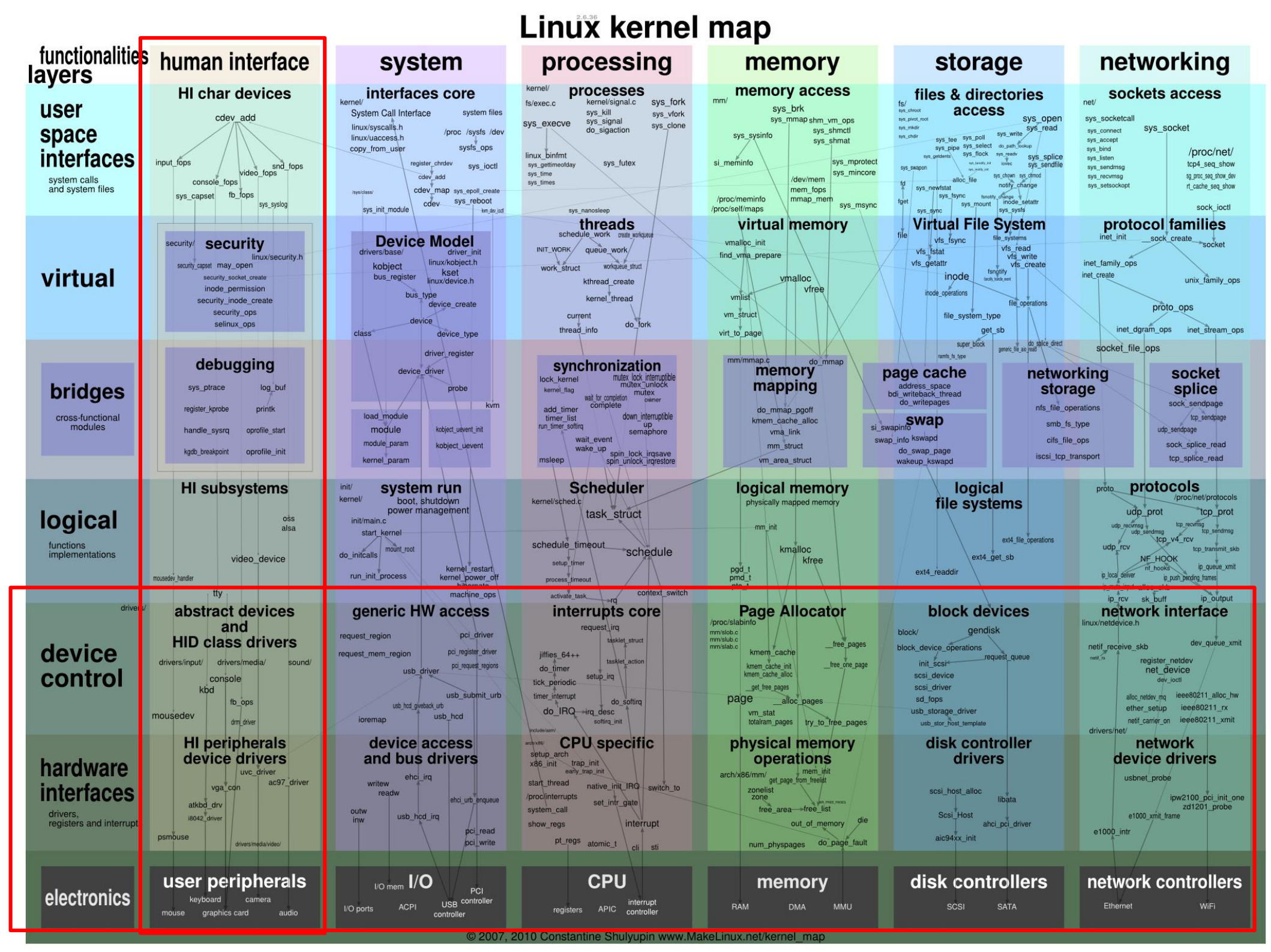
Linux kernel map



Devices

- There's more to a computer than CPU and memory
- I/O Devices:
  - Store information
  - Communicate to the outside world
- Role of the OS ^ Control I/O devices
  - Range of types and speeds
  - OS concern is with the interface between the hardware and the user

Block Devices

- Store information in blocks
  - Blocks addressed/accessed individually
  - 512 – 32k (common sizes)
  - Block is a unit of transfer
- HDD, CD-ROM, flash drives
- Typically have a seek operation
  - Random access

Character Device

- Deliver or accept a stream of characters
  - No block structure
  - Not addressable
  - No seek
- Printer, network interface, mouse…

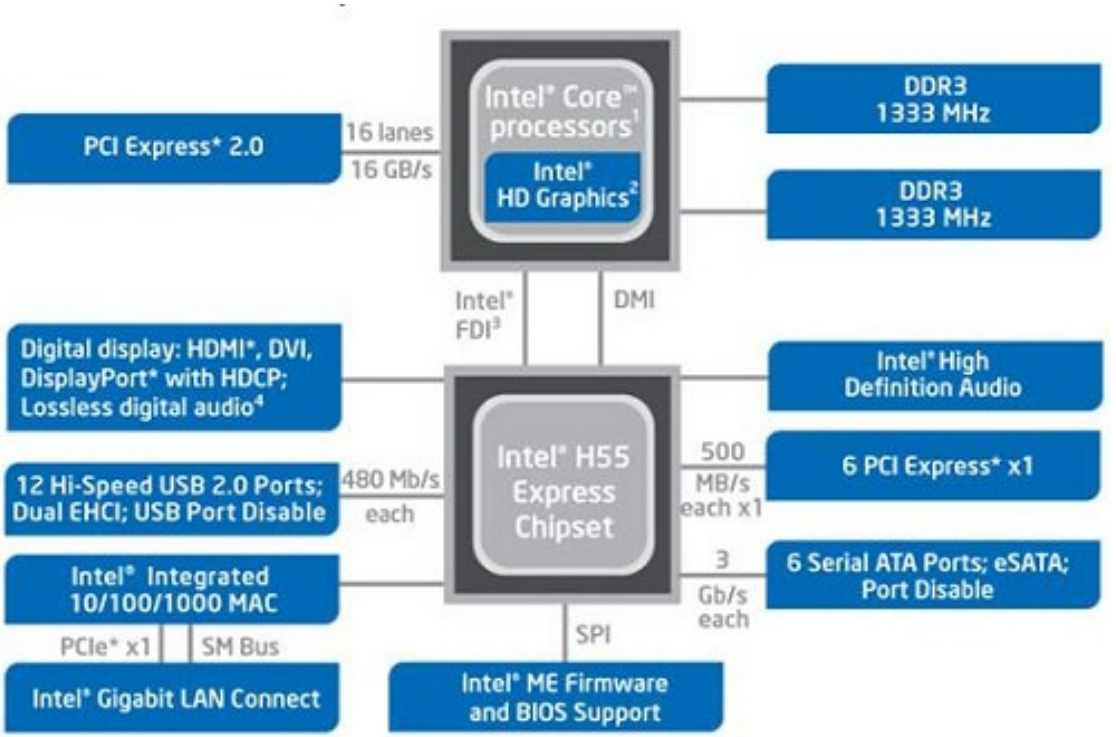
- Not all devices are block or character
  - Timers, clocks

Range of Speeds

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus	2.5 GB/sec
SONET OC-768 network	5 GB/sec

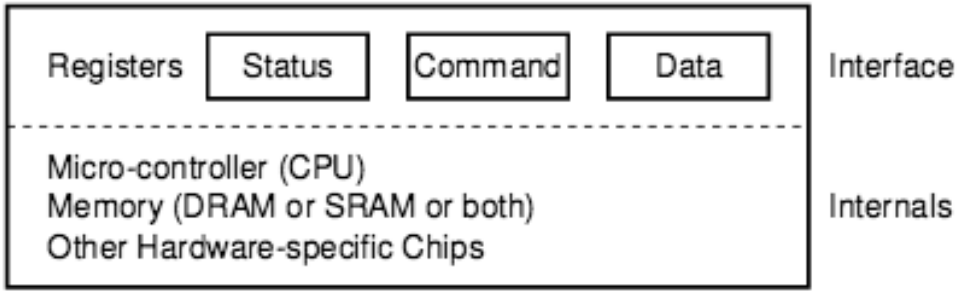
Architecture

- Northbridge
  - Fast devices
    - CPU
    - Memory
    - Graphics
- Southbridge
  - I/O Controller Hub
  - Slow devices



Canonical Device

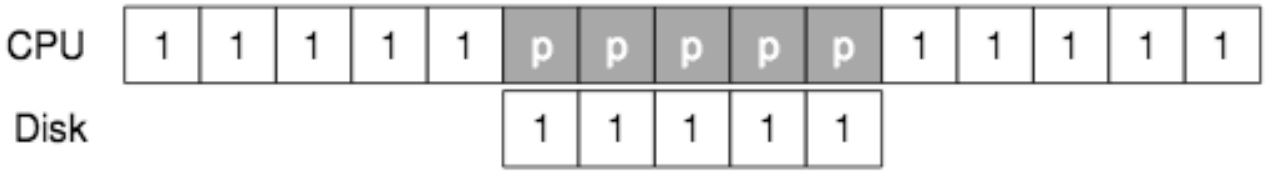
- Status registers
  - read to get status
- Command registers
  - write to perform tasks
- Data registers
  - read/write to pass data from/to device
- Typical interaction (programmed I/O):



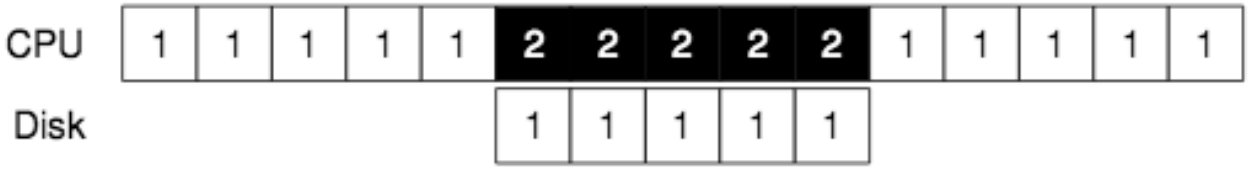

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

Interrupts

- Programmed I/O (polling) may waste CPU cycles



- Interrupts allow processes to overlap



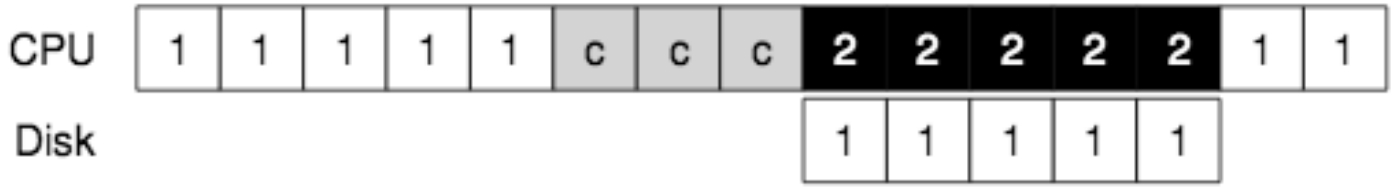
Polling or Interrupts

Polling is like picking up a phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring

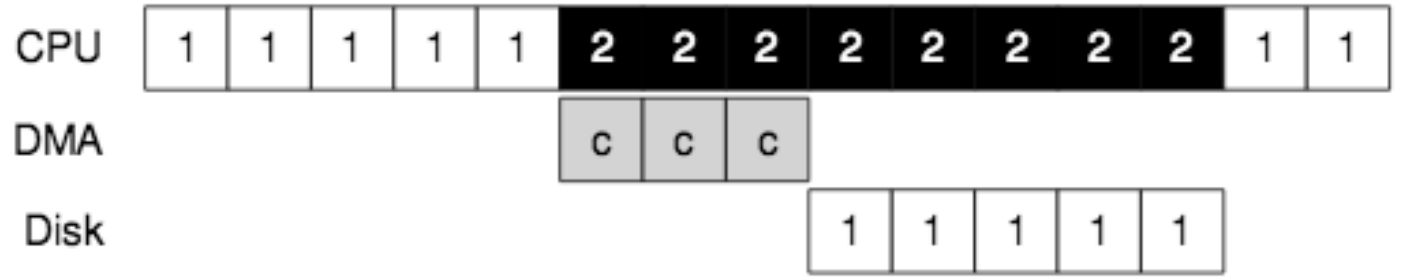
- Polling may be better if processor has to respond to an event ASAP (real-time?)
  - Or if device is fast
- Interrupts are better if the processor has other work to do and response time isn't critical
  - Could slow down the system, handling interrupts is expensive

DMA

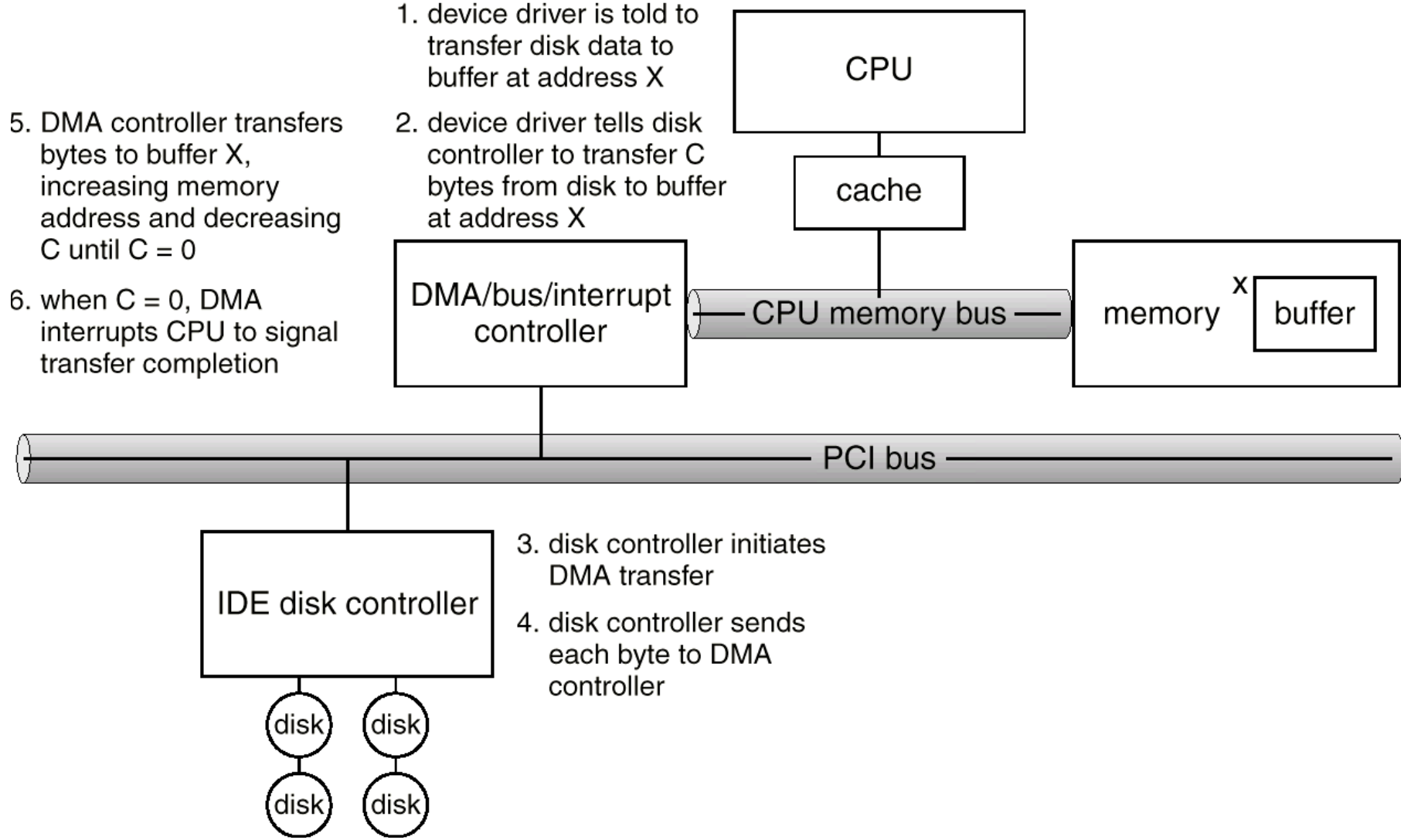
- Transferring data may waste CPU time



- Direct Memory Access devices handle data transfers with minimal CPU interaction

