB/sec (Gigabit Ethernet)
  - 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

# I/O hardware - I/O devices

---

- I/O devices – roughly divided as
  - Block devices – stored info in fixed-size, addressable blocks (e.g. 512 – 32KB), read/write in blocks
    - e.g. disk, CD-ROMs, USB sticks
  - Character devices – I/O stream of non-addressable characters (there's no seek operation)
    - e.g. printers, network interfaces
  - The boundaries are not well defined – tapes?
  - Of course, some devices don't fit in here
    - e.g. clocks

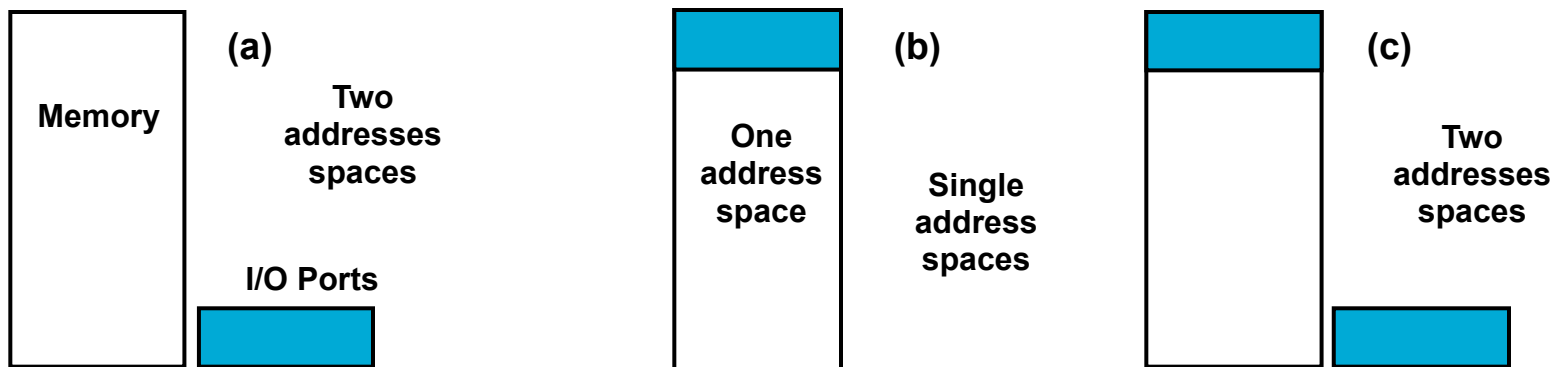
# I/O hardware - I/O devices

---

- I/O devices components
  - Device itself – mechanical component
  - Device controller or adapter – electronic component
- Device controller
  - Maybe more than one device per controller
  - Some standard interface between controller and devices: IDE, ATA, SATA, SCSI, FireWire, ...
  - Converts serial bit stream to block of bytes (think of a disk)
  - Performs error correction as necessary
  - Makes data available in main memory

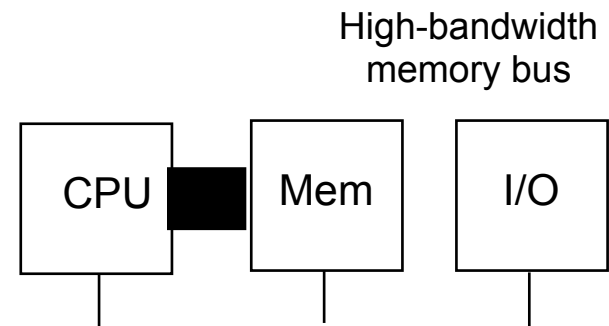
# I/O controller & CPU communication

- Device controllers have
  - A few registers for communication with CPU
    - Typically: 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 and cons

- ★ No special instructions or protection mechanism needed
  - Instruction that can reference mem, can reference control registers
- ★ Driver can be entirely written in C (how to do IN/OUT in C?)
- ✗ What to do with caching? Disable it on a per-page basis
- ✗ One AS, so all mem modules must check all references
  - Easy with single bus, but 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 (Pentium config.)  
(preloaded with range registers at boot time)



