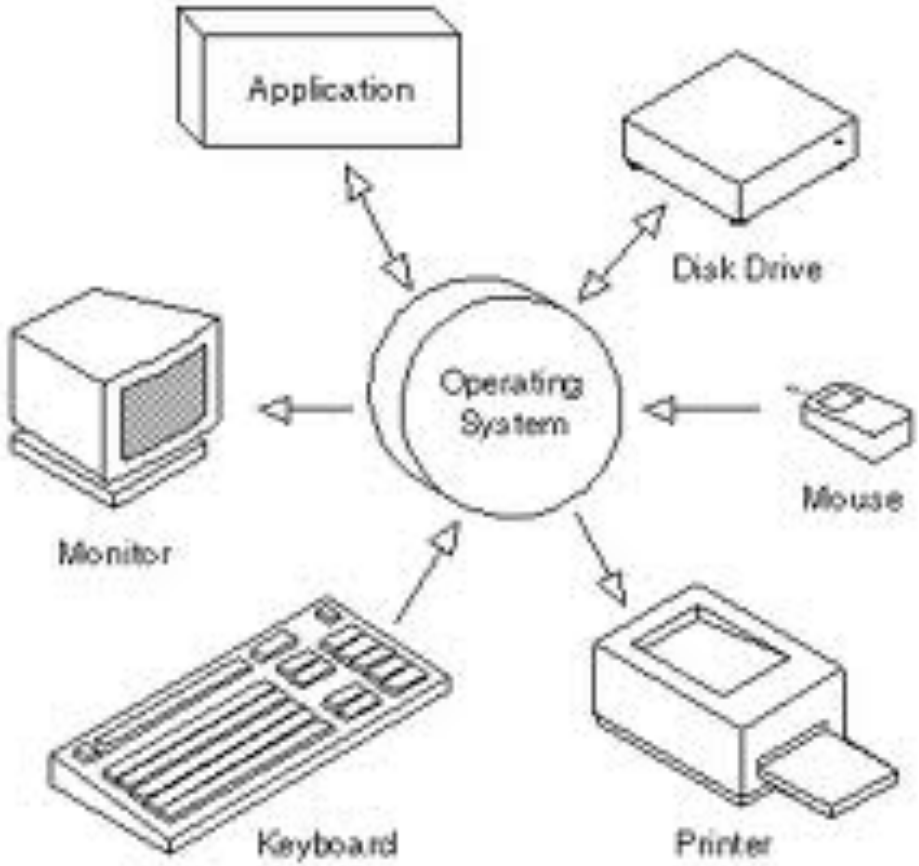
CSCI-GA.2250-001

Operating Systems

I/O

Hubertus Franke
frankeh@cs.nyu.edu


Application
Disk Drive
Operating System
Monitor
Mouse
Keyboard
Printer

Categories of I/O Devices

External devices that engage in I/O with computer systems can be grouped into three categories:

Human readable

- suitable for communicating with the computer user
- printers, terminals, video display, keyboard, mouse

Machine readable

- suitable for communicating with electronic equipment
- disk drives, USB keys, sensors, controllers

Communication

- suitable for communicating with remote devices
- modems, digital line drivers

A Simple Definition

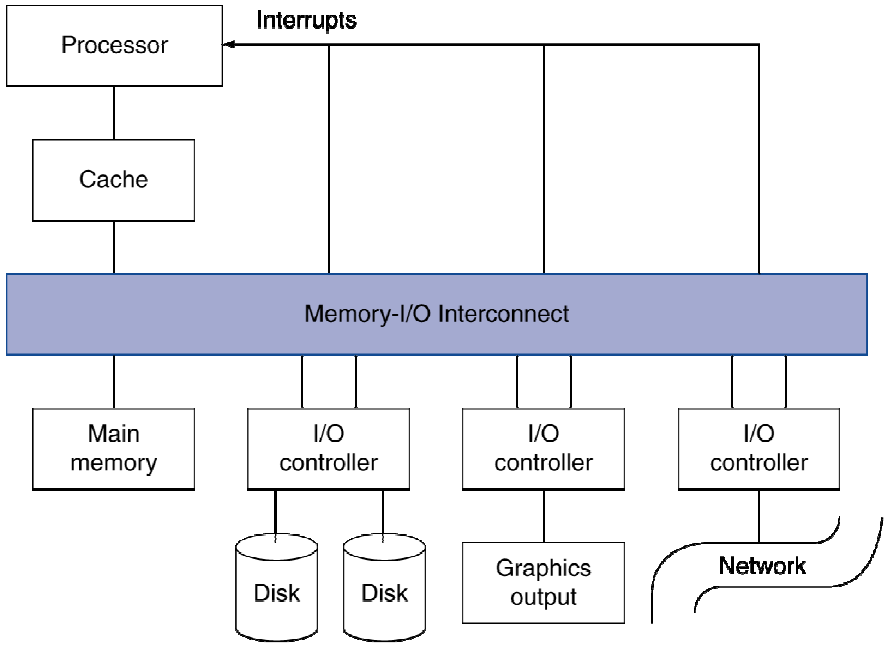
- The main concept of I/O is to move data from/to I/O devices to the processor using some modules and buffers.
- This is the way the processor deals with the outside world
- Examples: mouse, display, keyboard, disks, scanners, speakers, etc.

The OS and I/O

- The OS controls all I/O devices
  - Issue commands to devices
  - Catch interrupts
  - Handle errors
- Provides an interface between the devices and the rest of the system

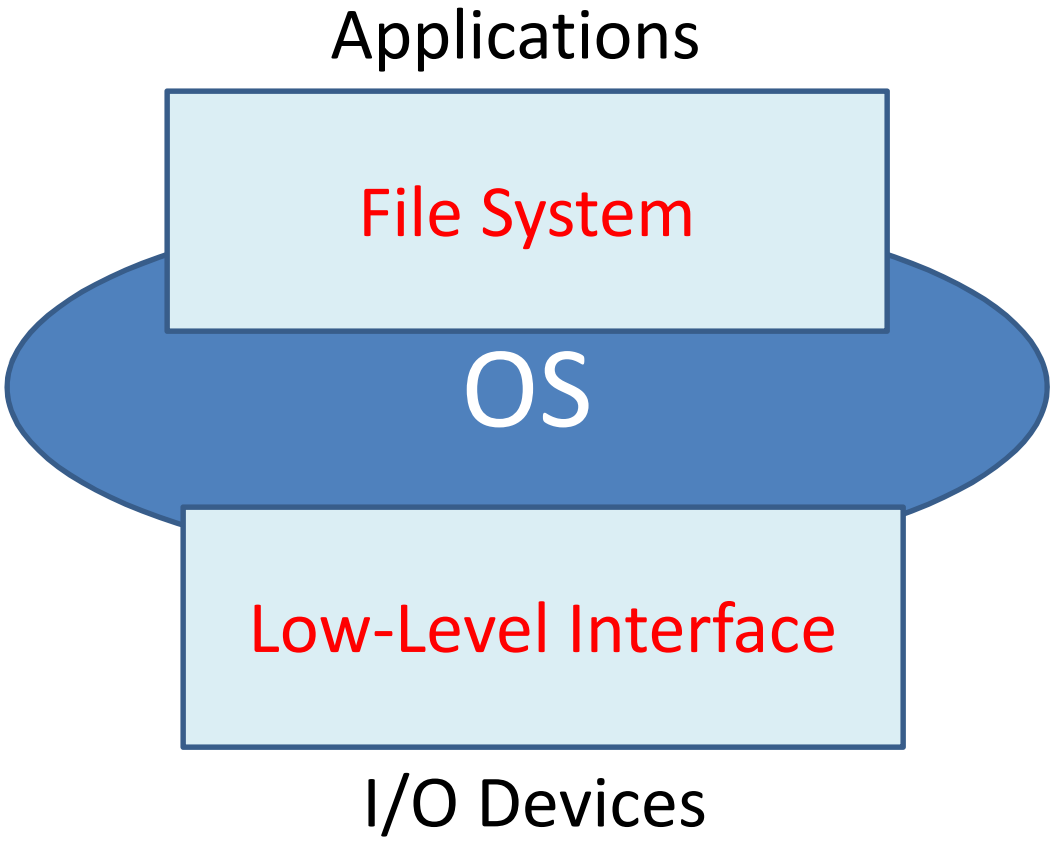
I/O Devices: Challenges

- Very diverse devices
  - behavior (i.e., input vs. output vs. storage)
  - partner (who is at the other end?)
  - data rate
- I/O Design affected by many factors (expandability, resilience)
- Performance:
  - access latency
  - throughput
  - connection between devices and the system
  - the memory hierarchy
  - the operating system
- A variety of different users