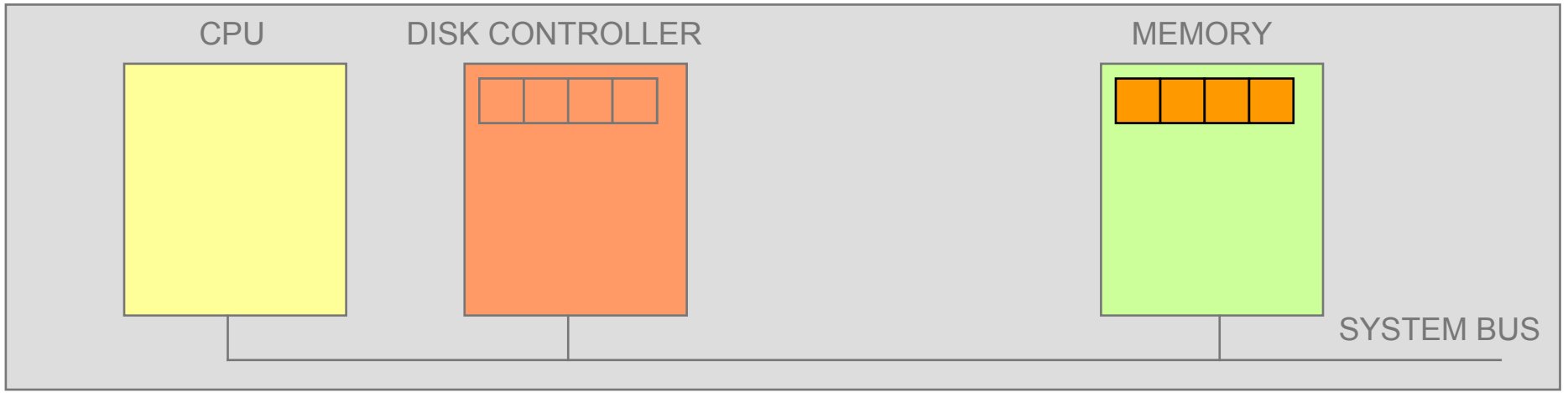


DMA Overview

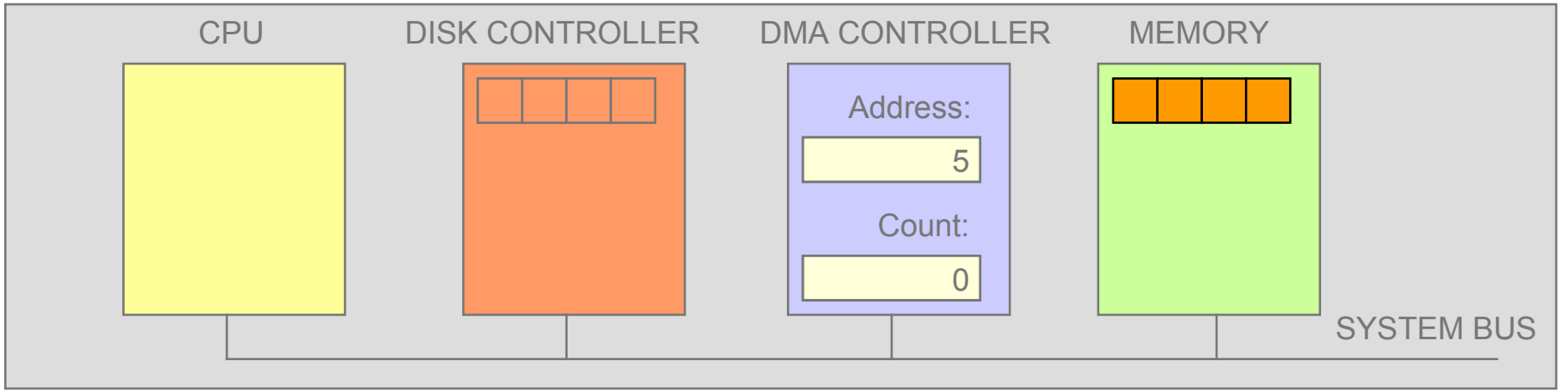


DMA Transfer Example

CPU controlled transfer



DMA controlled transfer



DMA or not?

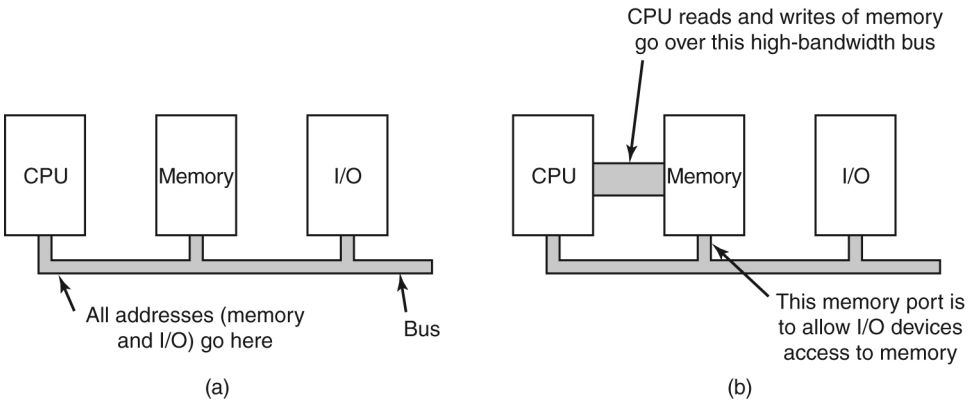
- Main CPU is much faster than DMA controller… so, no
  - If CPU has nothing to do, waiting for DMA is wasteful
- Excluding DMA controllers reduces HW cost

Memory-Mapped I/O

- Alternative to explicit device I/O
  - (e.g. using *in* and *out* in asm)
- Each device gets a portion of real address space
  - Access limited to kernel
  - Reads/writes to this memory interpreted as commands by I/O device
- Device control addressable by high-level language (C/C++)
- No special protection needed
  - Don't map device memory to user processes
- Existing instructions repurposed for I/O control

Memory-Mapped I/O (2)

- Potential problem with page caching
  - e.g. while (status!=0)
    - What if status is cached?
  - Have to have ability to disable caching per page
- All controllers must examine all memory accesses (problem?)
  - Intel uses PCI bridge to filter addresses that fall w/in non-addressable range

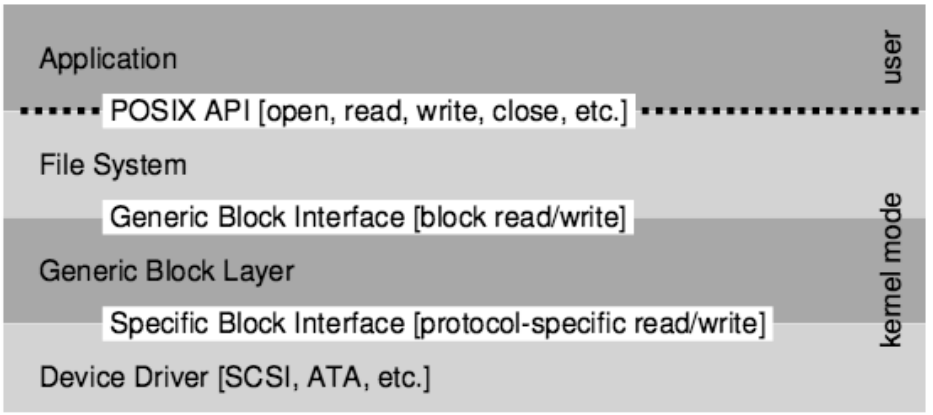


Device Driver

- Software
- Communicates with a controller
- Provides an abstracted interface for applications or operating systems
- OS and device specific
- 70% of OS code is device drivers
  - most of this code is inactive
  - primarily responsible for kernel crashes (written by amateurs)

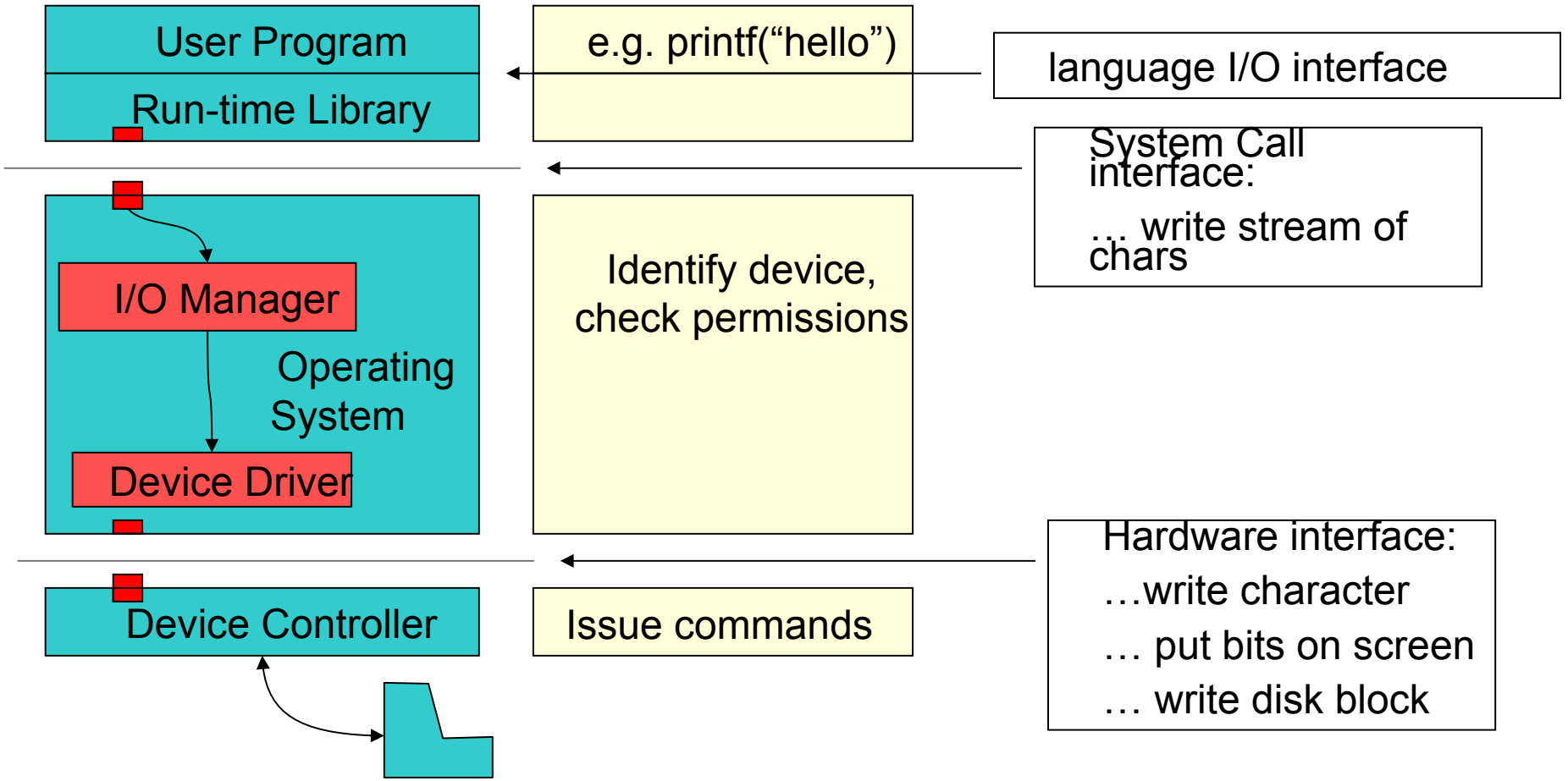
Device Driver Role

- Example: file systems
  - FS only issues block read/write requests
  - block layer routes them to correct device driver
  - device driver issues actual requests to hardware



  - Downside: generic interfaces result in some specialized device capabilities becoming useless

Overview


User Program
Run-time Library
e.g. printf("hello")
language I/O interface
System Call interface:
… write stream of chars
I/O Manager
Operating System
Device Driver
Identify device, check permissions
Hardware interface:
…write character
… put bits on screen
… write disk block
Device Controller
Issue commands

Example: IDE Disk Driver (Device)

```
Control Register:
  Address 0x3F6 = 0x80 (0000 1RE0): R=reset, E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7       6     5       4     3     2     1      0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1)
    7       6     5       4     3     2     1     0
   BBK     UNC    MC    IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

Example: IDE Disk Driver (Protocol)

- Wait for drive to become ready
  - Read register 0x1F7 until it shows not busy and READY
- Write to command registers
  - sector count, logical block address, drive number
  - write to 0x1F2-0x1F6
- Start I/O
  - write READ/WRITE to 0x1F7
- Data Transfer (for writes)
  - wait for status READY and DRQ (drive request for data)
  - write to port 0x1F0
- Handle Interrupts
  - one interrupt per sector or one interrupt once entire transfer is complete
- Error Handling
  - check status register after each operation, read status register

Example: IDE Disk Driver (Code)

- Wait for drive to become ready
  - Read register 0x1F7 until it shows not busy and READY