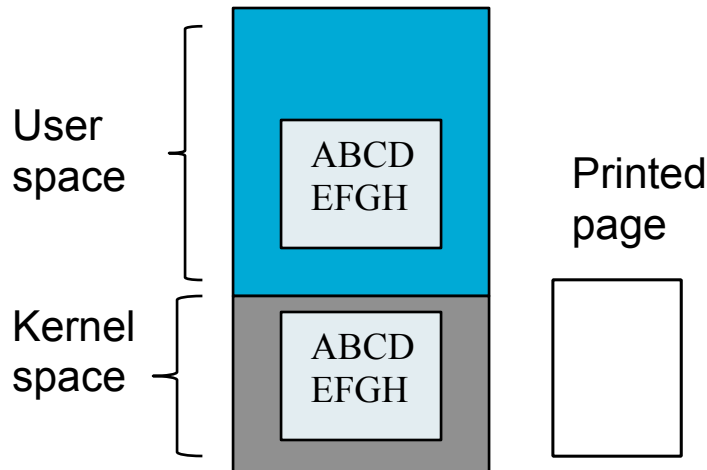
# I/O software – goals & issues

---

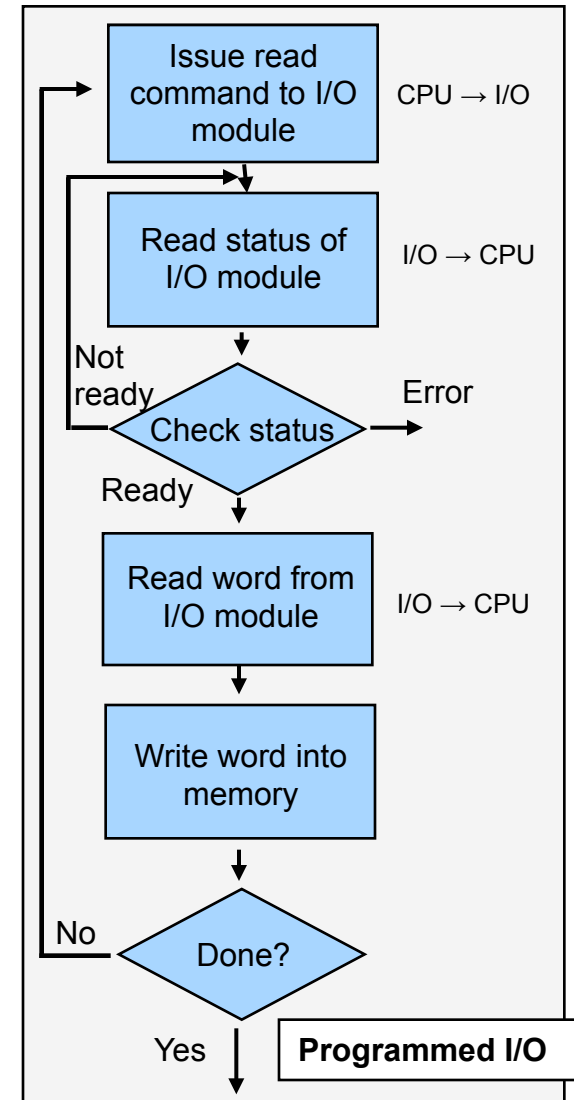
- Device independence
  - Programs can access any I/O device w/o specifying it in advance
- Uniform naming, closely related
  - Name independent of device
- Error handling
  - As close to the hardware as possible (first the controller should try, then the device driver, ...)
    - Many errors are transient or can be dealt with, transparently at a low level
- 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?

# Ways I/O can be done (OS take)

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



```
copy_from_user((buffer_p, count);  
for (i = 0; i < count; i++) {  
    while(*printer_status_reg != READY);  
    *printer_data_reg = p[i];  
}  
return_to_user();
```