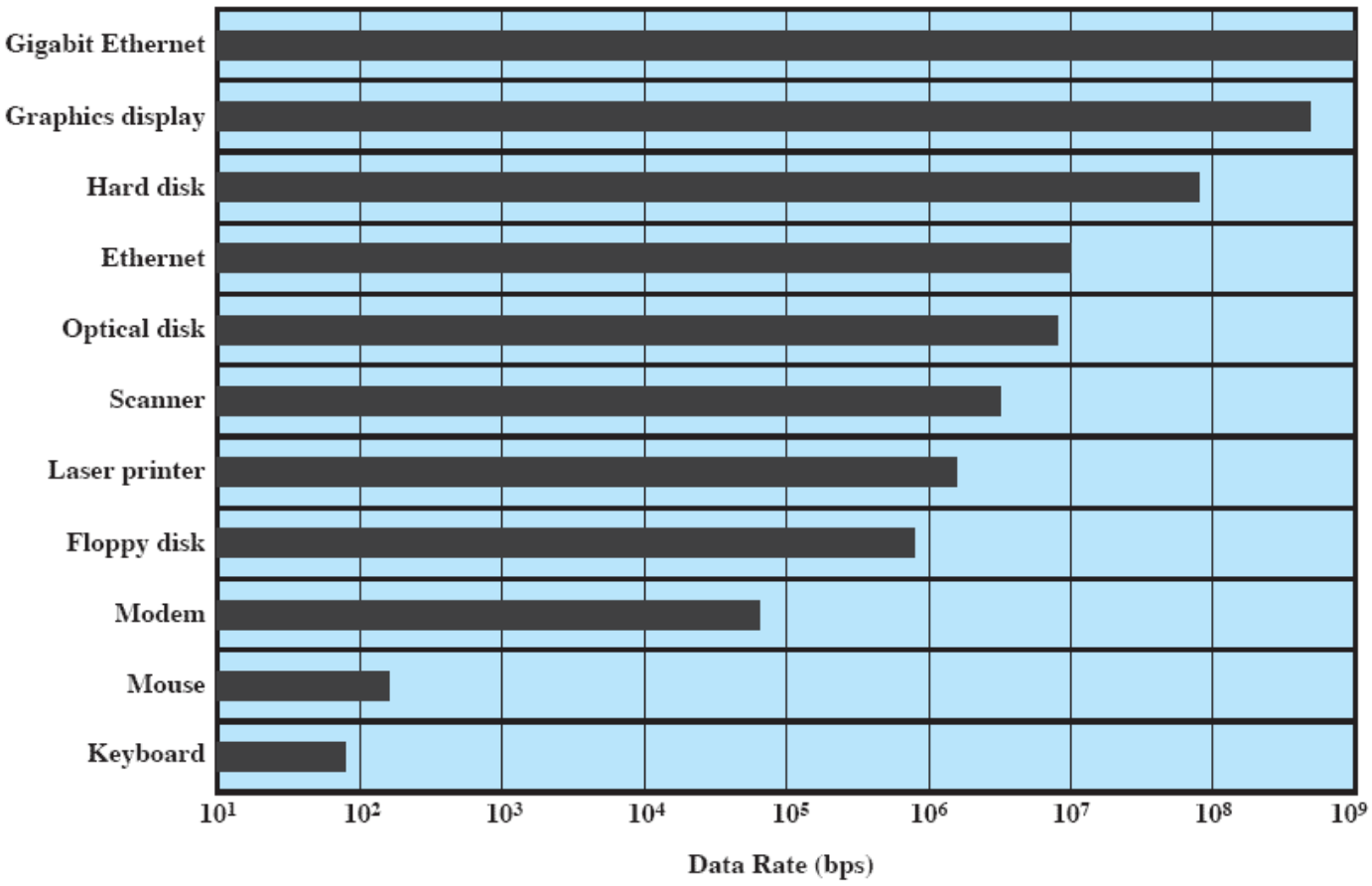


I/O Devices

- Block device
  - Stores information in fixed-size blocks
  - Each block has its own address
  - Transfers in one or more blocks
  - Example: Hard-disks, CD-ROMs, USB sticks
- Character device
  - Delivers or accepts stream of character
  - Is not addressable
  - Example: mice, printers, network interfaces

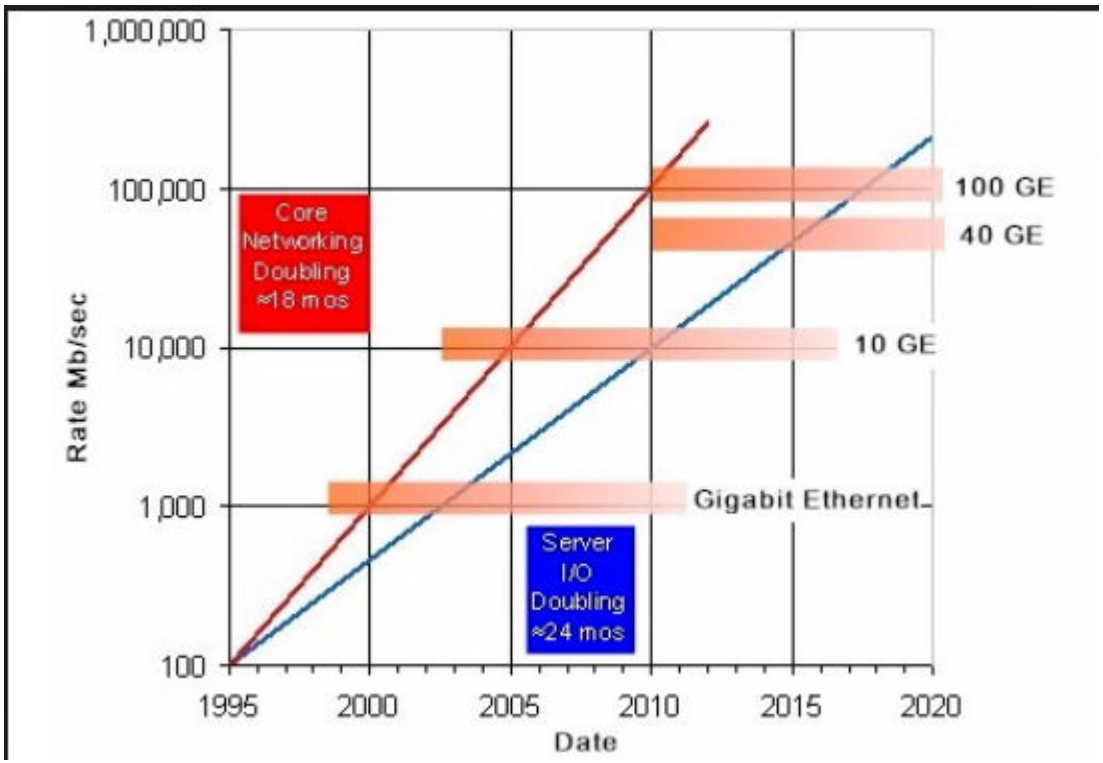

Applications
File System
OS
Low-Level Interface
I/O Devices

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner	400 KB/sec
Digital camcorder	3.5 MB/sec
802.11g Wireless	6.75 MB/sec
52x CD-ROM	7.8 MB/sec
Fast Ethernet	12.5 MB/sec
Compact flash card	40 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
SATA disk drive	300 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec


Gigabit Ethernet
Graphics display
Hard disk
Ethernet
Optical disk
Scanner
Laser printer
Floppy disk
Modem
Mouse
Keyboard
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9
Data Rate (bps)

Devices change !!!

- Network speed doubling every 18-24 month
- Approaching 100Gbps / 40Gbps



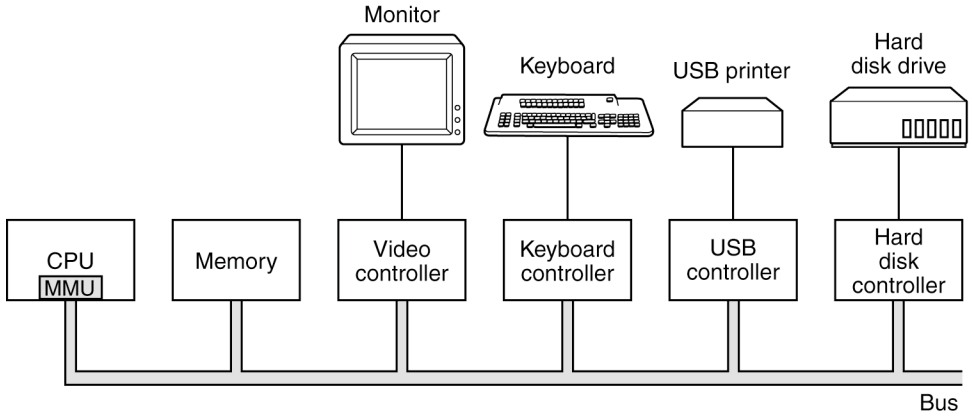
I/O Units

Mechanical component

Electronic Component

The Device Itself

Device Controller



Controller and Device

- Each controller has few registers used to communicate with CPU
- By writing/reading into/from those registers, the OS can control the devices.
- There are also data buffers in the device that can be read/written by the OS

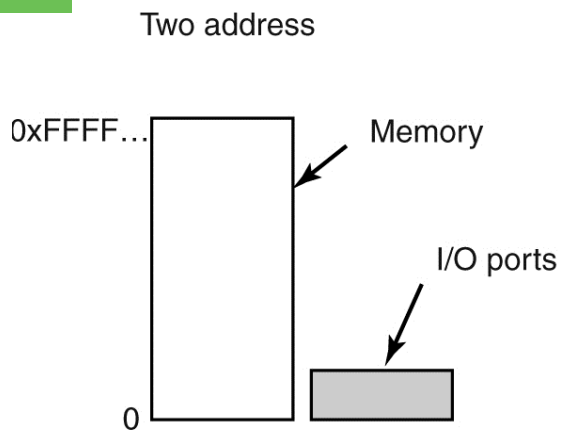
How does CPU communicate with control registers and data buffers?

Two main approaches

- I/O port space
- Memory-mapped I/O

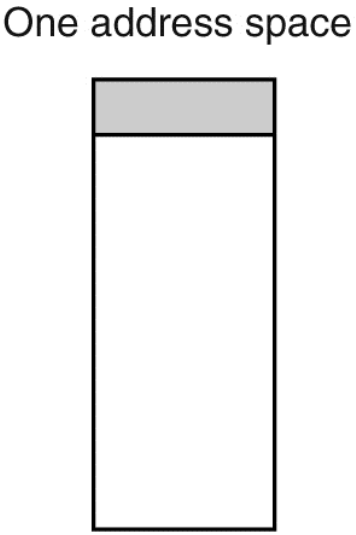
I/O Port Space

- Each control register is assigned an I/O port number
- The set of all I/O ports form the I/O port space
- I/O port space is protected



Memory-Mapped I/O

- Map control registers into the memory space
- Each control register is assigned a unique memory address