```
static int ide_wait_ready() {
  while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
    ;                                  // loop until drive isn't busy
}
```

Example: IDE Disk Driver (Code)

- Write to command registers
- Start I/O
- Data Transfer (for writes)

```
static void ide_start_request(struct buf *b) {
  ide_wait_ready();
  outb(0x3f6, 0);                    // generate interrupt
  outb(0x1f2, 1);                    // how many sectors?
  outb(0x1f3, b->sector & 0xff);     // LBA goes here ...
  outb(0x1f4, (b->sector >> 8) & 0xff);   // ... and here
  outb(0x1f5, (b->sector >> 16) & 0xff);  // ... and here!
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);      // this is a WRITE
    outsl(0x1f0, b->data, 512/4);    // transfer data too!
  } else {
    outb(0x1f7, IDE_CMD_READ);       // this is a READ (no data)
  }
}
```

Example: IDE Disk Driver (Code)

- Queue requests (not in the protocol, but necessary, as the disk may be busy)

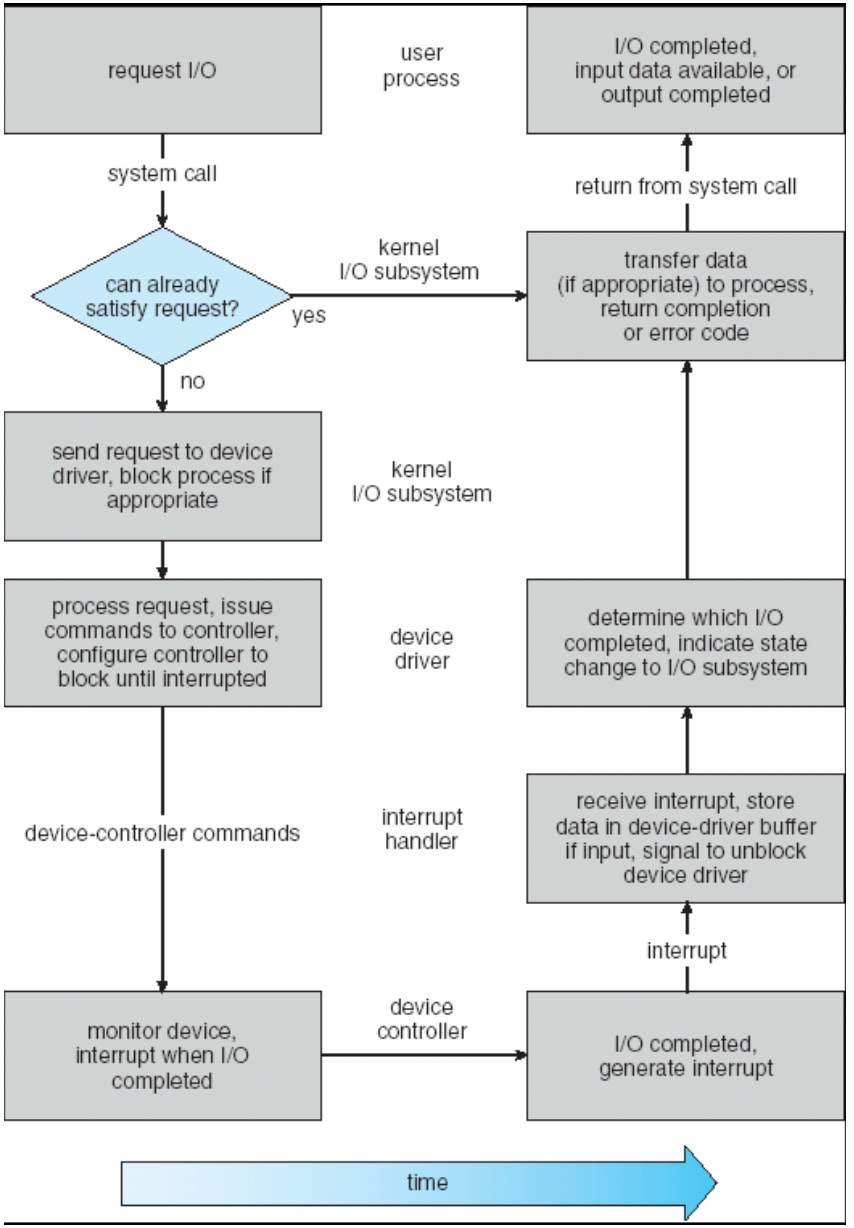
```
void ide_rw(struct buf *b) {
  acquire(&ide_lock);
  for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ;                                  // walk queue
  *pp = b;                             // add request to end
  if (ide_queue == b)                  // if q is empty
    ide_start_request(b);              // send req to disk
  while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &ide_lock);               // wait for completion
  release(&ide_lock);
}
```

Example: IDE Disk Driver (Code)

- Handle interrupts
  - invoke ide_intr when interrupt happens, read data from device (if req is a read)
  - wake the process waiting for I/O
  - launch next I/O request