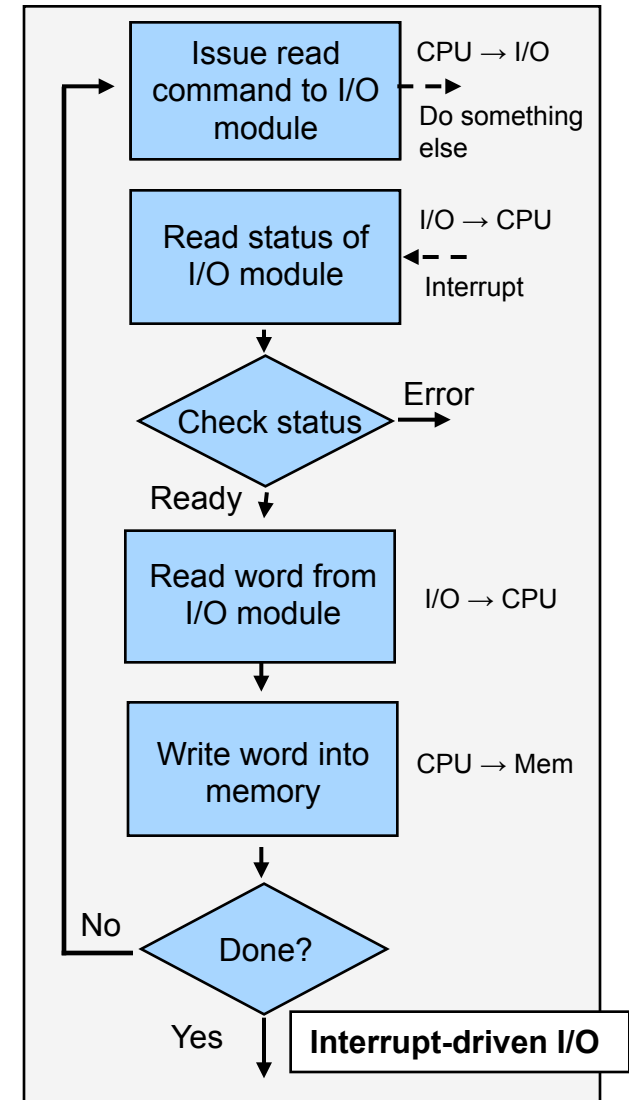
# Ways I/O can be done (OS take)

- 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(*printer_status_reg != READY);
scheduler();
```

## Interrupt service procedure

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



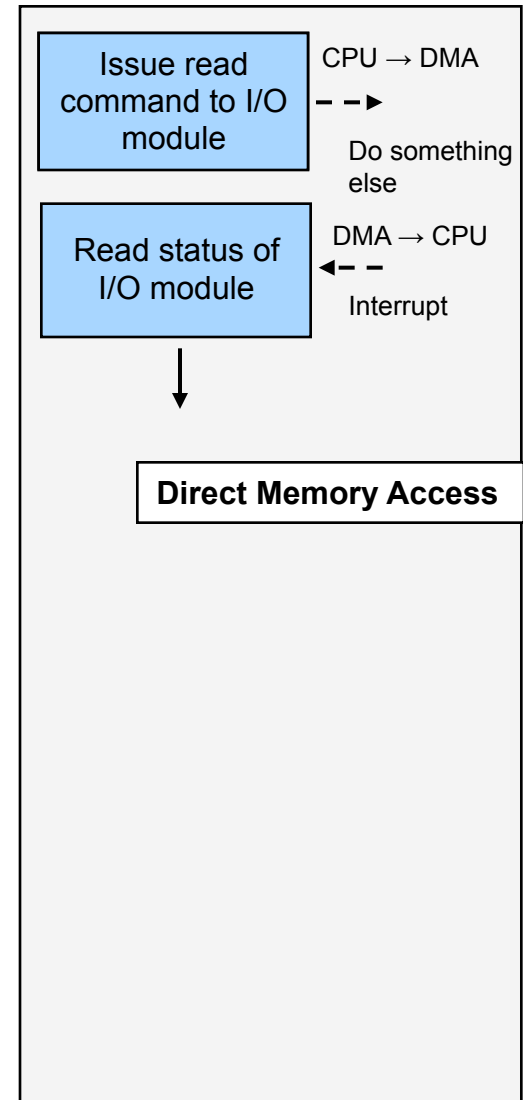
# Ways I/O can be done (OS take)

- 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();
```