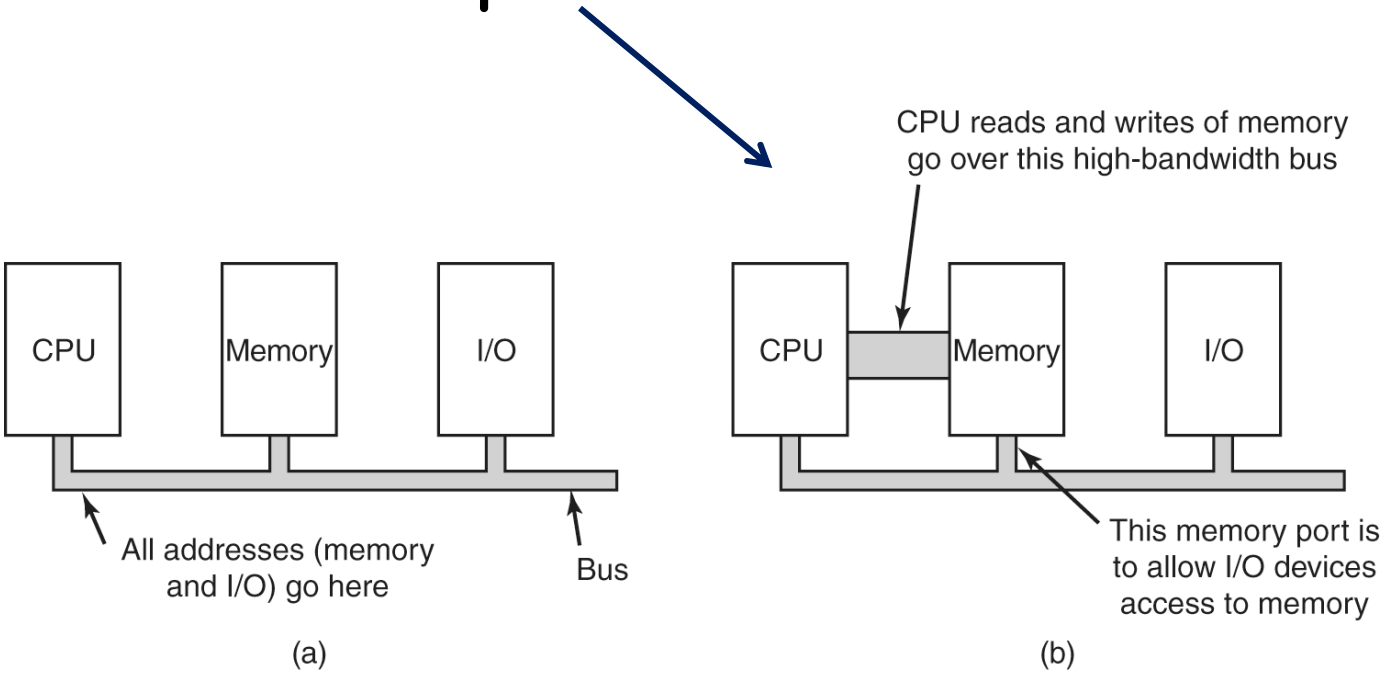


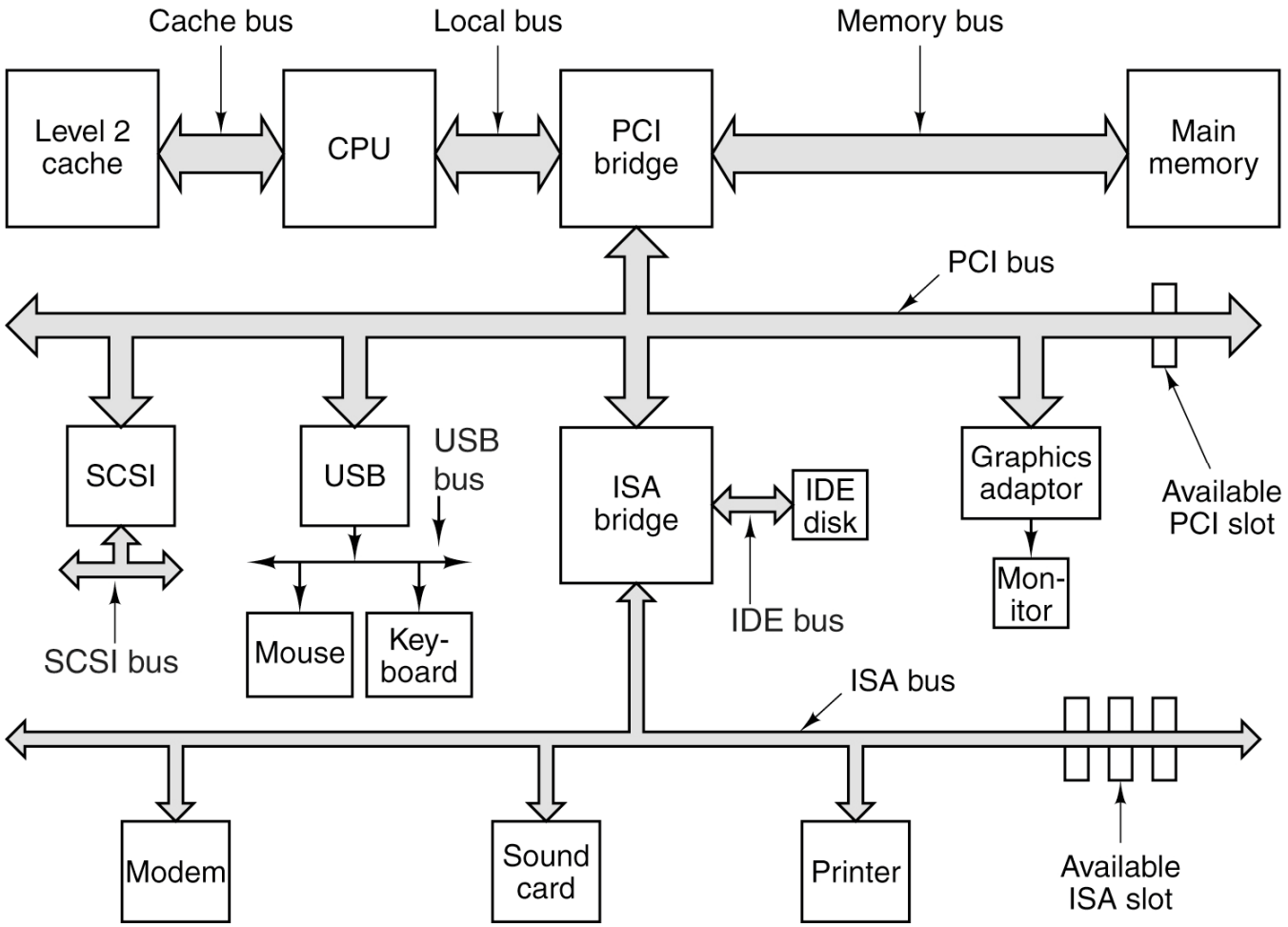
Advantages of Memory-Mapped I/O

- Device drivers can be written entirely in C (since no special instructions are needed)
- No special protection is needed from OS, just refrain from putting that portion of the address space in any user's virtual address space
- Every instruction that can reference memory can also reference control registers

Disadvantages of Memory-Mapped I/O

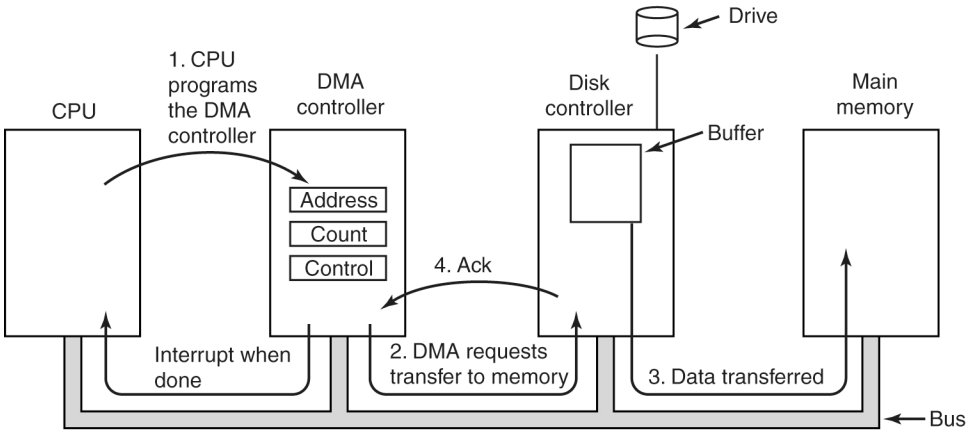
- Caching a device control register can be disastrous
- Hardware complications




Cache bus
Local bus
Memory bus
Level 2 cache
CPU
PCI bridge
Main memory
PCI bus
SCSI
USB
USB bus
ISA bridge
IDE disk
Graphics adaptor
Available PCI slot
Mon-itor
SCSI bus
Mouse
Key-board
IDE bus
ISA bus
Modem
Sound card
Printer
Available ISA slot

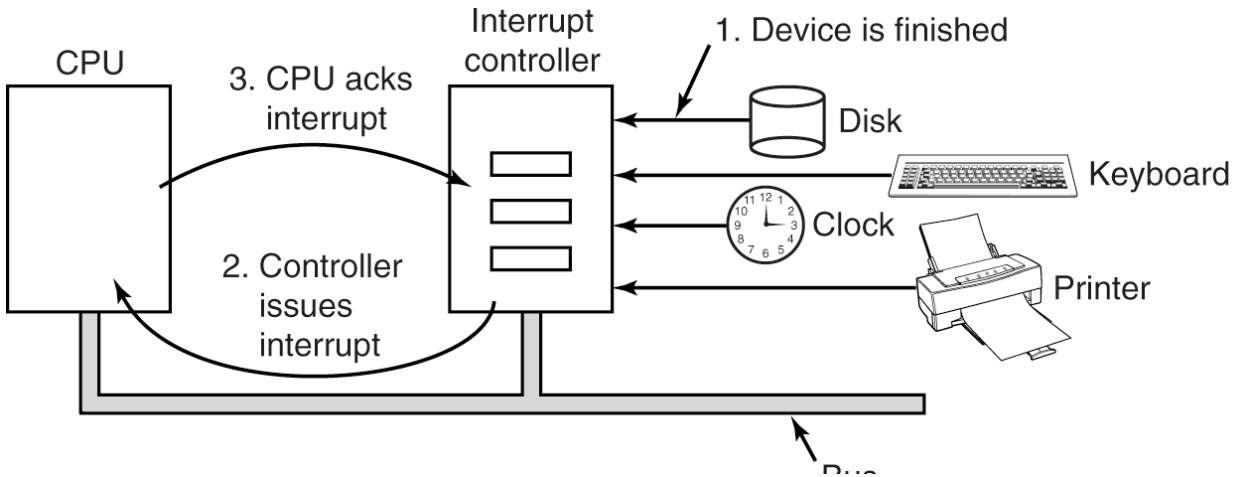
Direct Memory Access (DMA)