```
void ide_intr() {
  struct buf *b;
  acquire(&ide_lock);
  if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
    insl(0x1f0, b->data, 512/4);   // if READ: get data
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);                       // wake waiting process
  if ((ide_queue = b->qnext) != 0) // start next request
    ide_start_request(ide_queue);  // (if one exists)
  release(&ide_lock);
}
```

Life Cycle of an I/O Request



Hard Drive

- On the outside, a hard drive looks like this

Taken from "How Hard Disks Work"
http://computer.howstuffworks.com/hard-disk2.htm

Inside the Hard Drive

- If we take the cover off, we see that there actually is a “hard disk” inside

Taken from “How Hard Disks Work”
http://computer.howstuffworks.com/hard-disk2.htm

Hard Drive Platters

- A hard drive usually contains multiple disks, called platters
- These spin at thousands of RPM (5400, 7200, etc)

Taken from "How Hard Disks Work"
http://computer.howstuffworks.com/hard-disk2.htm

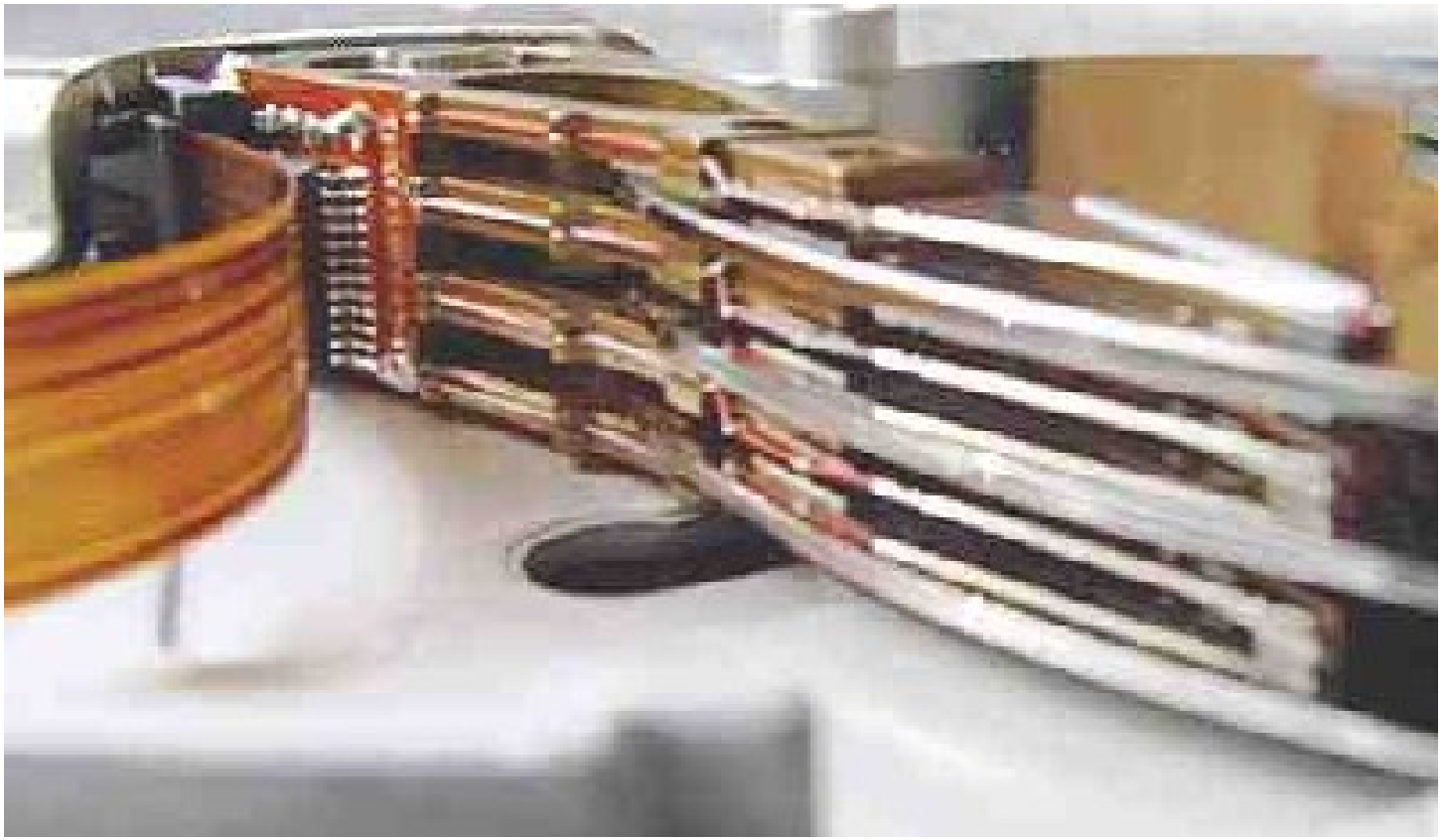
Hard Drive Arms

- Information is written to and read from the platters by the read/write heads on the disk arm

Taken from "How Hard Disks Work"
http://computer.howstuffworks.com/hard-disk2.htm

Platter Surface

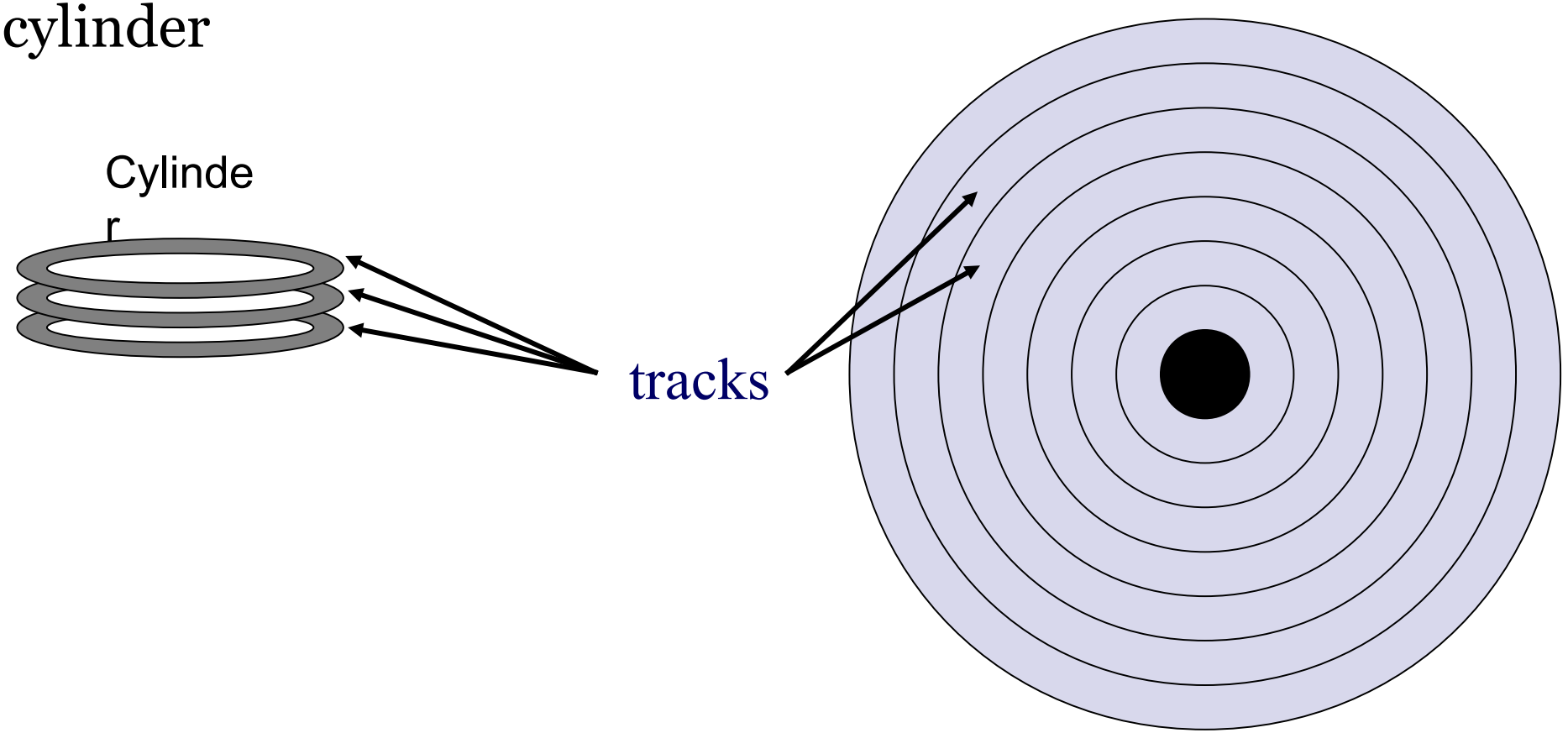
- Both sides of each platter store information
- Each side of platter is called a surface
- Each surface has its own read/write head



Taken from "How Hard Disks Work"
http://computer.howstuffworks.com/hard-disk2.htm

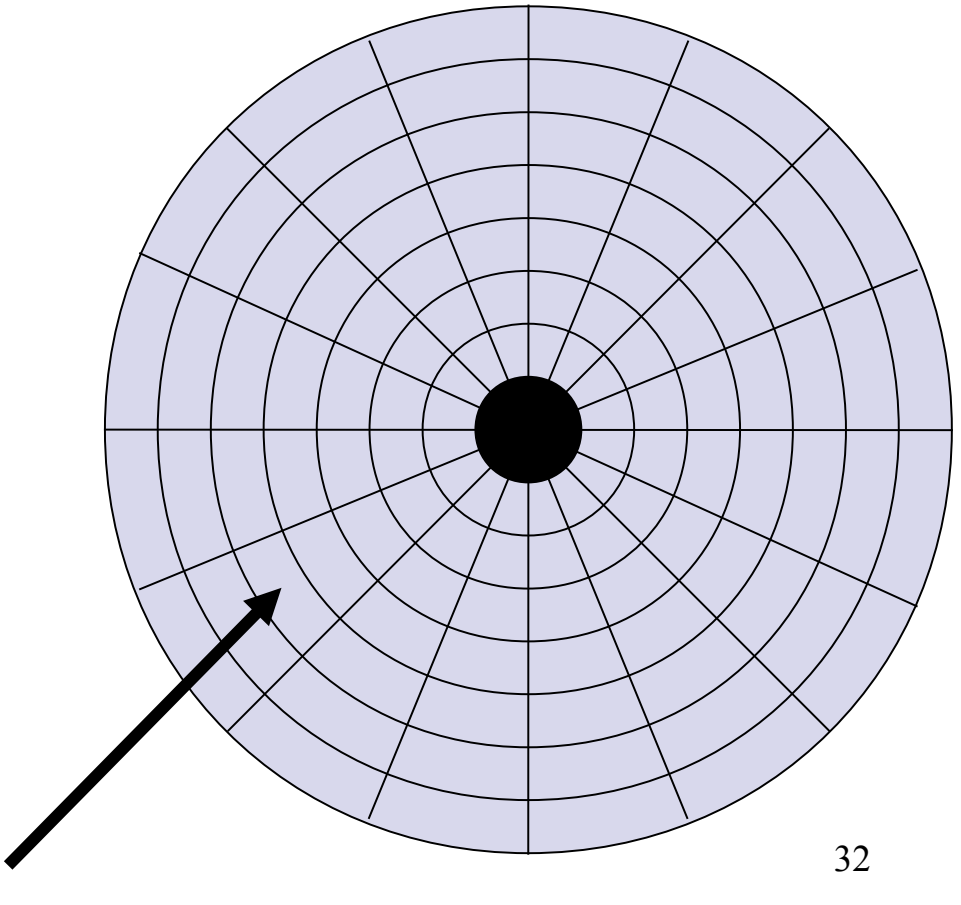
Disk Tracks & Cylinders

- Each surface is divided by concentric circles, creating tracks
- Matching tracks across surfaces are collectively called a cylinder



Disk Sectors

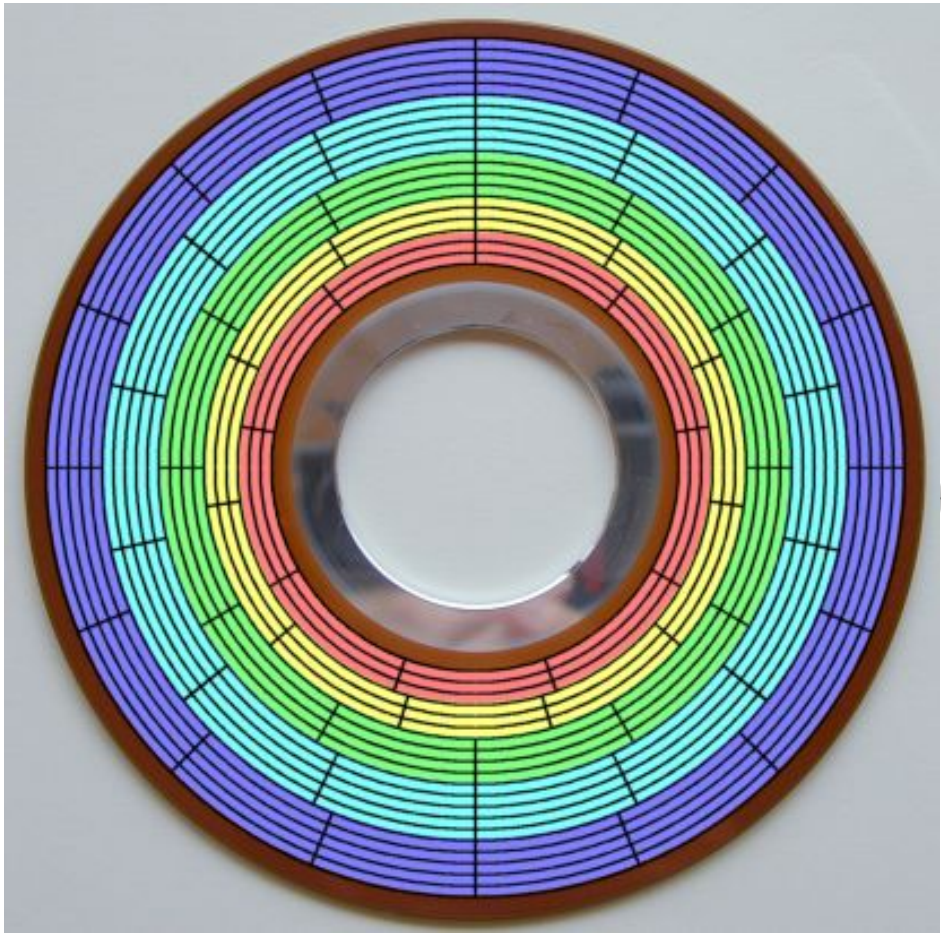
- These tracks are further divided into sectors
- A sector is the smallest unit of data transfer to or from the disk
- Most modern hard drives have 512 byte sectors
- CD-ROM sectors are 2048 bytes
- Gee, those outer sectors look bigger…



Disk Zones

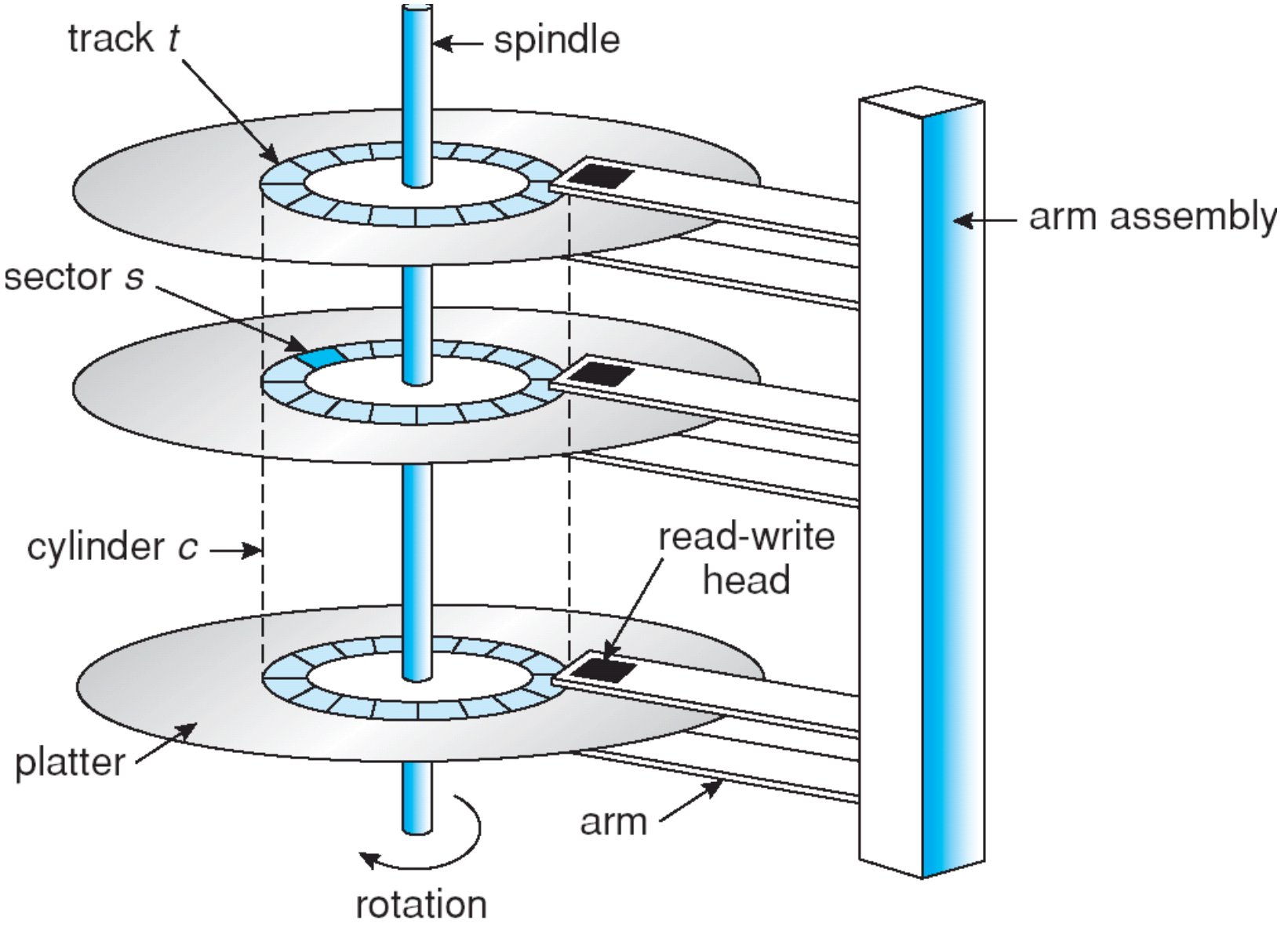
- Does this mean that sectors on the outside of a surface are larger than those on the inside?

- Modern hard drives fix this with zoned bit recording



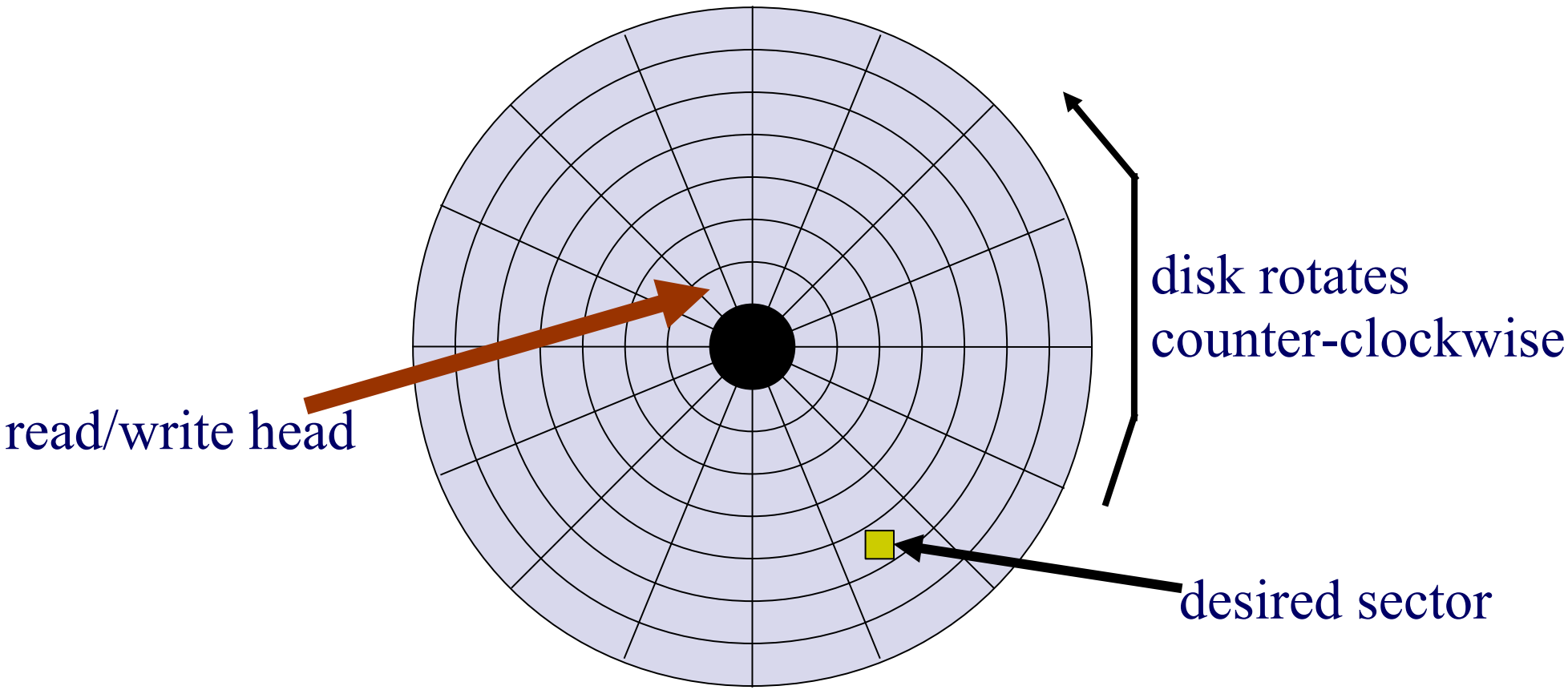
Taken from "Reference Guide – Hard Disk Drives"
http://www.storagereview.com/map/lm.cgi/zone

Anatomy of Hard Disk



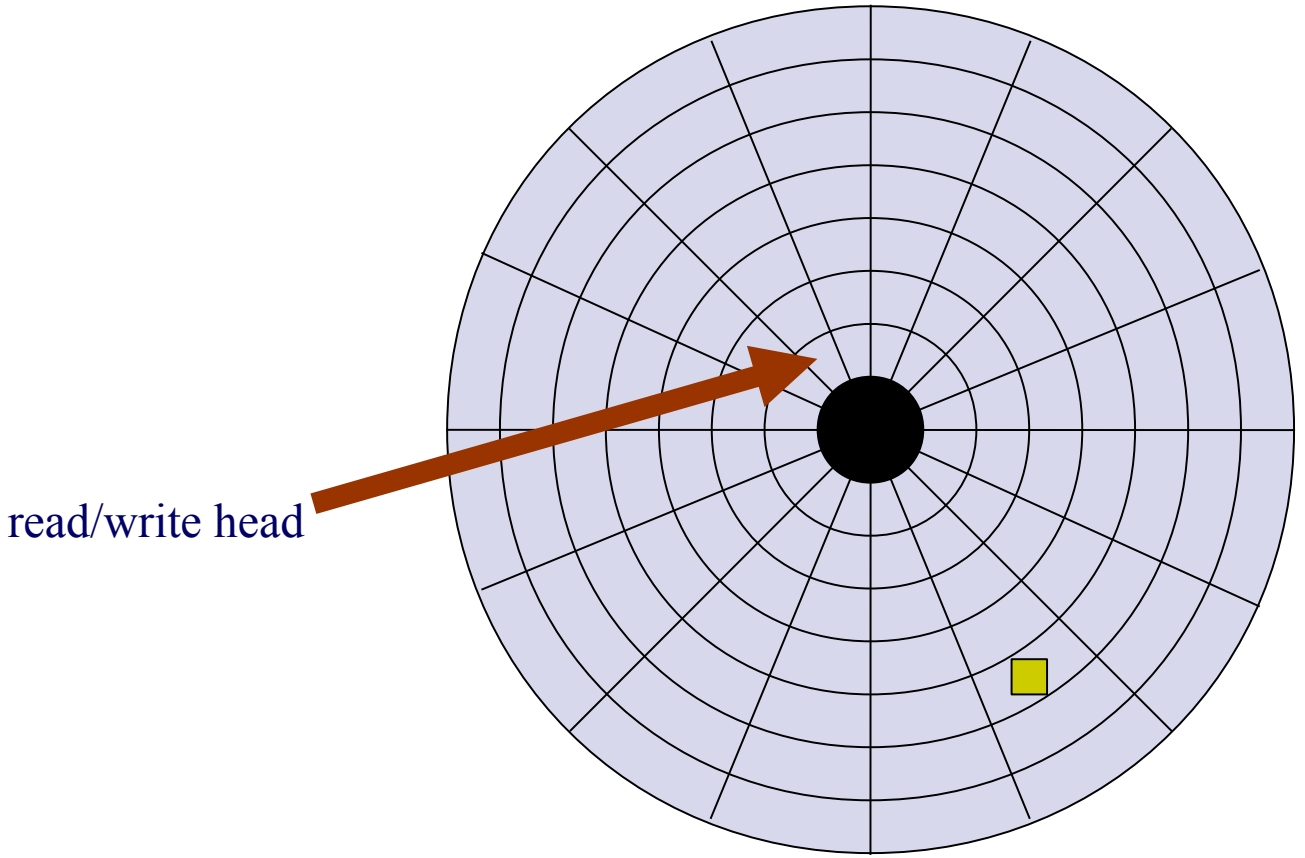
Disk Access

- Let's read in a sector from the disk