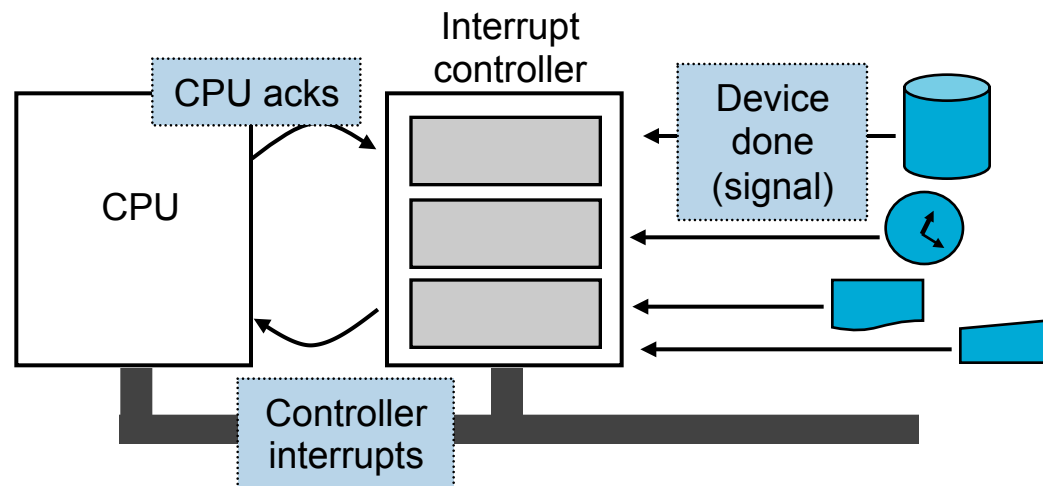
Interrupt service procedure

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



# Interrupts revisited

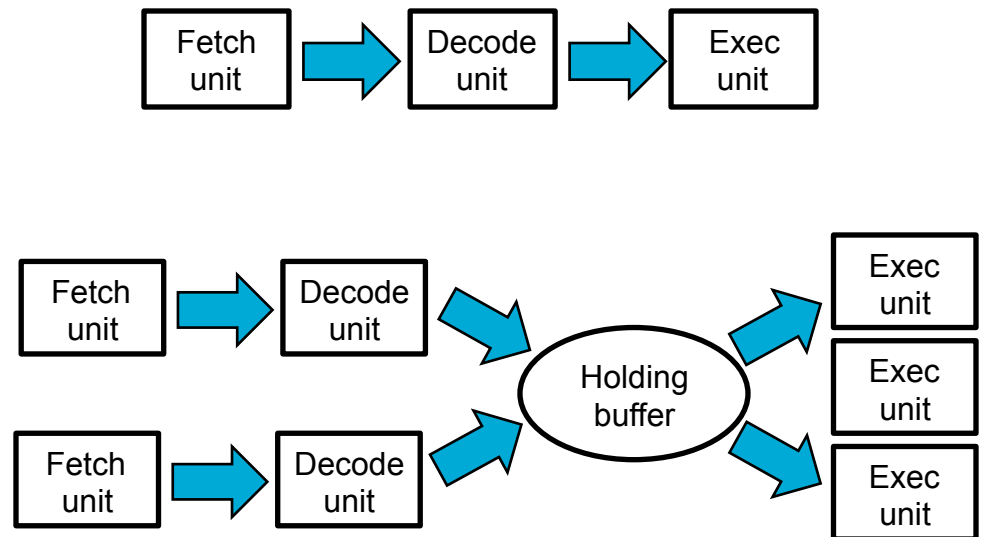
- When I/O is done – interrupt by asserting a signal on a bus line
- Interrupt controller, detects it and puts a # on address lines – index into interrupt vector
- Signal makes CPU stop what is doing and change PC to interrupt service procedure (found in vector)
- Interrupt service procedure ACK the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



# Interrupts revisited

- Not that simple ...
  - Where do you save the state?
  - Internal registers? Hold your ACK (avoid overwriting internal regs.)
  - In stack? You can get a page fault ... pinned page?
  - In kernel stack? Change to kernel mode \$\$\$
- Besides: pipelining, superscalar architectures, ...

Many instructions in various stages of execution; when the interrupt occurs the PC may not reflect the correct boundary bet/ executed/ nonexecuted instructions