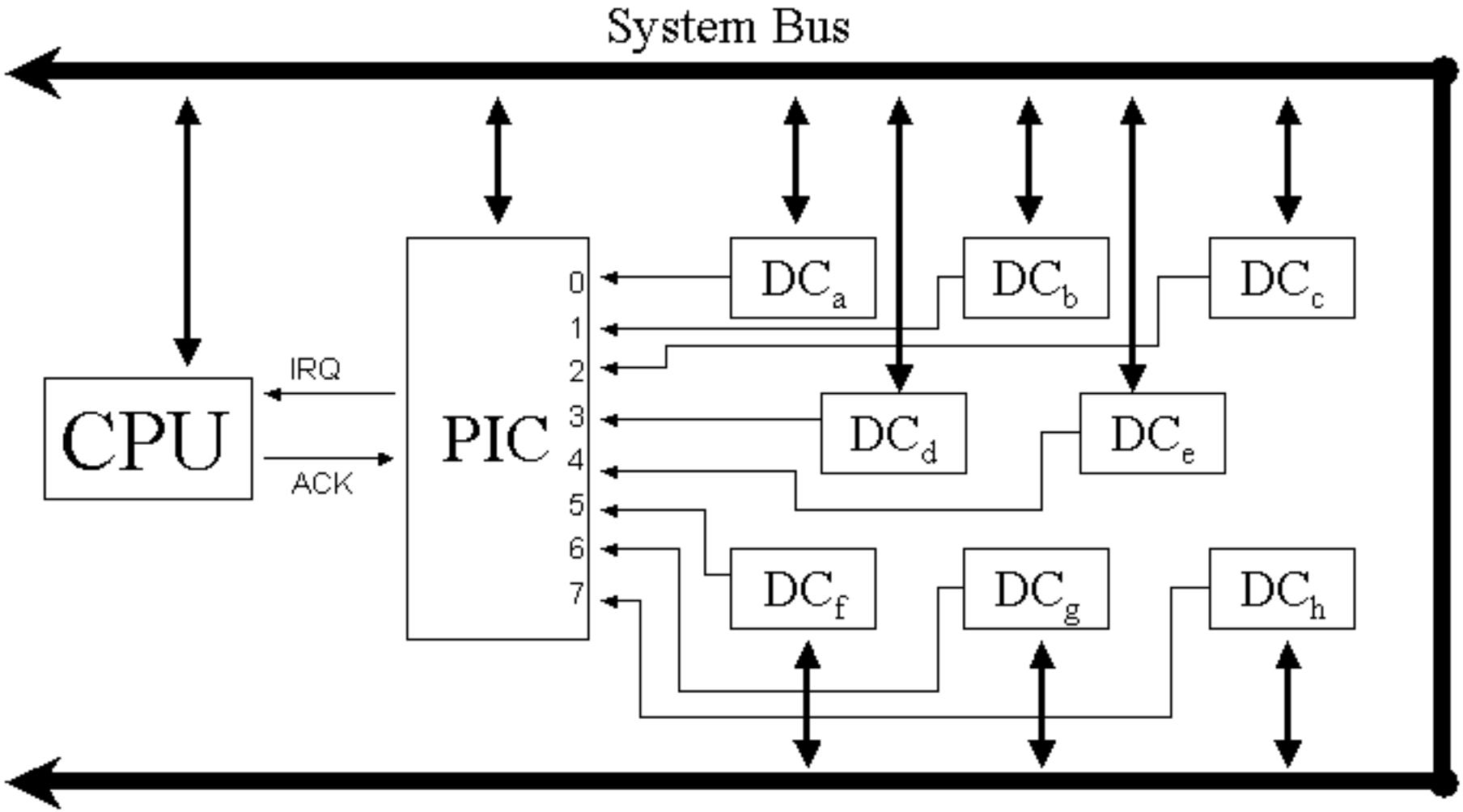
- It is not efficient for the CPU to request data from I/O one byte at a time
- DMA controller has access to the system bus independent of the CPU



Interrupts

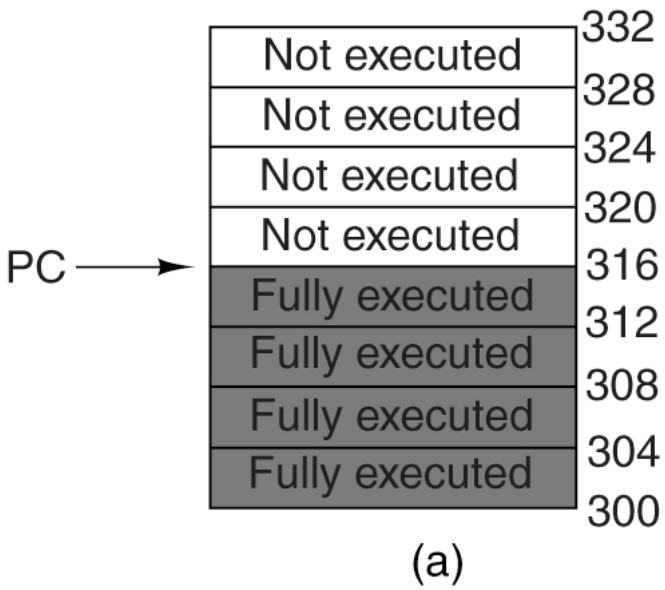
- When an I/O device has finished the work given to it, it causes an interrupt.



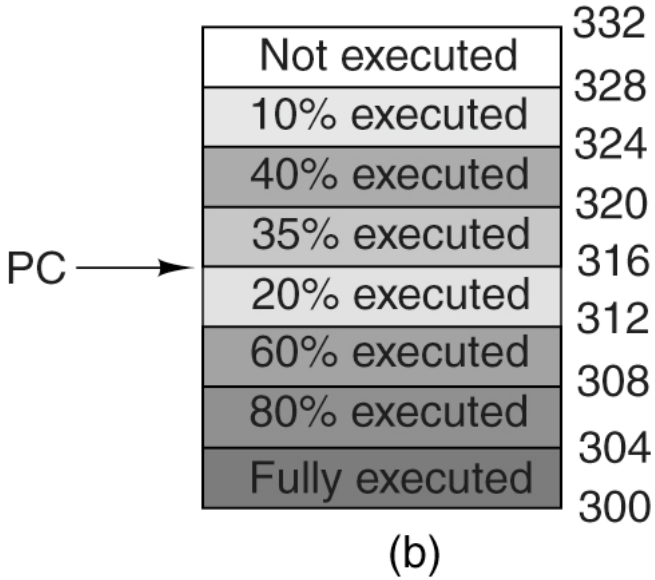

System Bus
CPU
IRQ
ACK
PIC
0
1
2
3
4
5
6
7
DC$_a$
DC$_b$
DC$_c$
DC$_d$
DC$_e$
DC$_f$
DC$_g$
DC$_h$

Precise Interrupts

- Makes handling interrupts much simpler
- Has 4 properties
  - The program counter (PC) is saved in known place
  - All instructions before the one pointed by PC have fully executed
  - No instruction beyond the one pointed by PC has been executed
  - The execution state of the instruction pointed to by the PC is known



Precise Interrupt



Imprecise Interrupt

I/O Software

- Device independence
- Uniform naming
- Error handling
  - Should be handled as close to the hardware as possible
- Synchronous vs asynchronous (interrupt-driven)
- Buffering
- Sharable verses dedicated devices

Three Ways of Doing I/O

- Programmed I/O
- Interrupt-driven I/O
- I/O Using DMA

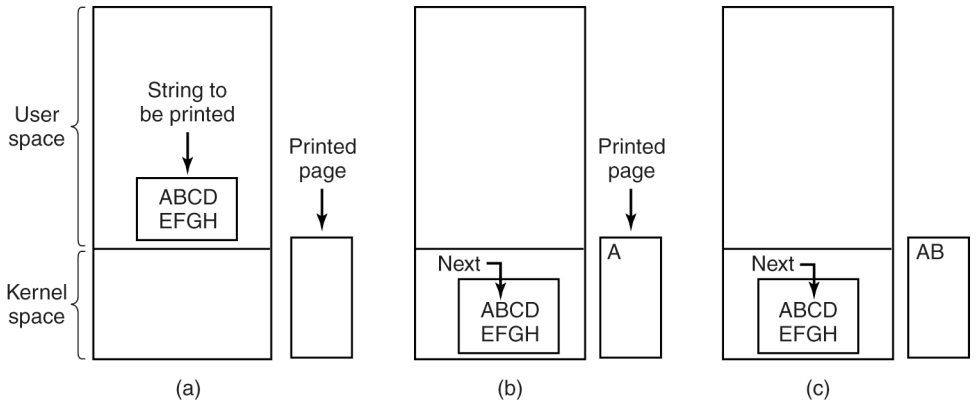
	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

Programmed I/O

- CPU does all the work
- Busy-waiting (polling)

Example:

```
copy_from_user(buffer, p, count);                    /* p is the kernel buffer */
for (i = 0; i < count; i++) {                        /* loop on every character */
        while (*printer_status_reg != READY) ;       /* loop until ready */
        *printer_data_register = p[i];               /* output one character */
}
return_to_user( );
```

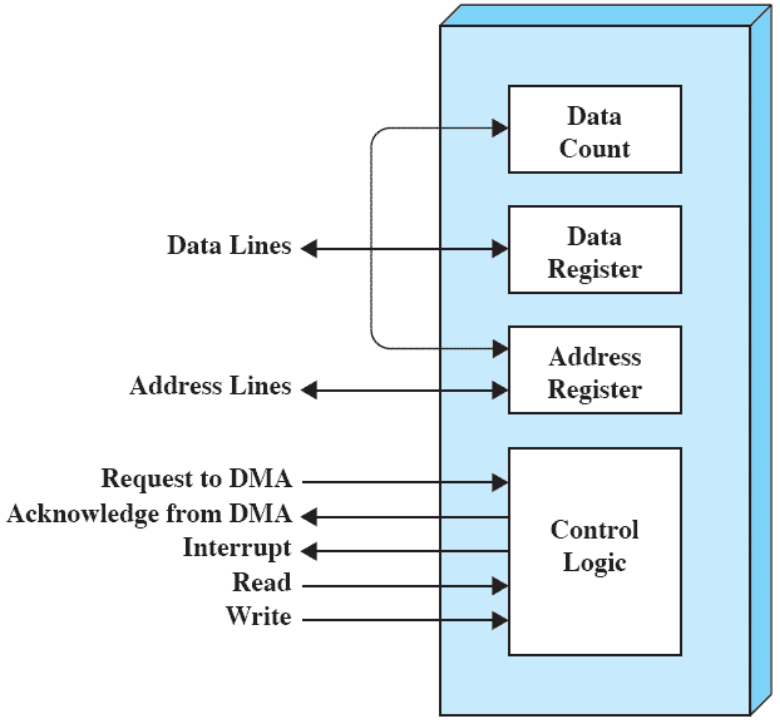


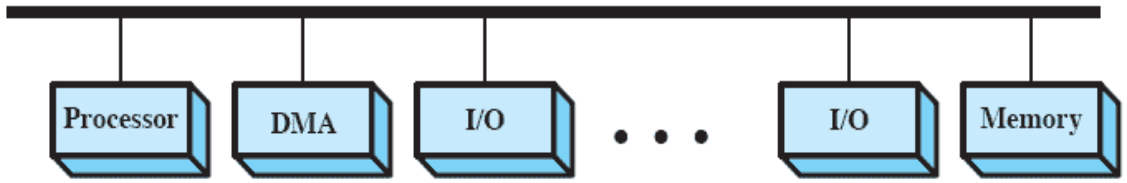
Interrupt-Driven I/O

- Waiting for a device to be ready, the process is blocked and another process is scheduled.
- When the device is ready it raises an interrupt.