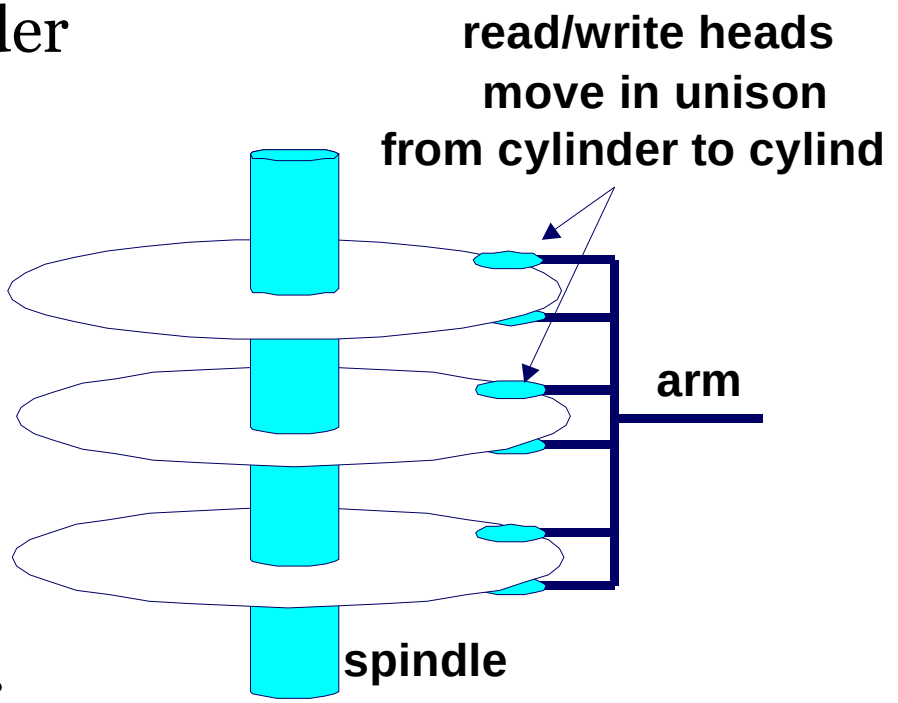
Reading a Sector From The Disk

- We need to do two things to transfer a sector
  - Move the read/write head to the appropriate track (seek)
  - Wait until the desired sector spins around



Cheap Access Within A Cylinder

- Heads on single arm
  - All heads always on same cylinder
- Switching heads is “cheap”
  - Deactive head 3
  - Activate head 4
  - Wait for 1st sector header
- Optimal transfer rate
  - Transfer all sectors on a track
  - Transfer all tracks on a cylinder
  - Then move elsewhere



Disk Examples (Summarized Specs)

	Seagate Cheetah	IBM Ultrastar 72ZX
Capacity, Interface & Configuration		
Capacity	600GB	73.4
Interface	6-Gb/s SAS	Ultra160 SCSI
Platters / Heads	2 / 4 ?	11/22
Bytes per sector	?	512-528
Performance		
Max Internal transfer rate (Mbytes/sec)	600MB/s?	53
Max external transfer rate (Mbytes/sec)	600MB/s	160
Avg Transfer rate(Mbytes/sec)		22.1-37.4
Disk cache (Kbytes)	16MB	16,384
Average seek (read/write) (msec)	3.4/3.9	5.3
Average rotational latency (msec)	2	2.99
Spindle speed (RPM)	15,000	10,000

Internal transfer rate: between platters and disk's integrated controller
External transfer rate: between disk and the rest of the PC

Disk Performance

- Seek
  - Position heads over cylinder, typically 5.3 - 8 ms
- Rotational delay
  - Wait for a sector to rotate underneath the heads
  - Typically 8.3 - 6.0 ms (7,200 – 10,000RPM) or ½ rotation takes 4.15-3ms
- Transfer rate
  - Average transfer bandwidth (15-37 Mbytes/sec)

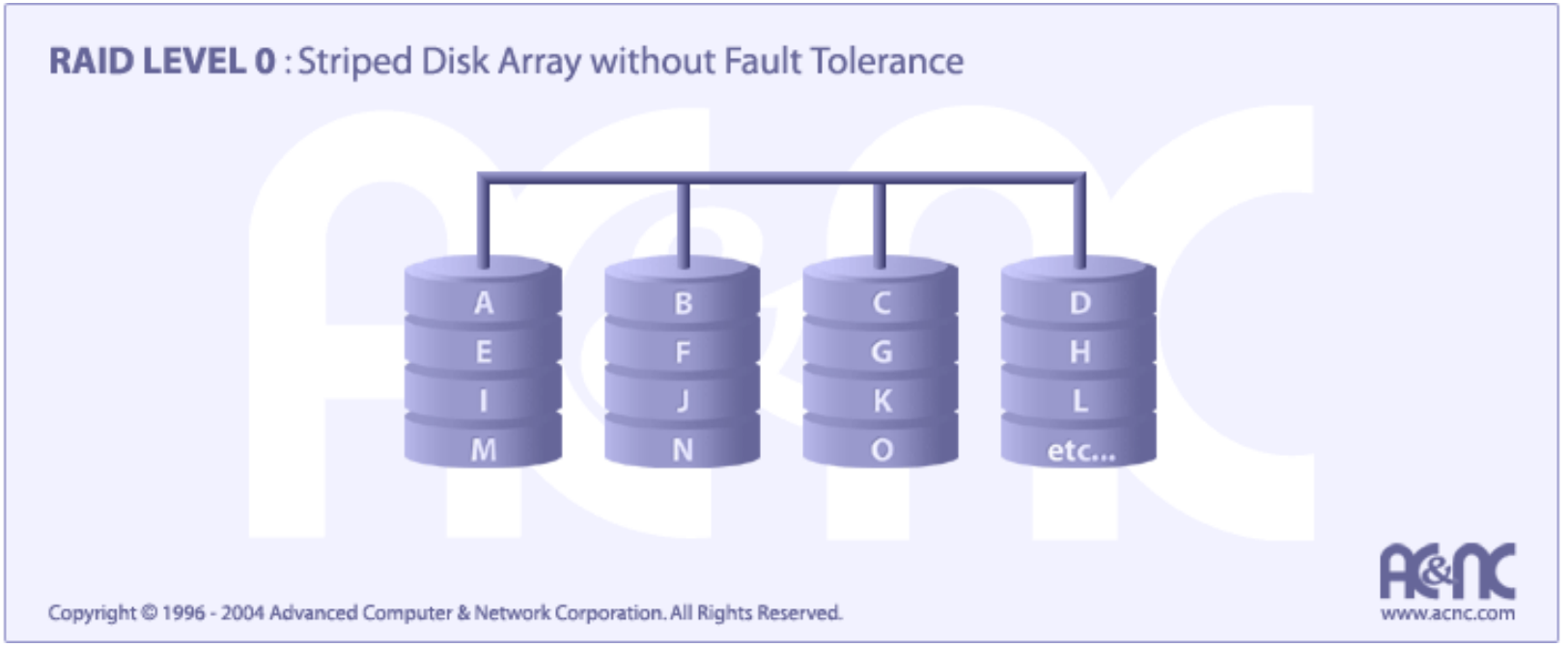
RAID

- Redundant Array of Inexpensive Disks

- Two motivations
  - Operating in parallel can increase disk throughput
  - Redundancy can increase reliability

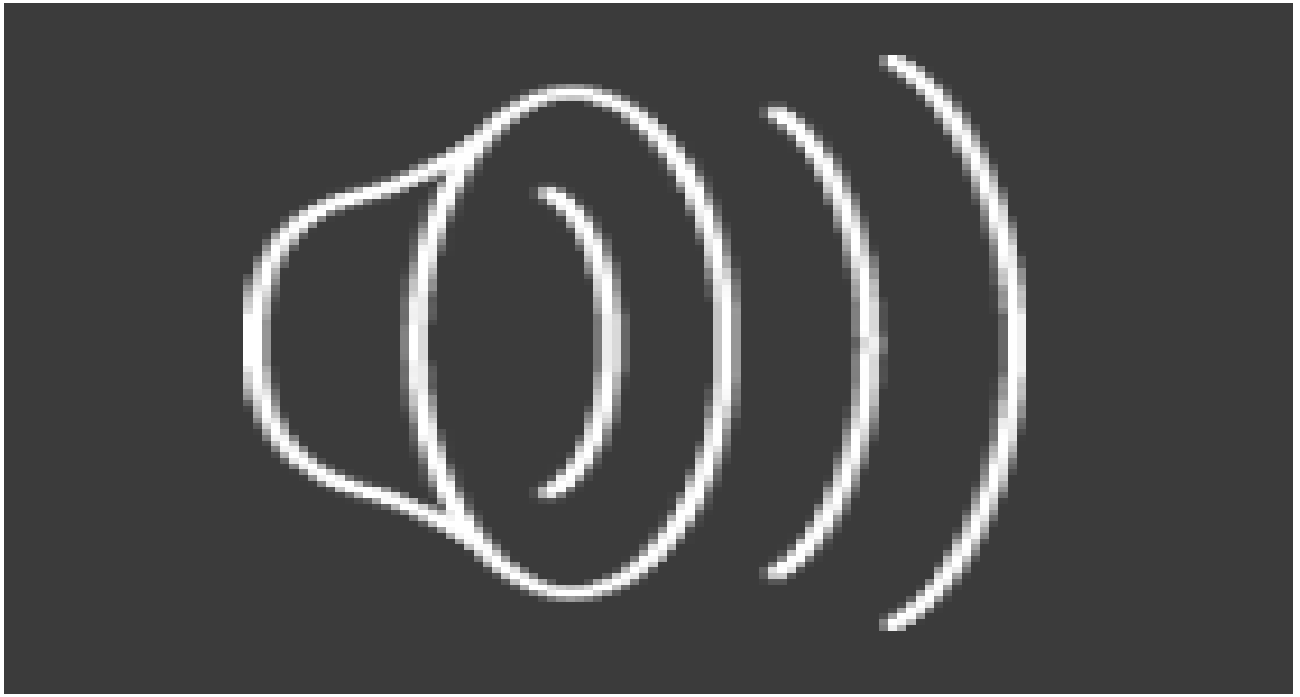
http://www.acnc.com/

RAID 0



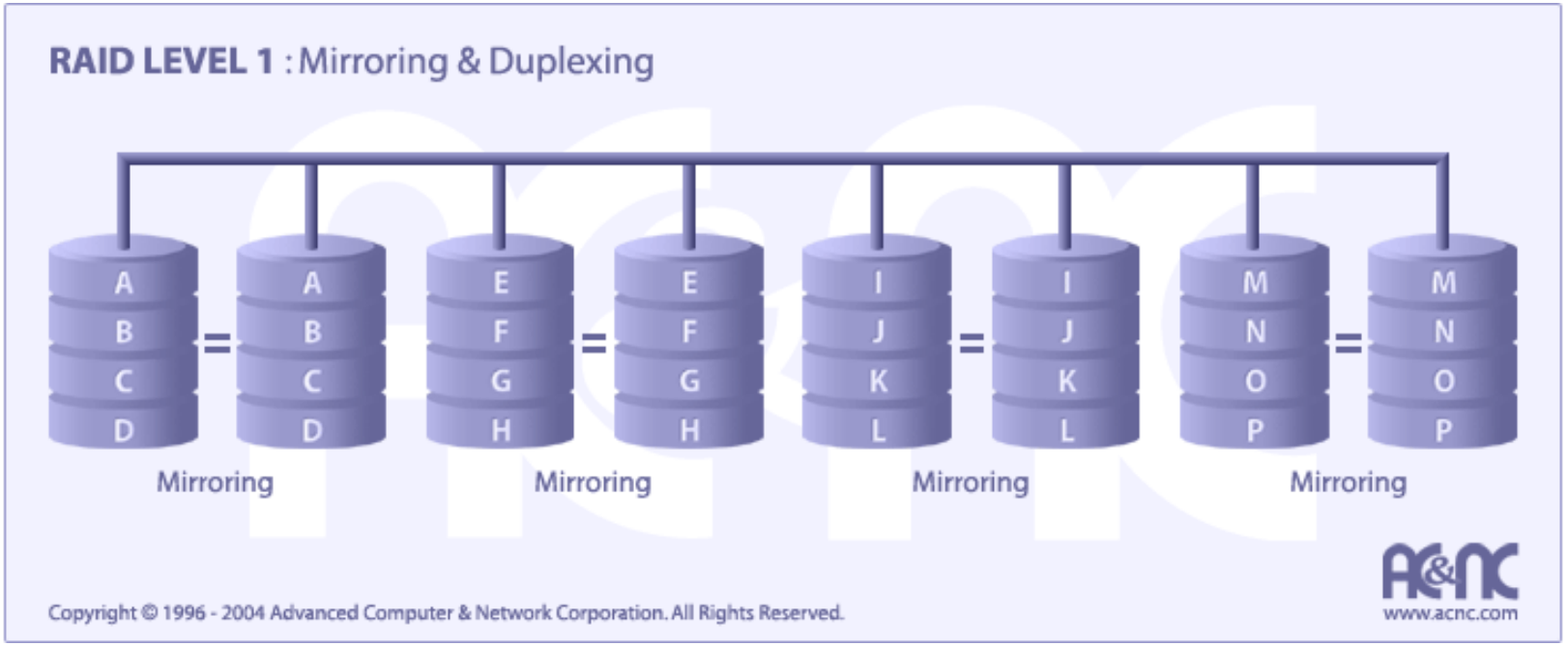
- Data striped across disks
- I/O performance NumDisksX (possibly)
- Not a "True" RAID because it is NOT fault-tolerant

RAID 0



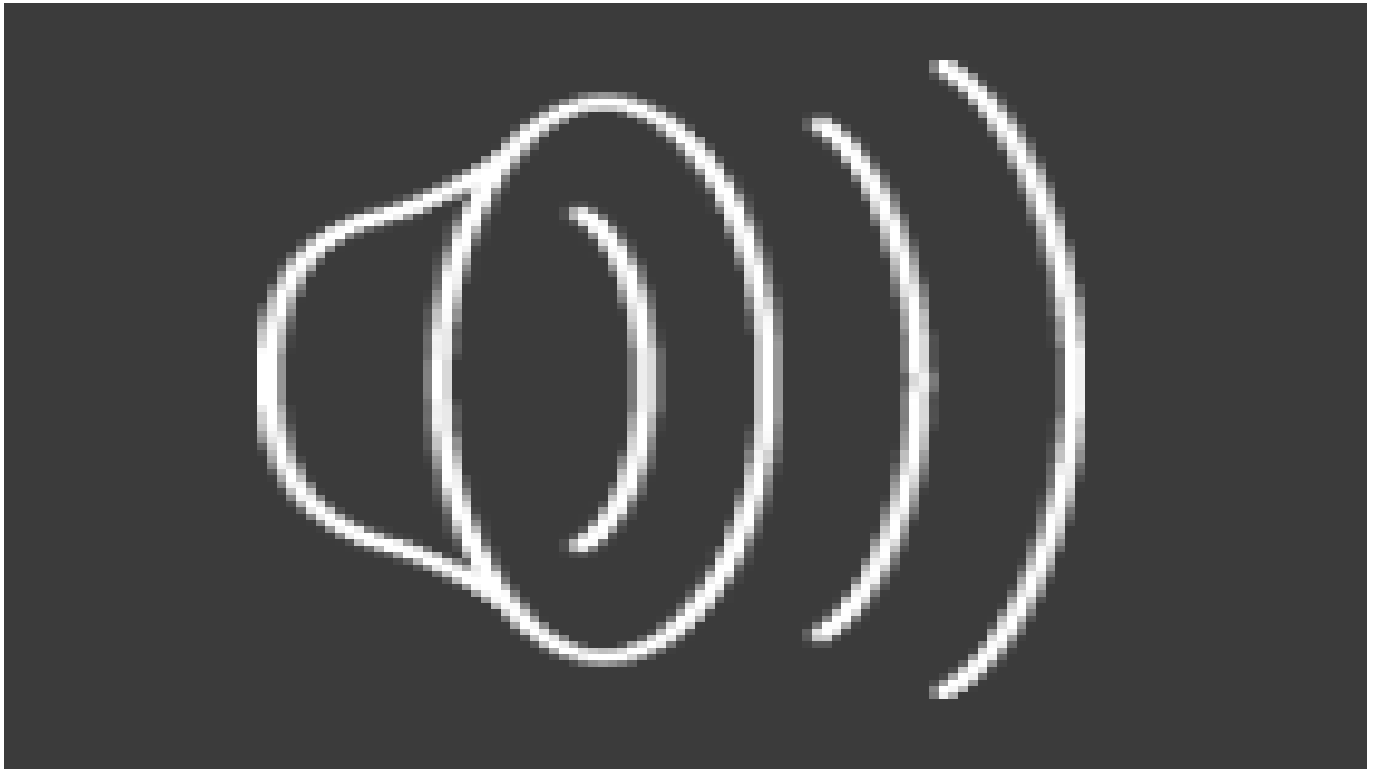
- Data striped across disks
- I/O performance NumDisksX (possibly)
- Not a "True" RAID because it is NOT fault-tolerant

RAID 1



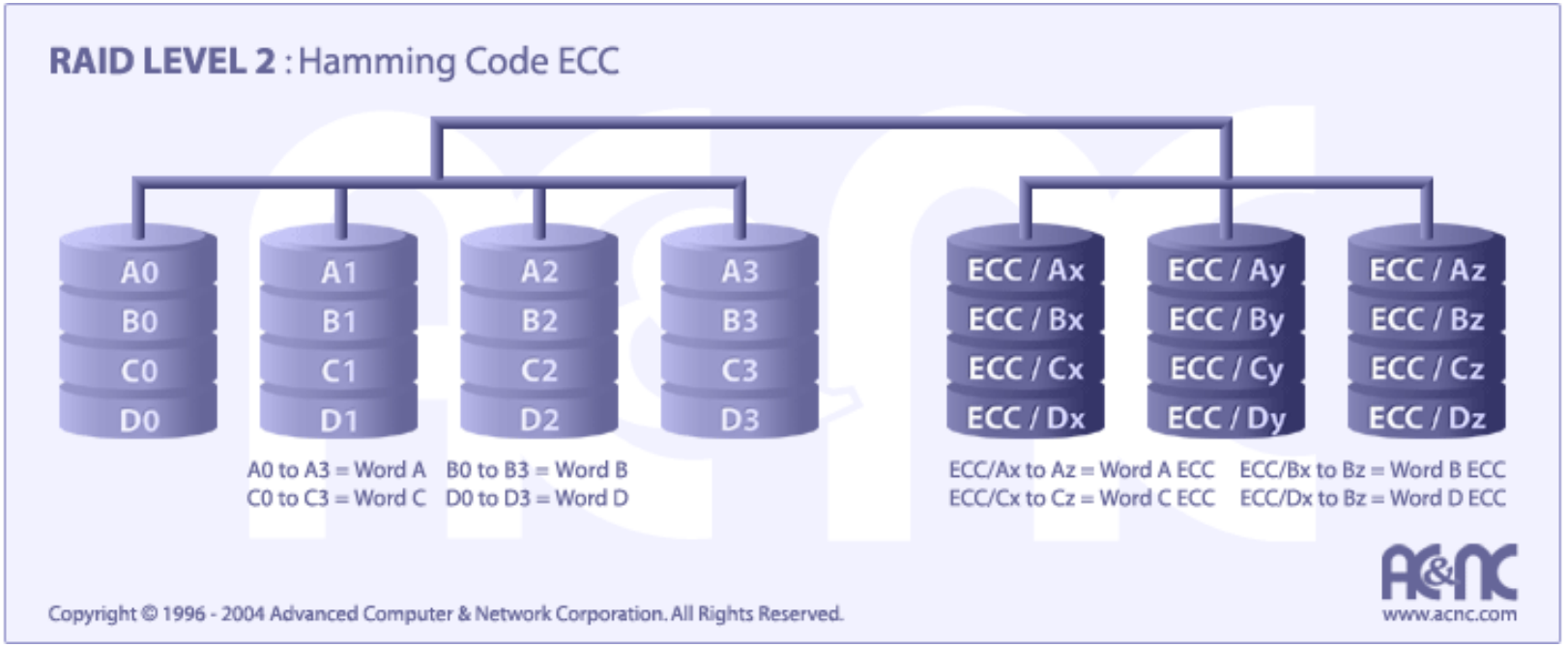
- Exact copy of the drive (100% redundancy)
- Write 1X Read 2X
- Highest disk overhead of all RAID types

RAID 1



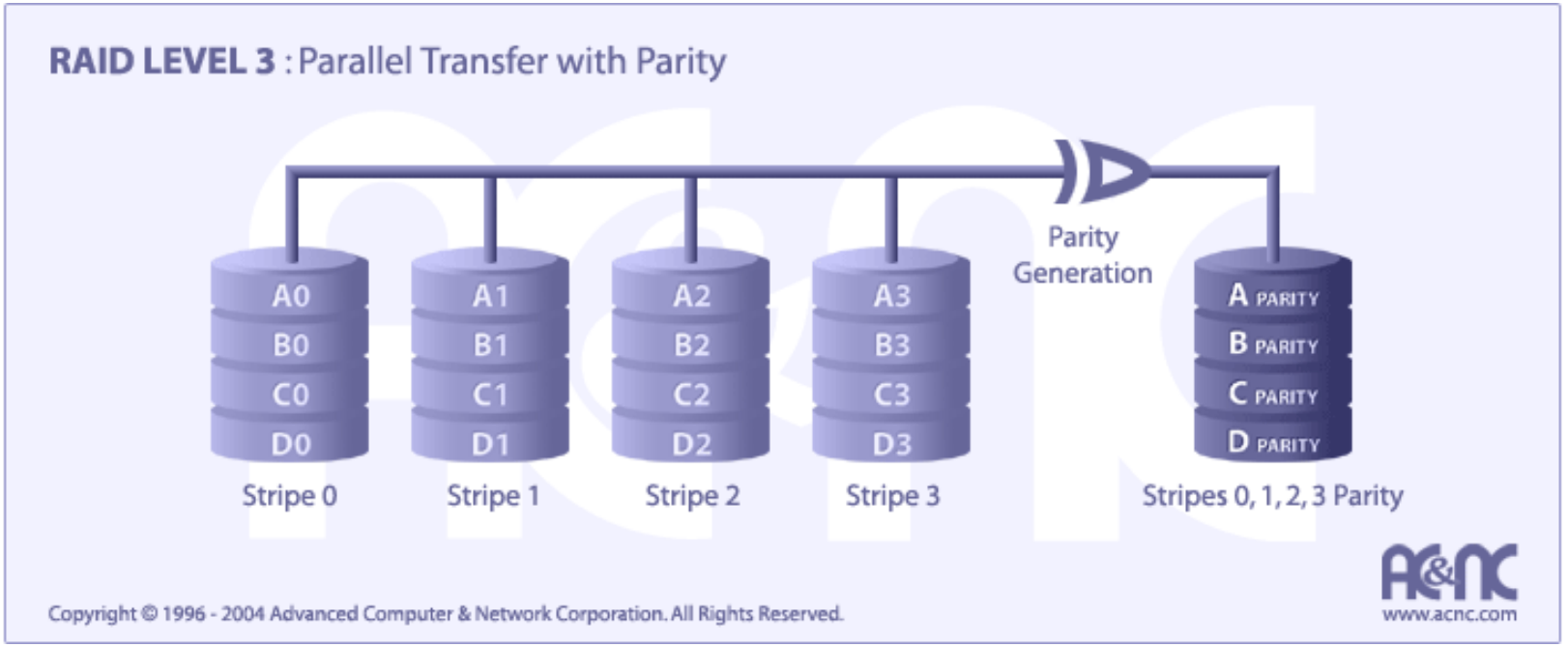
- Exact copy of the drive (100% redundancy)
- Write 1X Read 2X
- Highest disk overhead of all RAID types

RAID 2



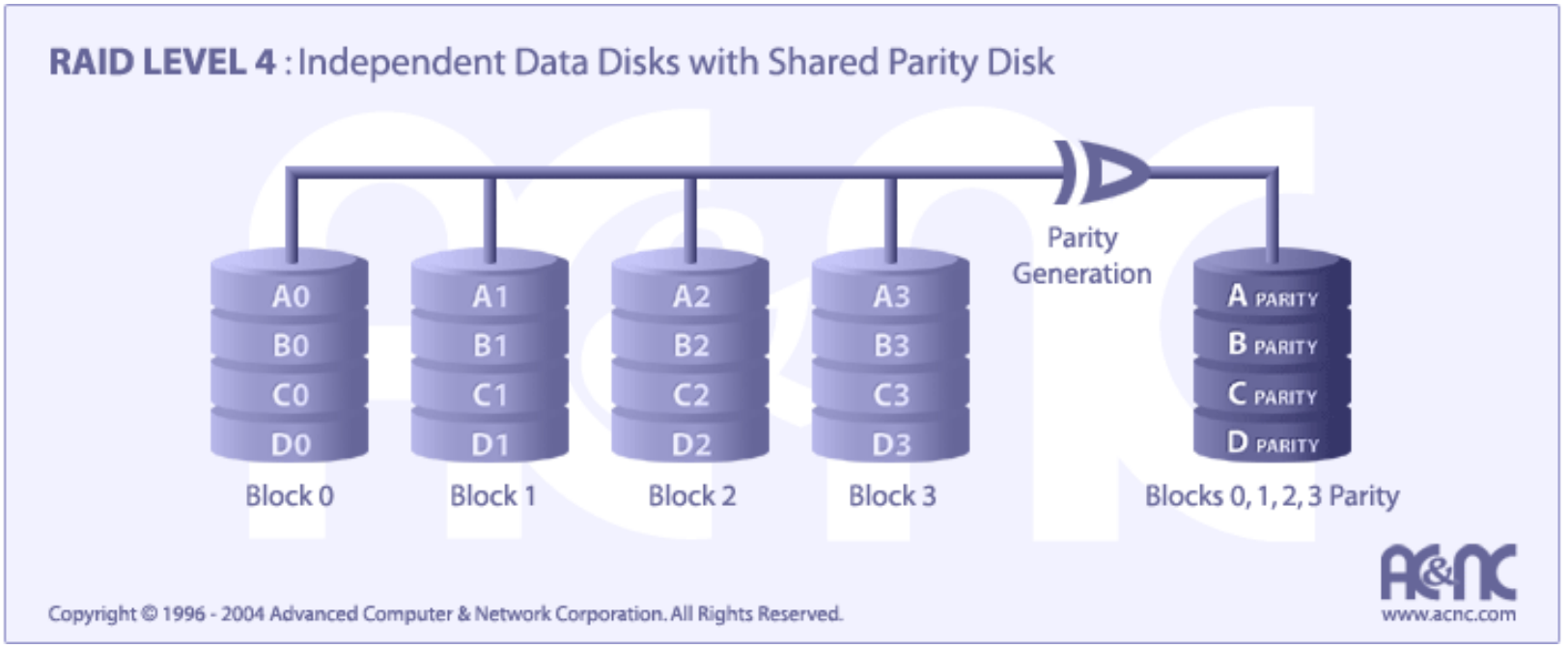