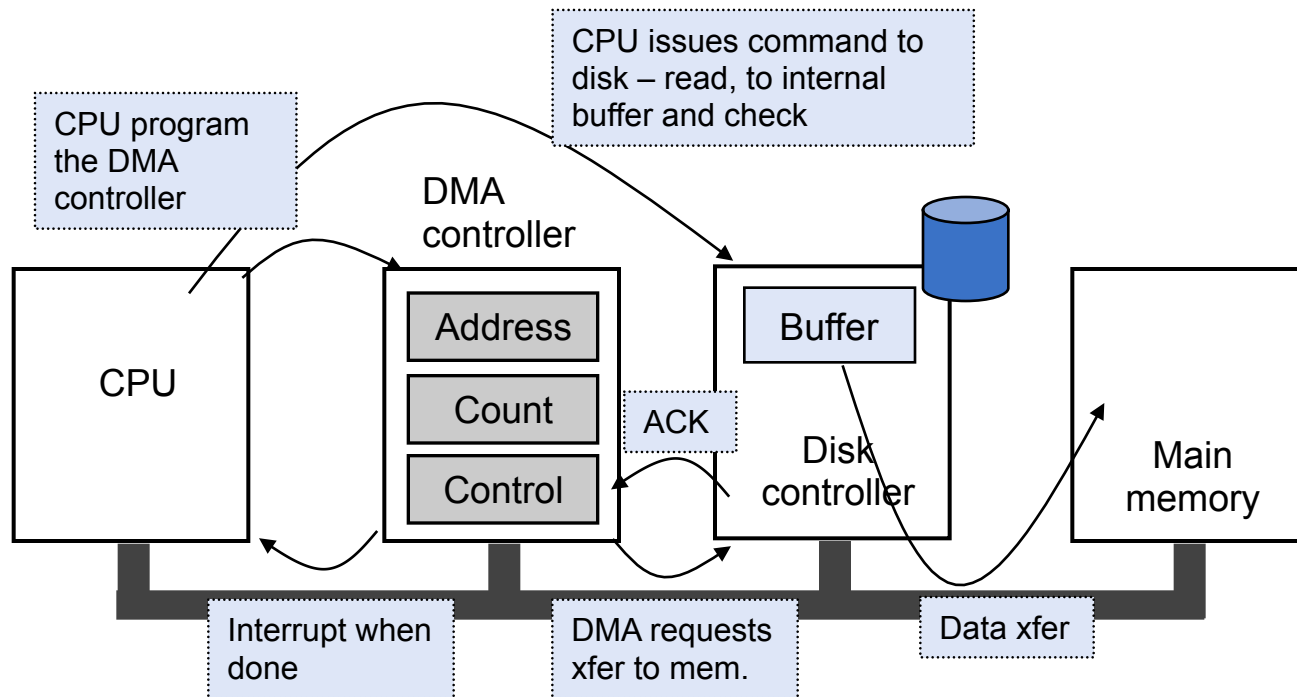
# Interrupts revisited

---

- 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
  4. The exec state of the instruction pointed by PC is known
- There is no prohibition on instructions beyond the one pointed by the PC from starting, but any changes to register or memory must be undone before the interrupt happens
- The tradeoff – complex OS or really complex interrupt logic within the CPU (design complexity & chip area)

# Direct Memory Access

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

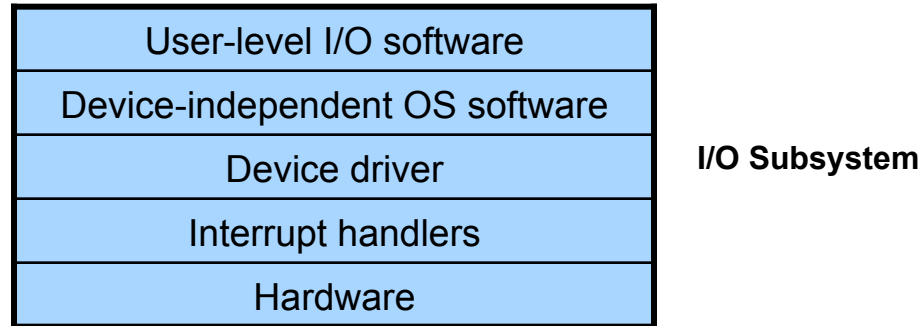


# Some details on DMA

- 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 – device controller occasionally steals the bus
  - 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*

# 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 (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 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, driver + unit, (kept by the i-node of the 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

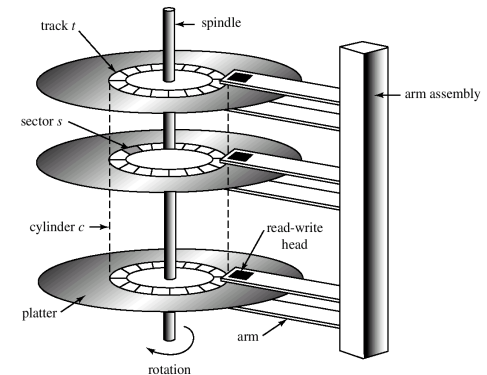
# User-space I/O software

---

- 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

# Disk – a concrete I/O device

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



20 years

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

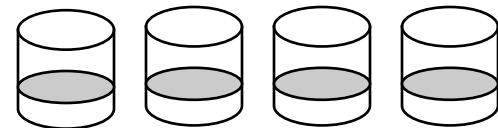
Note different rates of improvements on seek time, transfer rate and capacity