I/O Using DMA

- DMA does the work instead of the CPU
- Let the DMA do its work and then interrupts

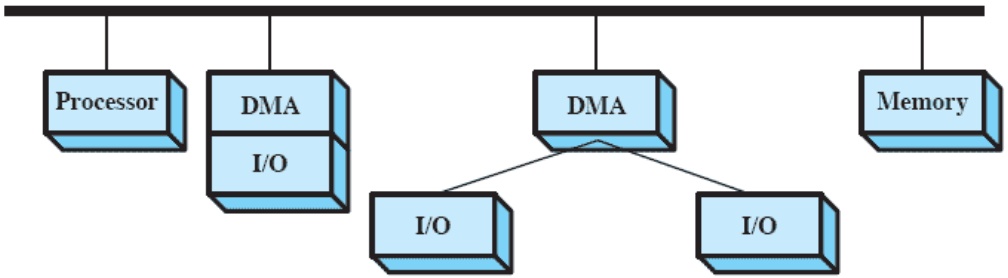




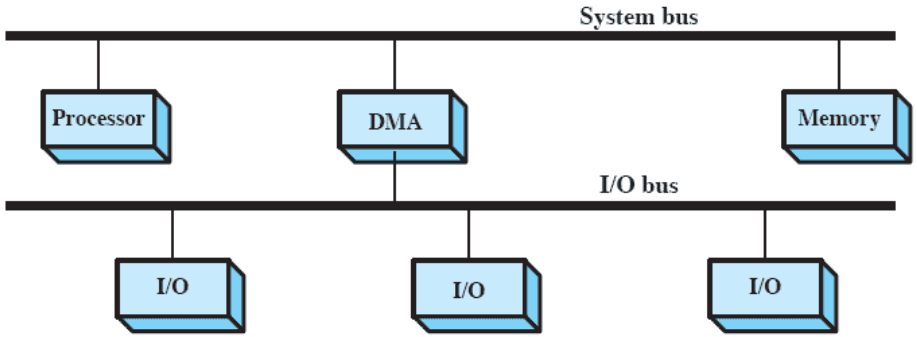
(a) Single-bus, detached DMA

DMA

Alternative



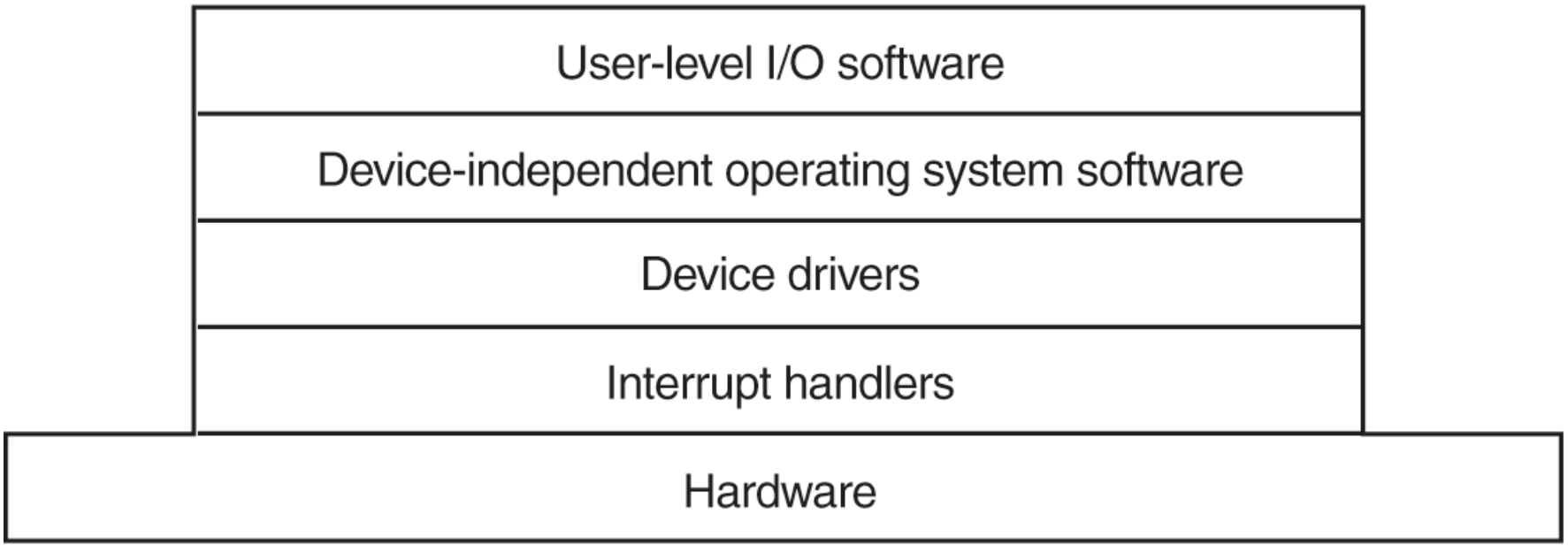
(b) Single-bus, Integrated DMA-I/O



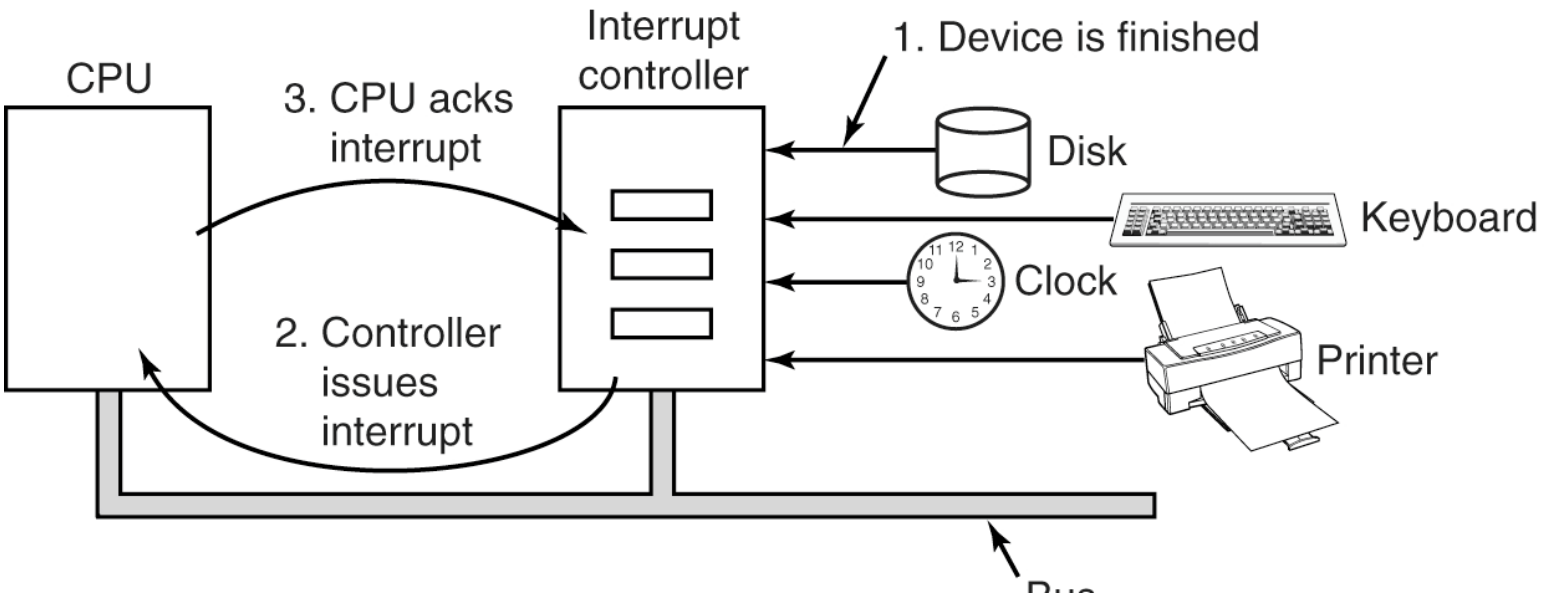
(c) I/O bus

Configurations

OS Software Layers for I/O



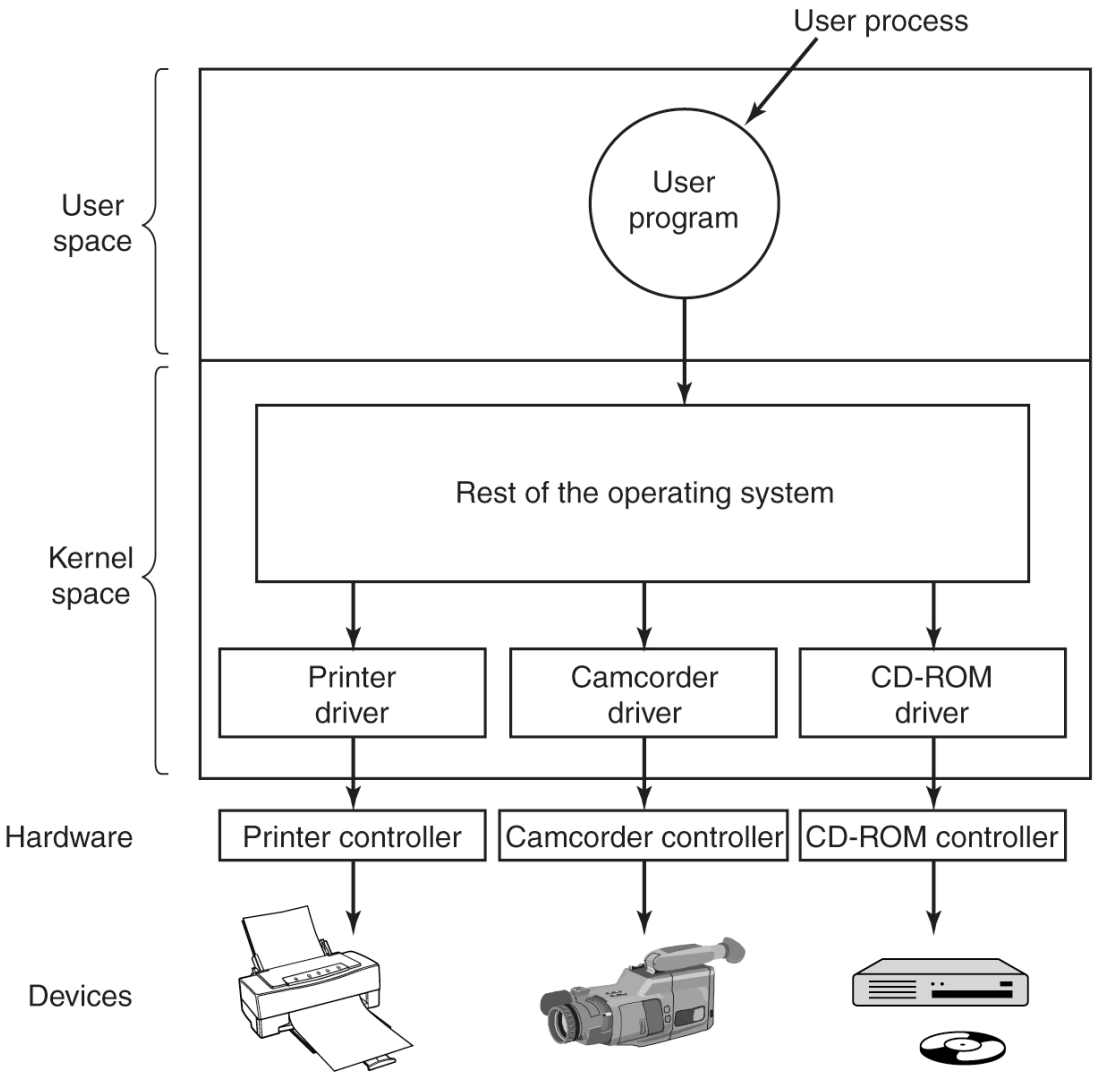
Interrupt Handlers



Device Drivers

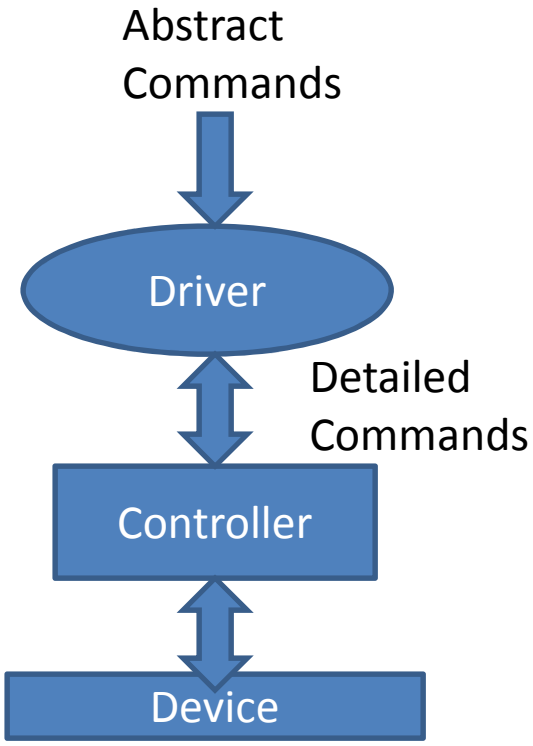
- Device specific code for controlling the device
  - Read device registers from controller
  - Write device registers to issue commands
- Usually supplied by the device manufacturer
- Can be part of the kernel or at user-space (with system calls to access controller registers)
- OS defines a standard interface that drivers for block devices must follow and another standard for driver of character devices