- Error Correction Code written to additional disks
- Very high ratio of ECC disks to data disks – inefficient
- No commercial implementation exists

RAID 3



- The data block is subdivided ("striped")
- Stripe parity is generated on Writes, checked on Reads
- Efficient – low number of extra disks

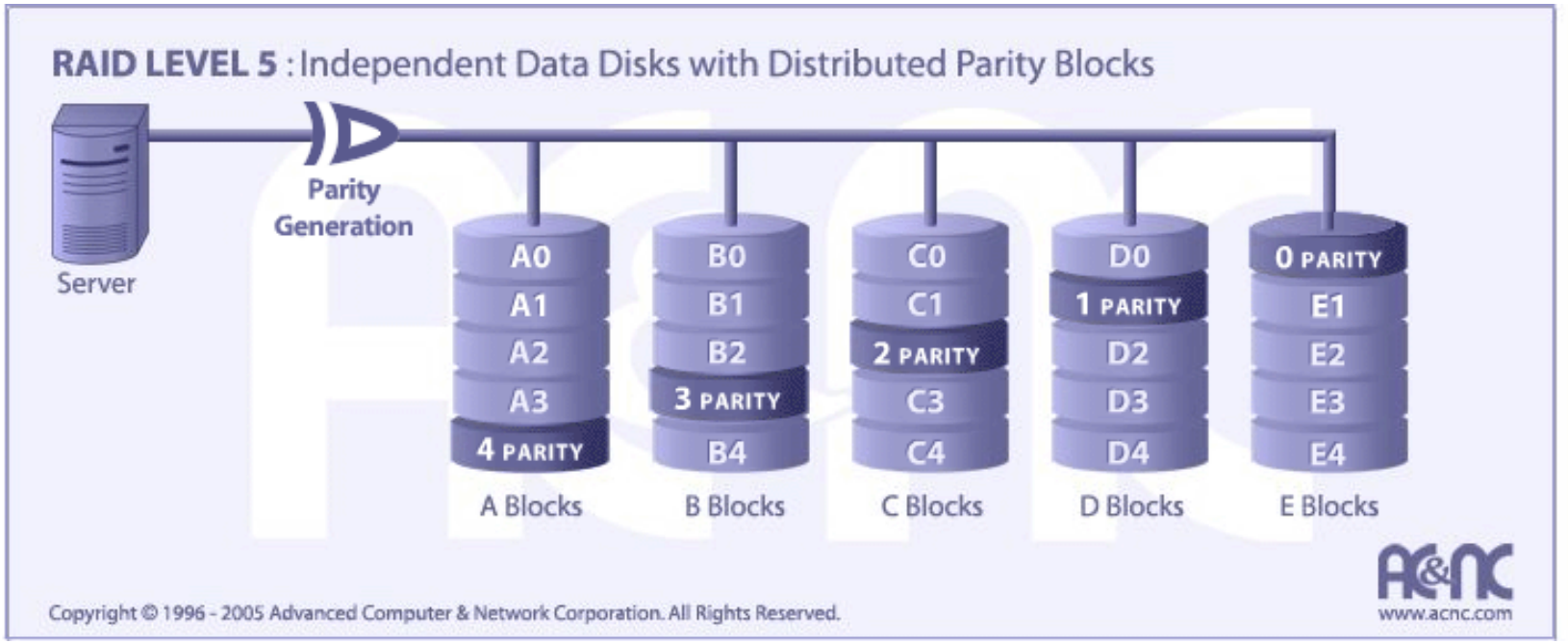
RAID 4



- Same as RAID 3
  - Block size stripes instead of bit interleaving

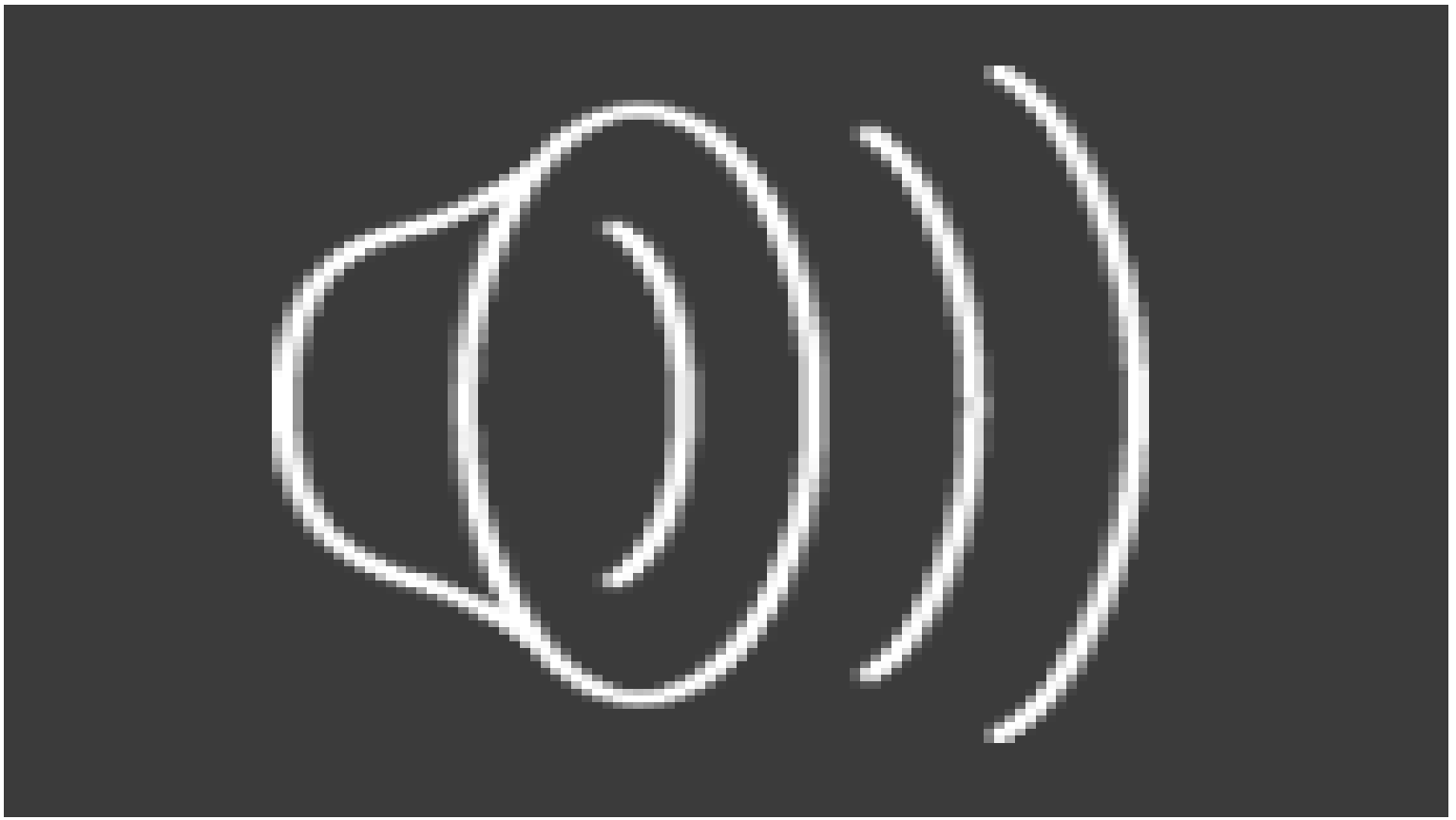
- Common stripe size = sector size (512 bytes)

RAID 5



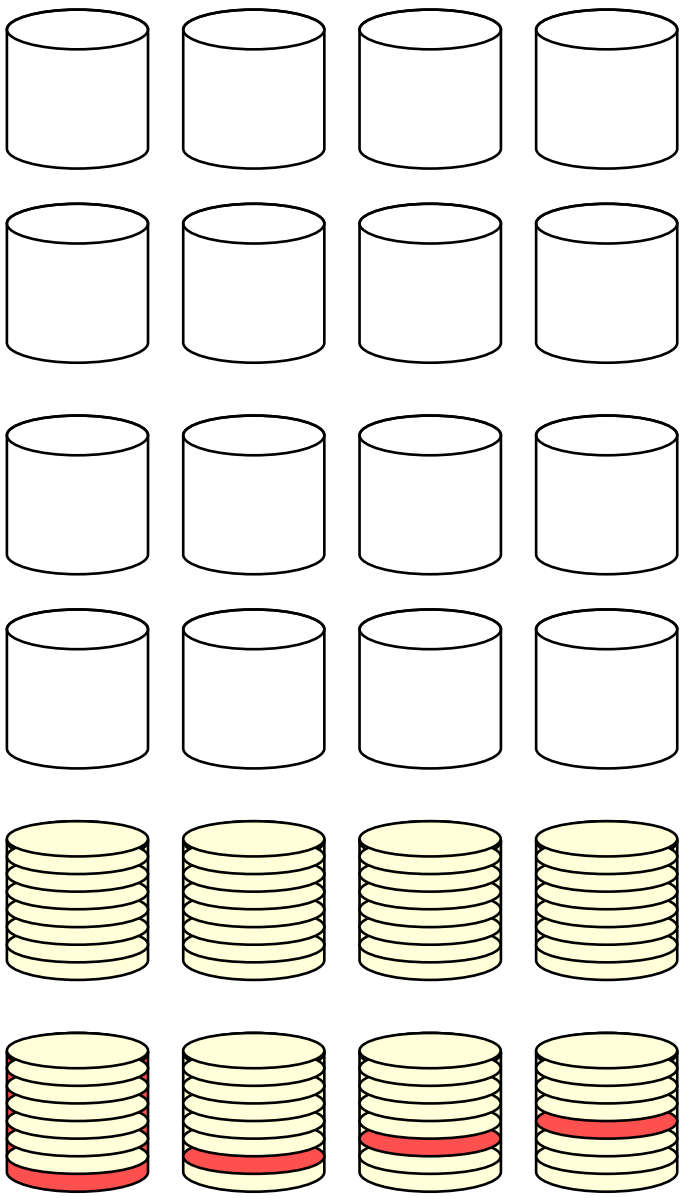
- Parity and data distributed across disks
- No single bottleneck disk for parity access
- Performance again similar to RAID 0

RAID 5

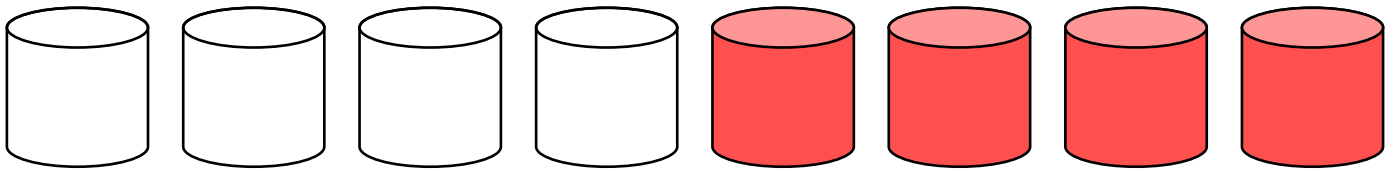


- Parity and data distributed across disks
- No single bottleneck disk for parity access
- Performance again similar to RAID 0

Synopsis of RAID Levels



RAID Level 0: Non redundant



RAID Level 1:
Mirroring

RAID Level 2:
Bit-interleaved, ECC

RAID Level 3:
Bit-interleaved, parity

RAID Level 4:
Block-interleaved, parity

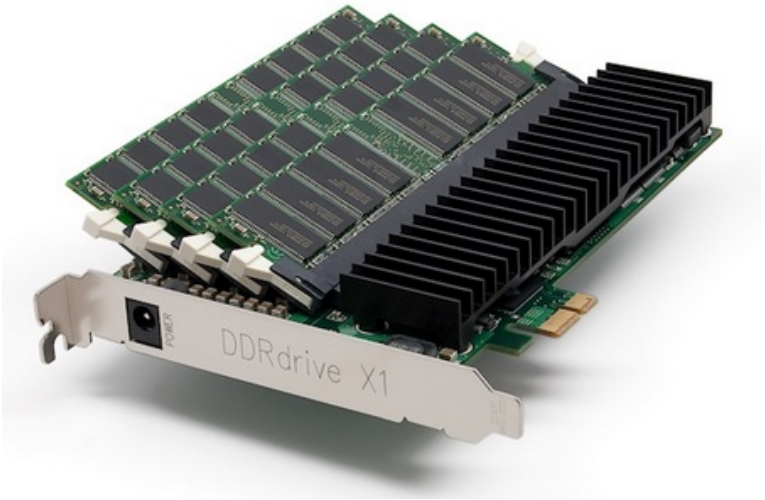
RAID Level 5:
Block-interleaved, distributed parity

Input/Output

- I/O Hardware
- I/O Software
- I/O Software Layers
- Disks
  - SSD
- Clocks
- User Interfaces
- Thin Clients
- Power Management

SSD

- Looks familiar…



RAM-based

(volatile)

Flash-based

(non-volatile)

Flash SSD

- Low latency
- Low power
- Solid-state reliability
- Wide range of applications
  - Embedded devices
  - Laptops/desktops
  - Servers
- Not without issues!

Flash SSD Issues

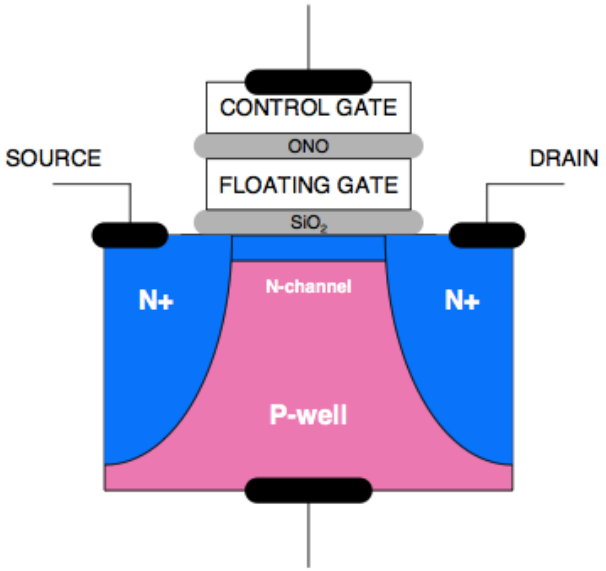
- Erase needed before write
  - Memory cells degrade with each write
  - At most 100K writes possible
  - Need wear leveling
- Unit of read/write differs
  - R/W: page (typically 2-4KB)
  - Erase: block (typically 64 pages)
- Latencies differ
  - Read (25us) < write (250us) < erase (500us)

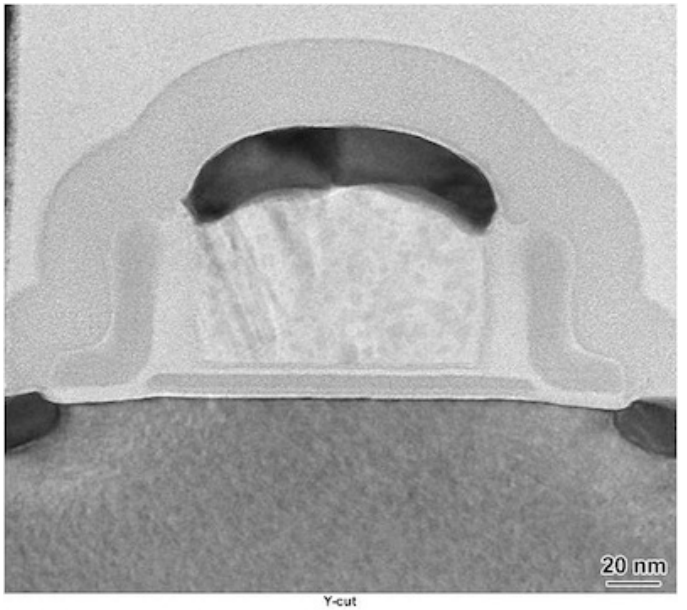
Flash SSD Issues

- Complex controller, supporting:
  - Error correction
  - Wear leveling
  - Bad block mapping
  - Read scrubbing
  - Read/write caching
  - Garbage collection
  - Encryption
- We'll cover a couple of these, but first let's see how SSDs work...

The Mechanism

- Floating gate transistors
  - To see how transistors work, see this video:
    https://www.youtube.com/watch?v=RdYHljZi7ys

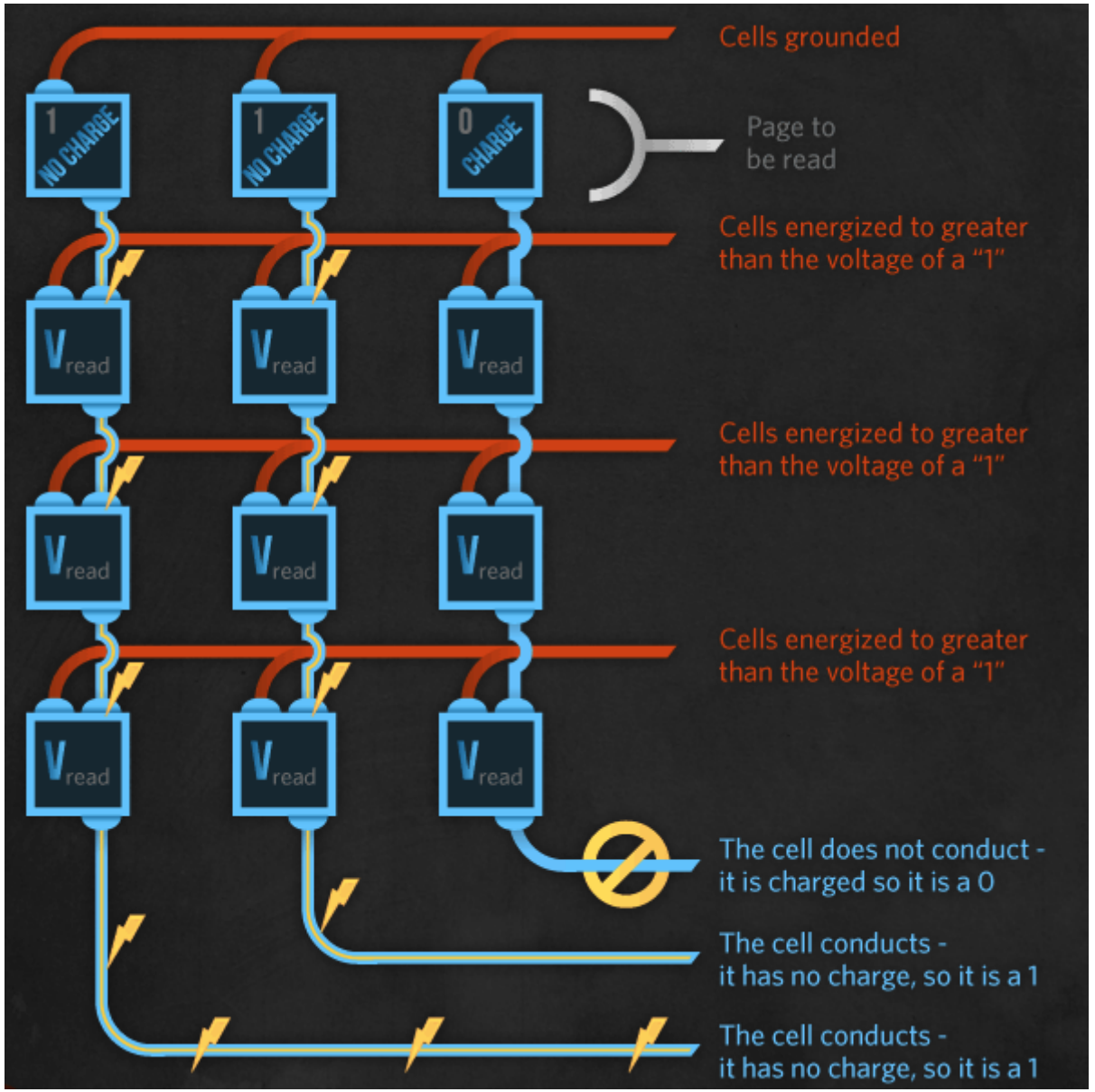




The Mechanism

- Programming
  - Apply high voltage to control gate to set cell to 1