- 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

# RAIDs

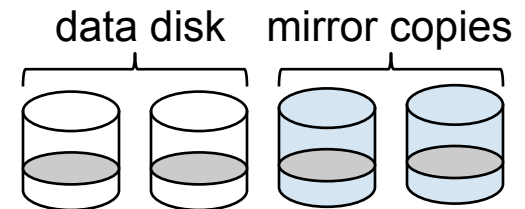
## Independent

- Redundant Array of ~~Inexpensive~~ 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
- RAID 0 – non-redundant disk array
  - Files are striped across disks, non redundant info
  - High read throughput
  - Best write throughput (nothing extra to write)
  - Worst reliability than with a single disk



# RAIDs

- But striping reduces reliability ( $n \times \text{MTBF}$ )
  - 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

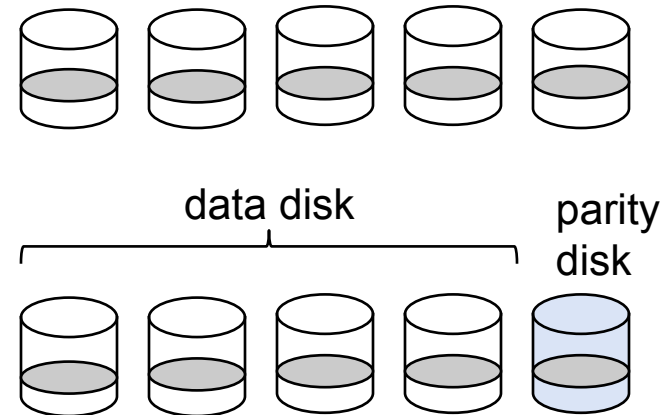
- Any single missing bit can be reconstructed
  - More complex schemes can detect/correct multiple bit errors

- 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, or

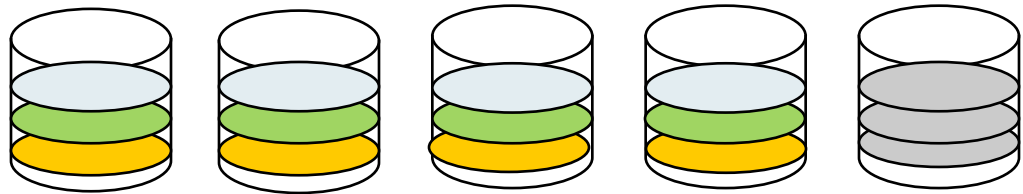
- A single parity bit is compute per word and written to the parity disk – RAID 3

- A read can access all data disks
- A write updates 1+ data disks and parity disk

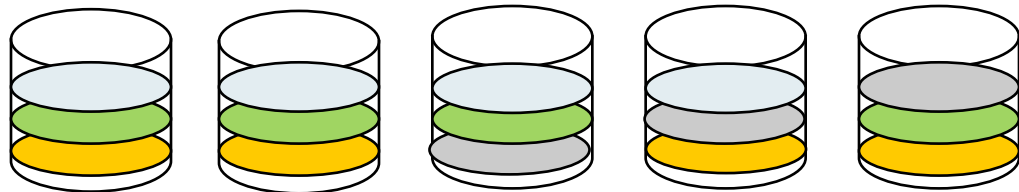


# RAIDs

- RAID 4 and 5
  - Work with strips (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)





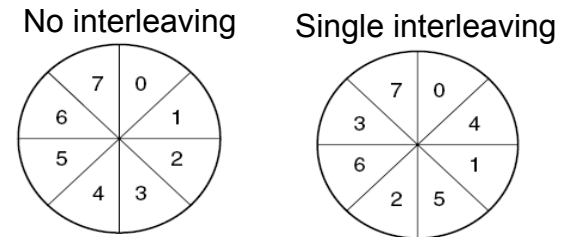
# RAIDs tradeoffs

---

- 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 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)
- Last step, high-level formatting
  - Boot block, free storage admin, root dir, empty file system