Device Drivers



Device Drivers

- Main functions:
  - Receive abstract read/write from layer above and carry them out
  - Initialize the device
  - Log events
  - Manage power requirements
- Drivers must be reentrant
- Drivers must deal with events such as a device removed or plugged



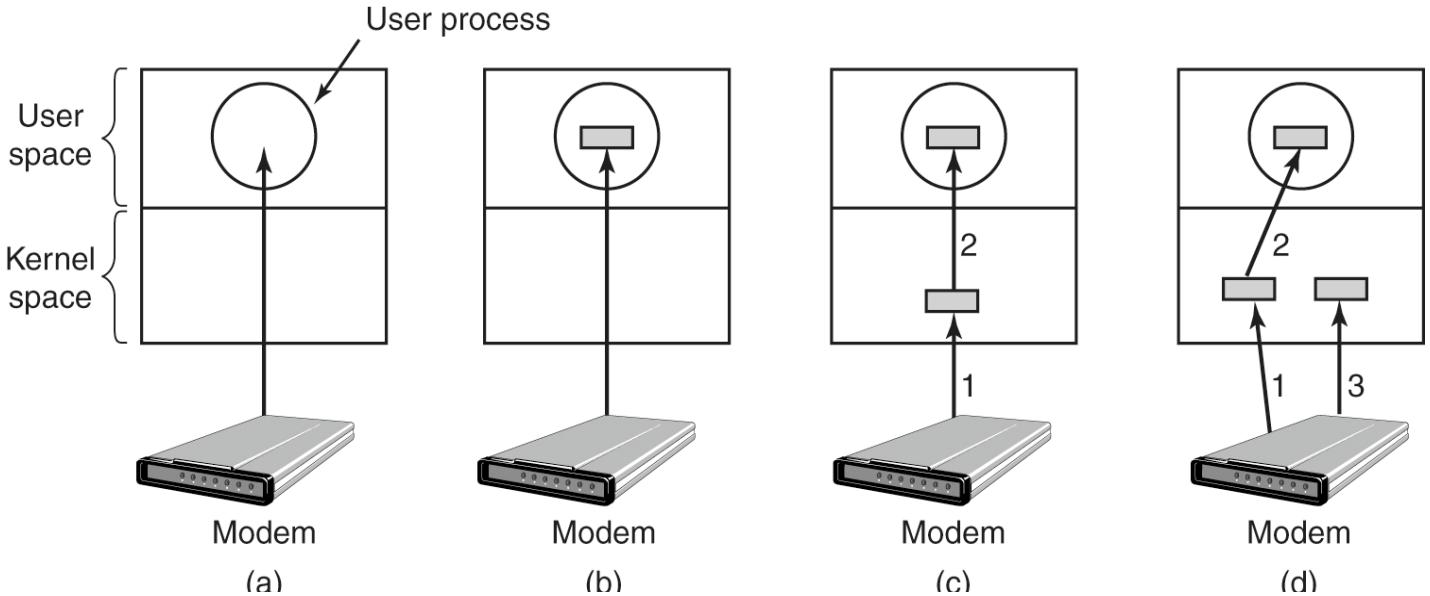
Device Independent I/O Software

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Device Independent I/O Software

- Uniform interfacing for device drivers
  - Trying to make all devices look the same
  - For each class of devices, the OS defines a set of functions that the driver must supply.
  - This layer of OS maps symbolic device names onto proper drivers

Device Independent I/O Software



Interrupt with every character

Very inefficient

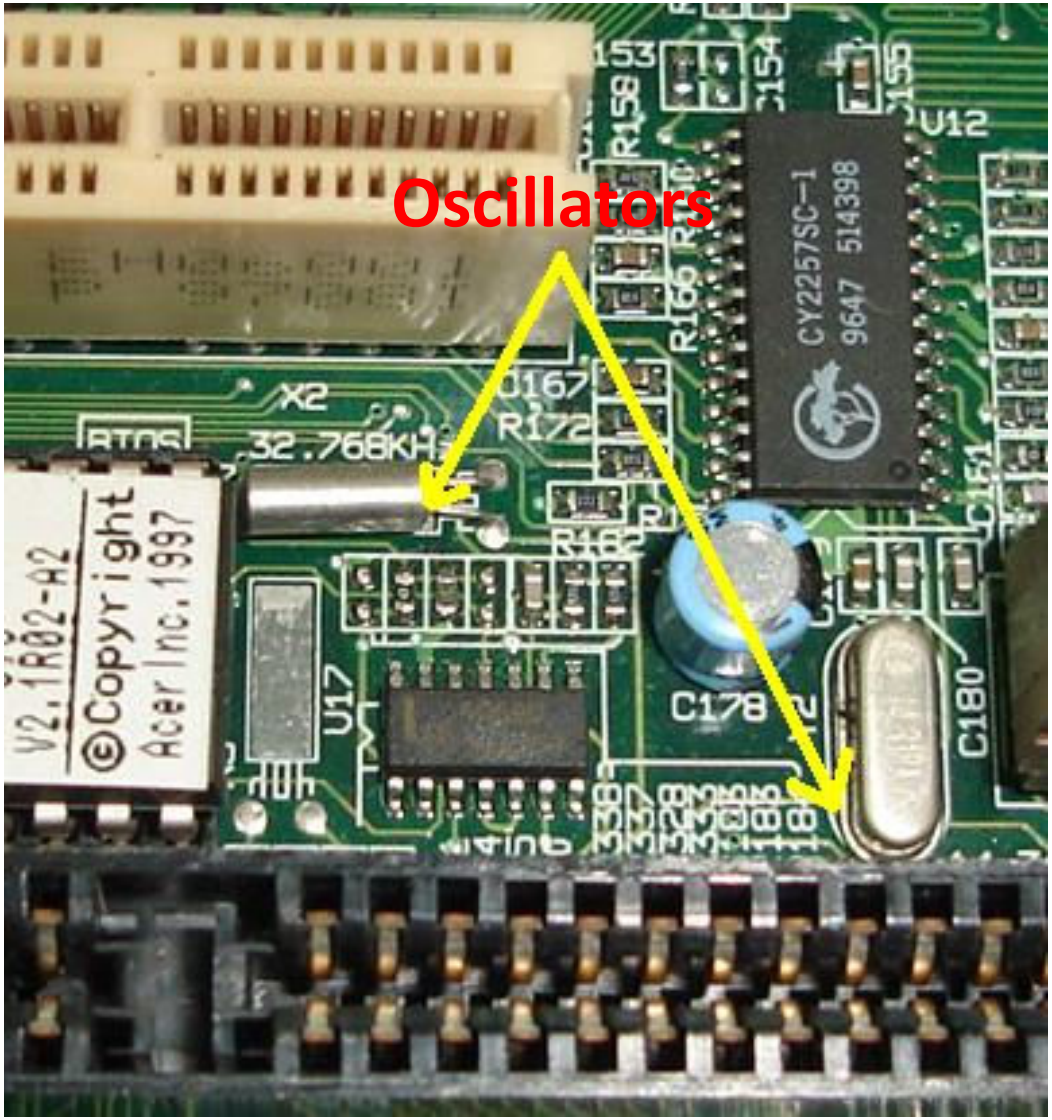
Buffering n char.

What if buffer is paged out?