  - Applying lower voltage to control gate allows current to flow from source to drain
- Erasing
  - Applying high voltage of opposite polarity pulls electrons out of the floating gate
- Read
  - Applying voltage at some threshold gives 1 if current flows (cell charged), 0 if not

NAND Flash


Cells grounded
1
NO CHARGE
1
NO CHARGE
0
CHARGE
Page to
be read
Cells energized to greater
than the voltage of a "1"
V read
Cells energized to greater
than the voltage of a "1"
Cells energized to greater
than the voltage of a "1"
The cell does not conduct -
it is charged so it is a 0
The cell conducts -
it has no charge, so it is a 1
The cell conducts -
it has no charge, so it is a 1

Wear Leveling

- Typically 3-5k erase cycles are possible
- No wear leveling
  - Every write causes: read, erase, modify, write to same block
- Dynamic wear leveling
  - Logical block addresses (from OS) mapped to new physical block (original block marked invalid)
- Static wear leveling
  - Same as dynamic, but data in blocks that do not change are periodically moved

Read Disturb

- Thousands of reads of blocks can cause nearby cells to change
- Keep track of reads
  - Rewrite nearby data periodically

Input/Output

- I/O Hardware
- I/O Software
- I/O Software Layers
- Disks
  - SSD
- Clocks
- User Interfaces
- Thin Clients
- Power Management

Clocks (Timers)

- Crystal oscillator
- Counter
- Holding register
- Operation (square-wave mode)
  - * Holding register copied to counter
  - Counter decremented at each pulse
  - If counter at 0, issue interrupt and go to *

(clock ticks)

Programmable Clock

- Assume 500 MHz crystal
  - Counter is pulsed every 2 nsec
- Assume unsigned 32-bit register
  - Interrupts can occur at intervals
    - 2 nsec to ~8.6 sec
- Battery powered backup clock keeps current time between powered down periods

Uses

- Prevent processes from running too long
  - Initialize counter to the value of a process quantum
- Account for CPU usage
  - Start timer when process starts
  - Check timer when process is stopped
- Handle alarm system calls
  - Processes may require timed warnings
    - e.g. retransmission of packets

Uses

- Provide watchdog timers
  - System timer that triggers a reset or corrective action
  - Regular heartbeat signal expected to reset watchdog, else it triggers a processor reset or non-maskable interrupt
    - Aka: service pulse, kicking the dog, feeding the watchdog
- Profiling, monitoring, and statistic gathering