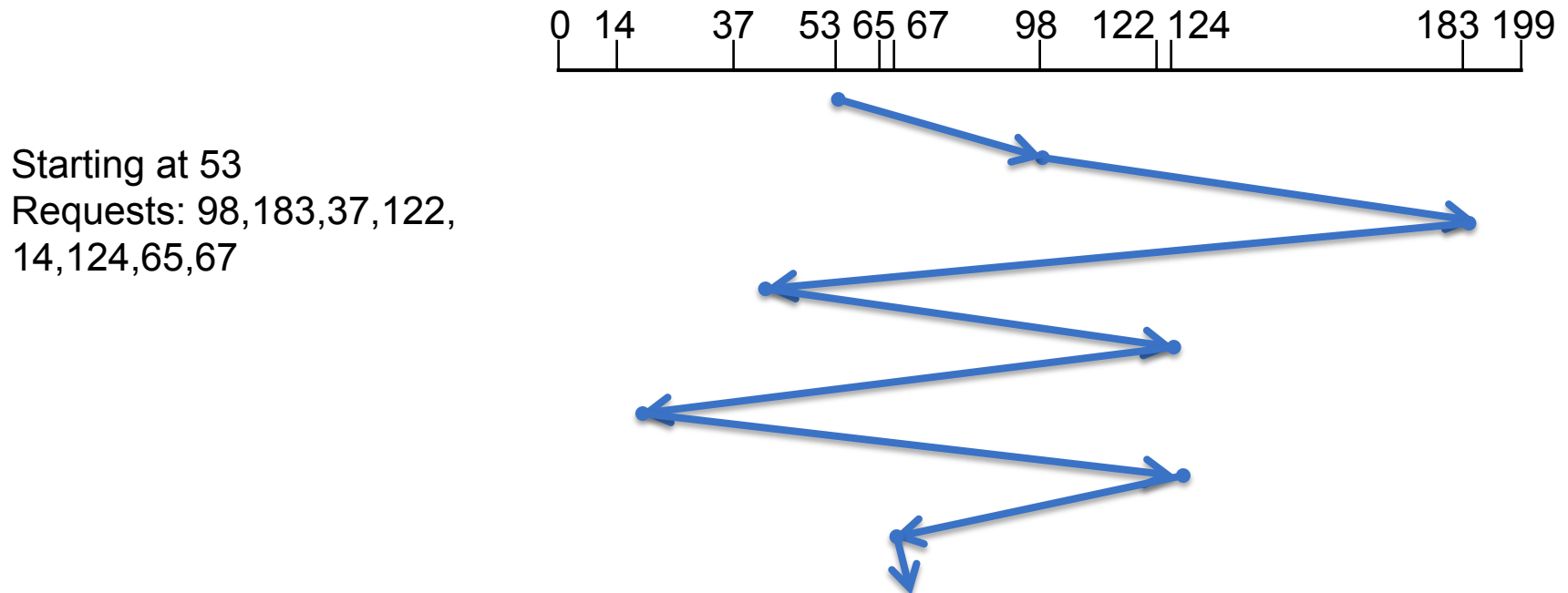
# Disk attachment

---

- Host-attached storage
  - Accessed through local I/O ports
  - Standards interfaces like SATA, SCSI, Fiber Channel
- Network-attached storage
  - Usually implemented as RAID
  - Clients access storage over the network, usually same data LAN, over NSF or CIFS (Windows)
  - Easy to access and share, slower performance
- Storage-area network
  - Private network connecting clients to storage units using storage protocols (rather than networking protocols)
    - Over FC or iSCSI
  - Multiple hosts and storage array can connect to the same SAN; storage can be dynamically allocated

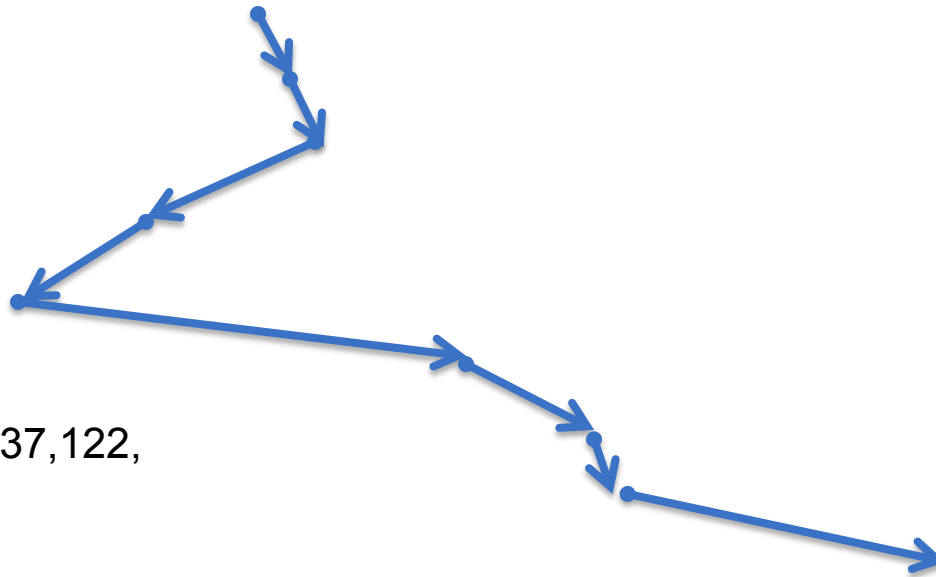
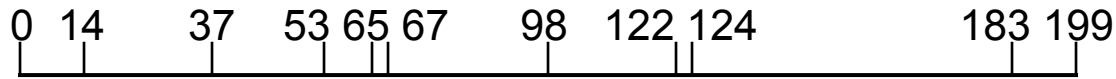
# Disk arm scheduling