The need for buffering

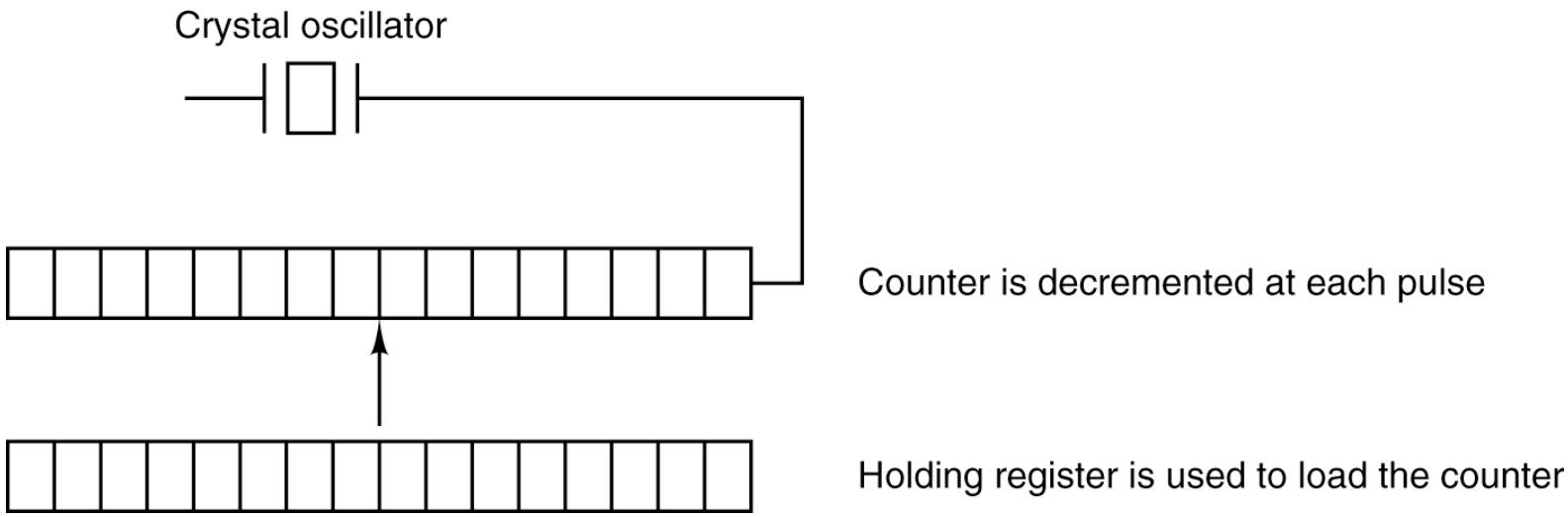
Example of Devices

Clocks


Oscillators
X2
32.768KH
U12
CY2257SC-1
9647 514398
U17
C178
C180
©Copyright
Acer Inc. 1997

Clock Hardware

- Old: tied to the power line and causes an interrupt on every voltage cycle
- New: Crystal oscillator + counter + holding register



Clock Software

- Maintaining the time of the day
- Preventing processes from running longer than they are allowed to
- Accounting for CPU usage
- Handling alarm system call made by user processes
- Providing watchdog timers for parts of the system itself
- Doing profiling, monitoring, and statistics gathering

User Interfaces

- Keyboard
- Mouse
- Monitor

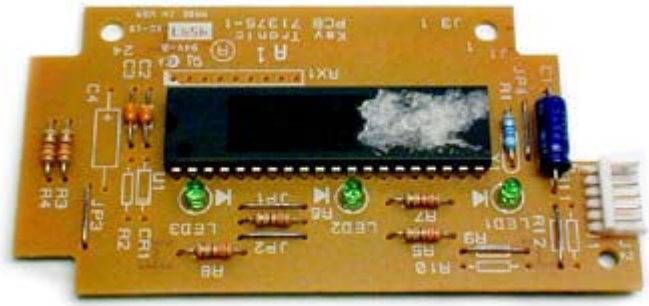
As examples, we will take a closer look at the keyboard and mouse.

Keyboards

- Contains simplified embedded processor that communicates through a port with the controller at the motherboard
- An interrupt is generated whenever a key is struck and a second one whenever a key is released
- Keyboard driver extracts the information about what happens from the I/O port assigned to the keyboard
- The number in the I/O port is called the scan code (7 bits for code + 1 bit for key press/release)

Keyboards

Microprocessor of the keyboard



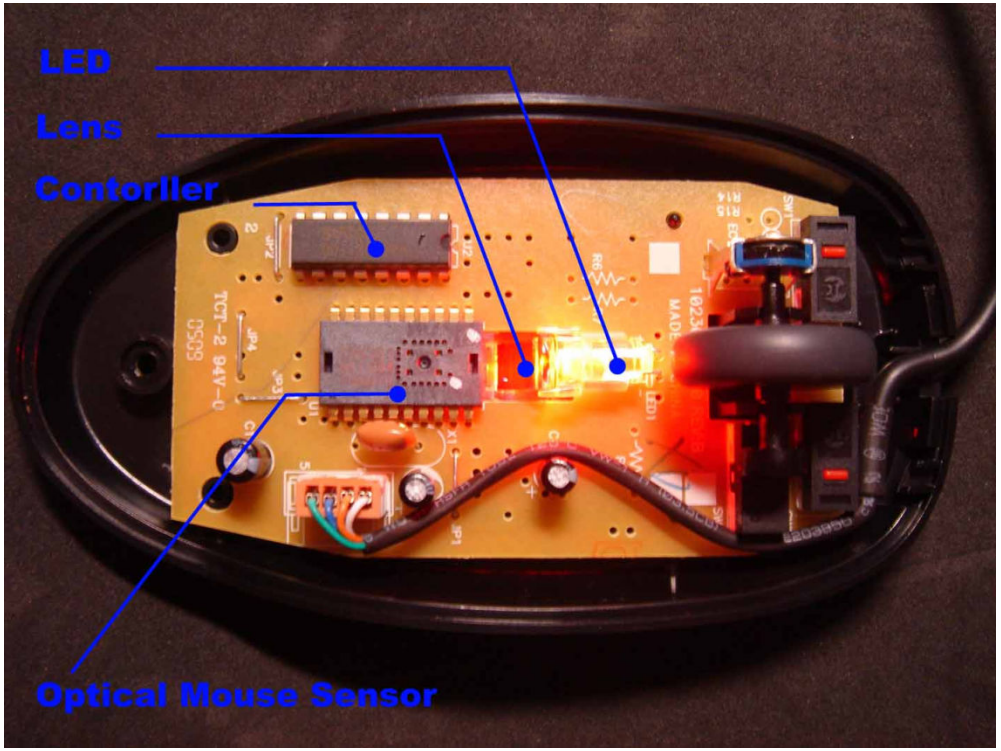
Key matrix

Keyboards

- There are two philosophies for programs dealing with keyboards
  1. The programs gets a raw sequence of ASCII codes (raw mode, or noncanonical mode)
  2. Driver handles all the intraline editing and just delivered corrected lines to the use programs (cooked mode or canonical mode)
- Either way, a buffer is needed to store characters

Mouse

- Mouse only indicates changes in position, not absolute position (delta_x and delta_y)



I/O Software Layer: Principle

- Interrupts are facts of life, but should be hidden away, so that as little of the OS as possible knows about them.
- The best way to hide interrupts is to have the driver starting an IO operation block until IO has completed and the interrupt occurs.
- When interrupt happens, the interrupt handler handles the interrupt.
- Once the handling of interrupt is done, the interrupt handler unblocks the device driver that started it.
- This model works if drivers are structured as kernel processes with their own states, stacks and program counters.

Example for character driver

- http://www.codeproject.com/Articles/112474/A-Simple-Driver-for-Linux-OS

- Registers a kernel module
- Defines the functions that exist
- …

APIs and Datastructures

```
int register_chrdev (unsigned int   major,
                     const char *   name,
                                    const struct   fops);
                                    file_operations *
```

```
struct file_operations {
       struct module *owner;
       loff_t (*llseek) (struct file *, loff_t, int);
       ssize_t (*read) (struct file *, char *, size_t, loff_t *);
       ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
       int (*readdir) (struct file *, void *, filldir_t);
       unsigned int (*poll) (struct file *, struct poll_table_struct *);
       int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
       int (*mmap) (struct file *, struct vm_area_struct *);
       int (*open) (struct inode *, struct file *);
       int (*flush) (struct file *);
       int (*release) (struct inode *, struct file *);
       int (*fsync) (struct file *, struct dentry *, int datasync);
       int (*fasync) (int, struct file *, int);
       int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
          loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
          loff_t *);
    };
```

Declare and Register

```
static struct file_operations simple_driver_fops =
{
        .owner    = THIS_MODULE,
        .read     = device_file_read,
};
```

```
ssize_t device_file_read (struct  file *, char *, size_t, loff_t *);
```

```
static int device_file_major_number = 0;
static const char device_name[] = "Simple-driver";

static int register_device(void)
{
                int result = 0;

                printk( KERN_NOTICE "Simple-driver: register_device() is called." );

                result = register_chrdev( 0, device_name, &simple_driver_fops );
                if( result < 0 )
                {
                        printk( KERN_WARNING "Simple-driver:  can\'t register
                                character device with errorcode = %i", result );
                        return result;
                }

                device_file_major_number = result;
                printk( KERN_NOTICE "Simple-driver: registered character device
                        with major number = %i and minor numbers 0...255"
                         , device_file_major_number );

                return 0;
}
```