  - Use system clock to keep track of execution time of various program components (see gprof and bprof)

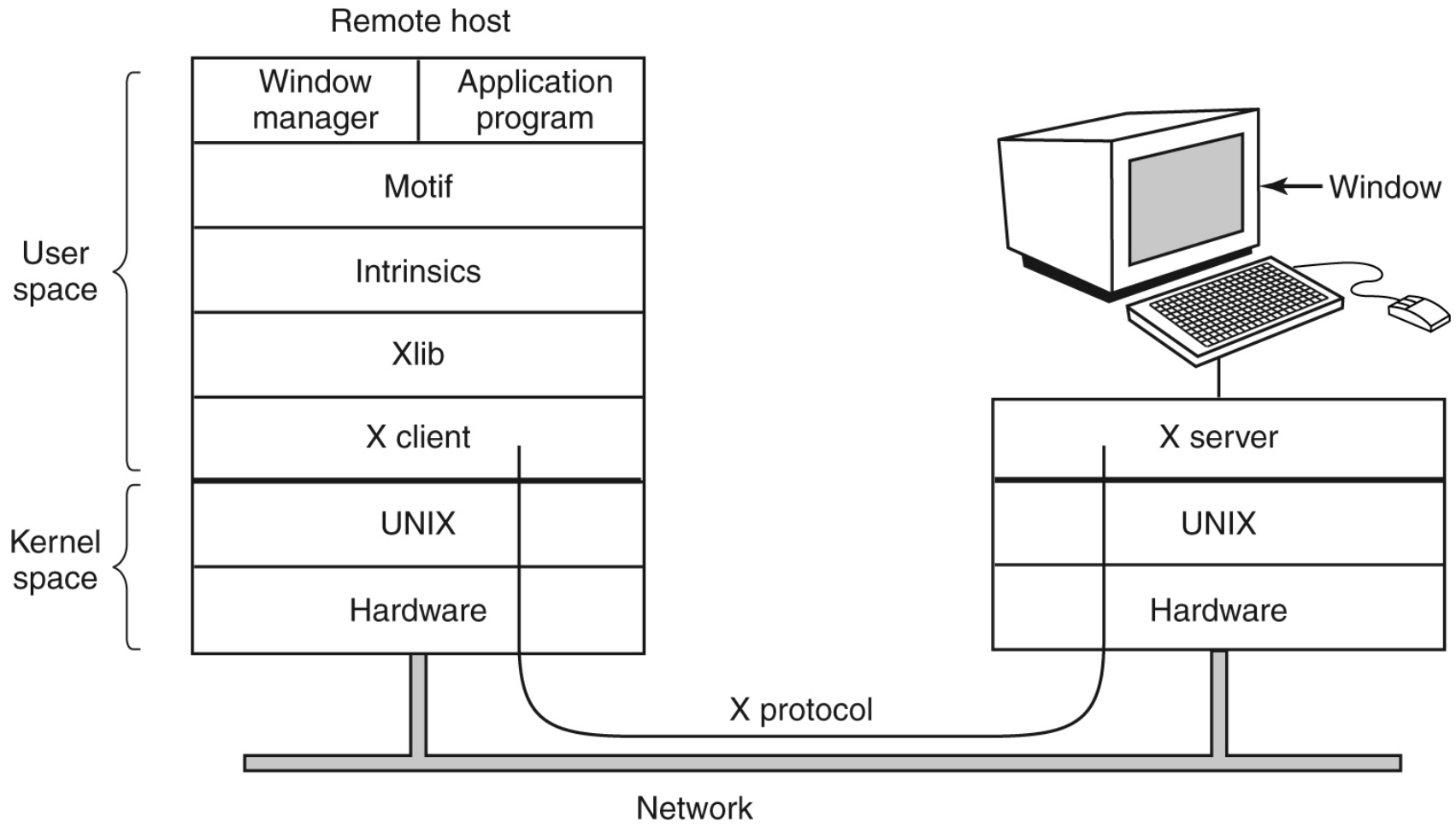
Input/Output

- I/O Hardware
- I/O Software
- I/O Software Layers
- Disks
  - SSD
- Clocks
- User Interfaces
- Thin Clients
- Power Management

GUIs

- 1973 – Xerox develops bitmapped screen
- 1973 – Xerox develops first WYSIWYG editor
  - Also develops the Alto and Star
- 1979 – Apple visits Xerox
  - Releases Macintosh in 1980
- 1983 – Microsoft releases Windows 1.0
  - Using licensed elements from both Apple and Xerox
- 1984 – MIT releases X Window Server for Unix
  - 1987 sees protocol version (more in a bit)

X Window System



Input/Output

- I/O Hardware
- I/O Software
- I/O Software Layers
- Disks
  - SSD
- Clocks
- User Interfaces
- Thin Clients
- Power Management

Thin Clients

- Computer/Program that depends on a server to fulfill its computational needs
  - Server is a single point of failure for all clients
  - Clients are cheap
  - Clients are simple
  - Clients' graphics are slow
- Aaaand done.

  With thin clients that is…

Keyboard

- Interrupt for each key press and release
- Driver just translates port data
- Number in I/O port ^ key number/scan code
- Scan code can be translated to ASCII
- Raw mode (noncanonical mode)
  - All characters typed are given by driver
    - Character oriented
- Cooked mode (canonical mode)
  - Following intraline editing, driver returns line
    - Line oriented

Canonical/Noncanonical Modes

- Characters stored until entire line is accumulated
  - $DSTE]\ \ ]\ \ ]\ \ ATE$ (typed)
  - $DATE$ (returned by driver)
- To allow user to see characters typed: echoing
- In noncanonical mode, programs must buffer input
- Canonical/noncanonical mode set via library

Canonical Mode: Special Characters

Character	POSIX name	Comment
CTRL-H	ERASE	Backspace one character
CTRL-U	KILL	Erase entire line being typed
CTRL-V	LNEXT	Interpret next character literally
CTRL-S	STOP	Stop output
CTRL-Q	START	Start output
DEL	INTR	Interrupt process (SIGINT)
CTRL-\	QUIT	Force core dump (SIGQUIT)
CTRL-D	EOF	End of file
CTRL-M	CR	Carriage return (unchangeable)
CTRL-J	NL	Linefeed (unchangeable)

Pointers

- Ball mouse
  - Rubber ball rotates x-axis/y-axis rollers
- Optical mouse
  - Capture images of underlying surface
  - Find differences to detect movement
- Touchpad
  - Sense capacitance or conductance
- Send interrupt based on distance traveled
  - Or key presses