- 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



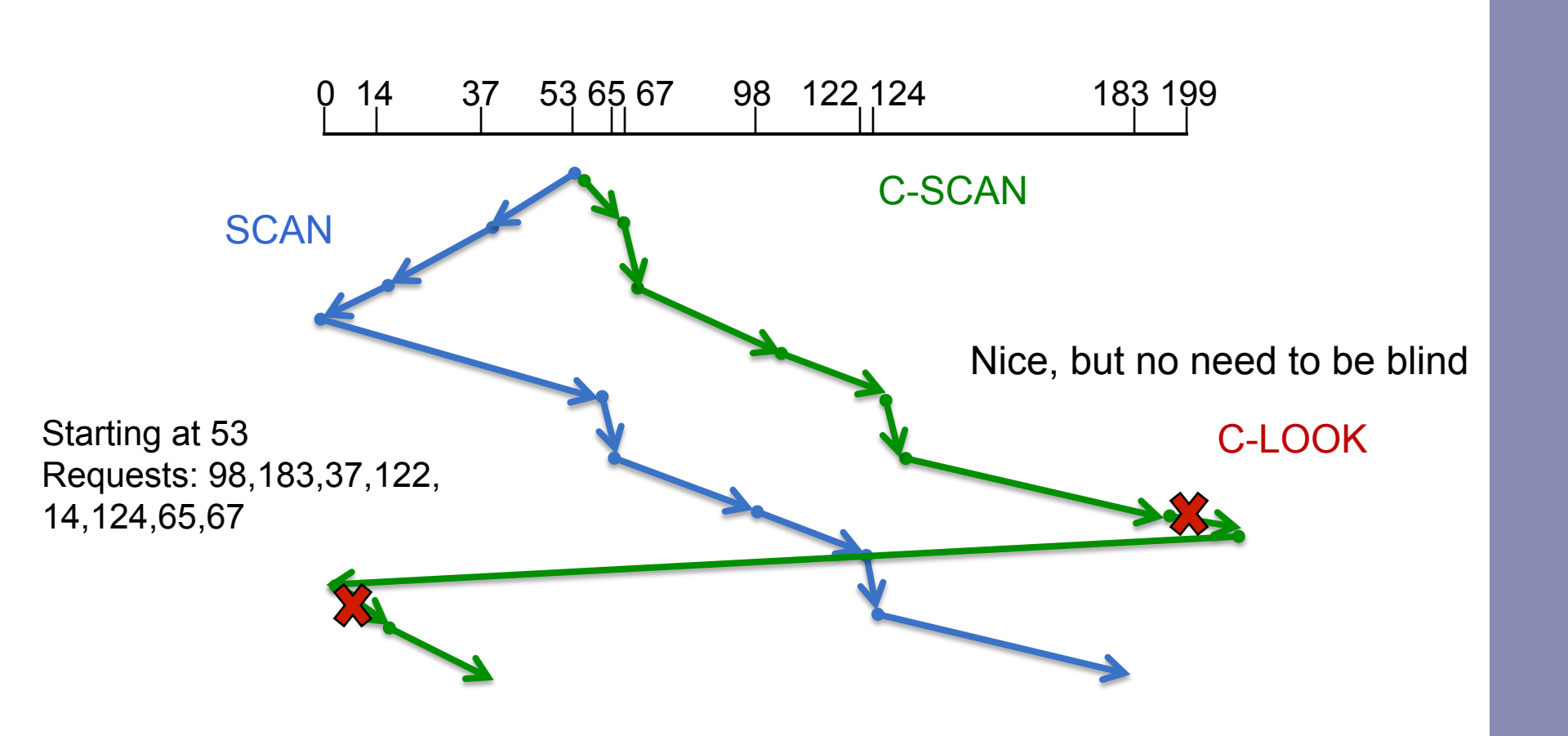
# SSTF

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



- As SJF, possible starvation

## SCAN, C-SCAN and C-LOOK



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

# Next time

---

- File systems interface and implementation