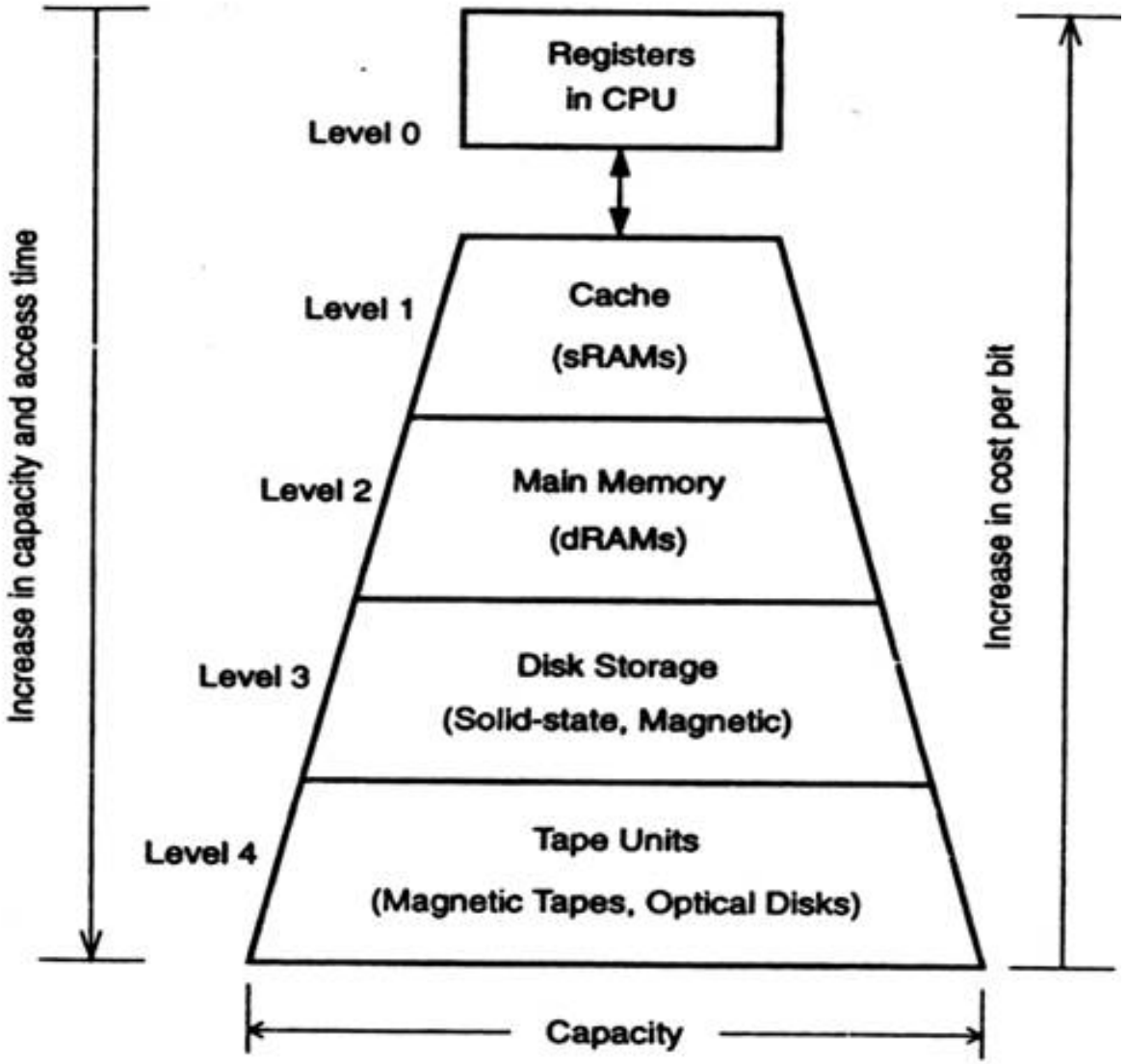
Implement

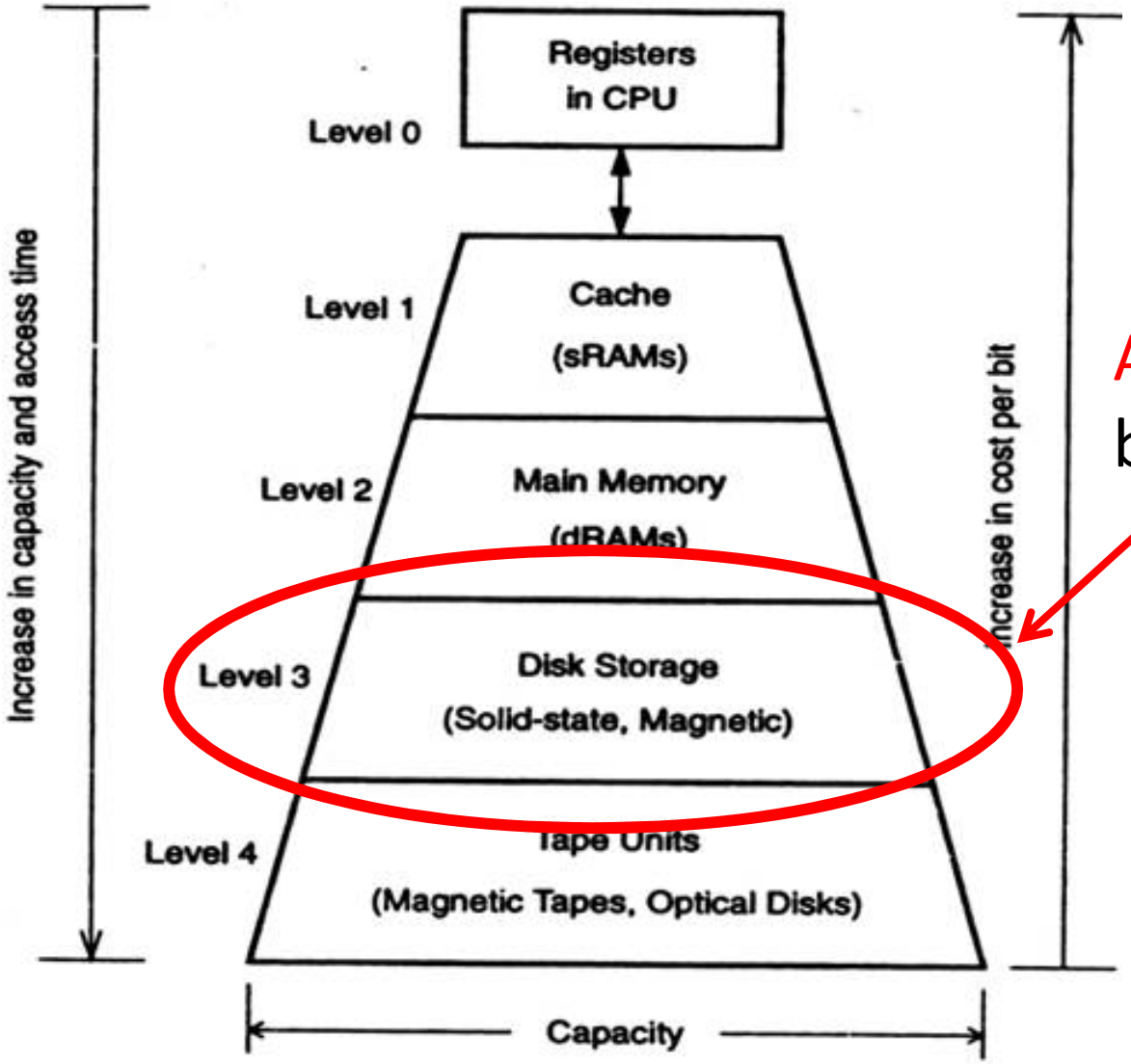
```
static const char    g_s_Hello_World_string[] = "Hello world from kernel mode!\n\0";
static const ssize_t g_s_Hello_World_size = sizeof(g_s_Hello_World_string);

static ssize_t device_file_read(
                                                struct file *file_ptr
                                              , char __user *user_buffer
                                              , size_t count
                                              , loff_t *position)
{
        printk( KERN_NOTICE "Simple-driver:
                Device file is read at offset = %i, read bytes count = %u"
                                , (int)*position
                                , (unsigned int)count );

        /* If position is behind the end of a file we have nothing to read */
        if( *position >= g_s_Hello_World_size )
                return 0;

        /* If a user tries to read more than we have, read only
        as many bytes as we have */
        if( *position + count > g_s_Hello_World_size )
                count = g_s_Hello_World_size - *position;

        if( copy_to_user(user_buffer, g_s_Hello_World_string + *position, count) != 0 )
                return -EFAULT;

        /* Move reading position */
        *position += count;
        return count;
}
```

Conclusions

- The OS provides an interface between the devices and the rest of the system.
- The I/O part of the OS is divided into several layers.
- The hardware: CPU, programmable interrupt controller, DMA, device controller, and the device itself.
- OS must *expand* as new I/O devices are added

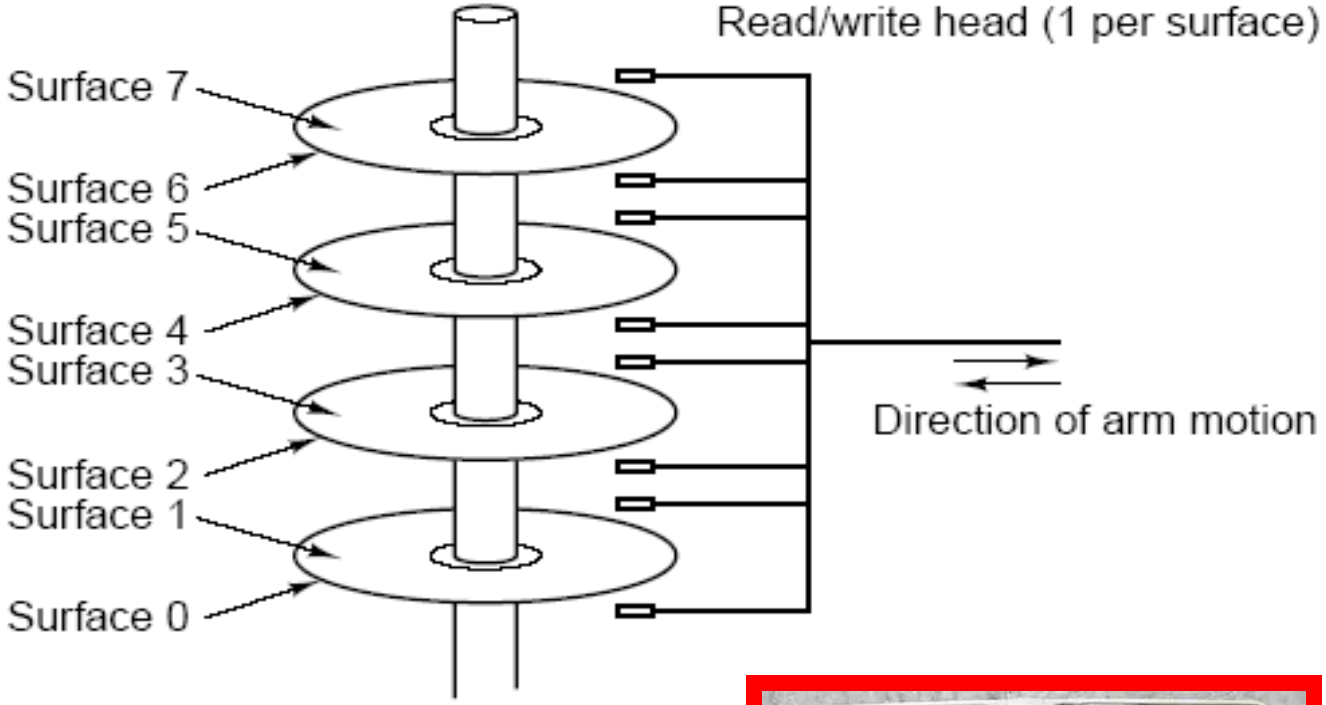
Disks Scheduling

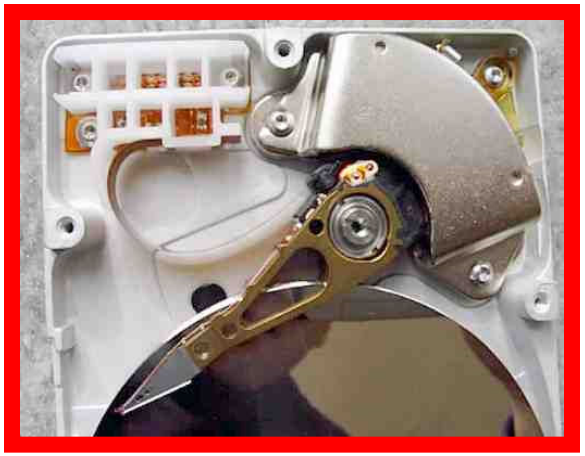

Registers
in CPU
Level 0
Level 1
Cache
(sRAMs)
Level 2
Main Memory
(dRAMs)
Level 3
Disk Storage
(Solid-state, Magnetic)
Level 4
Tape Units
(Magnetic Tapes, Optical Disks)
Increase in capacity and access time
Increase in cost per bit
Capacity

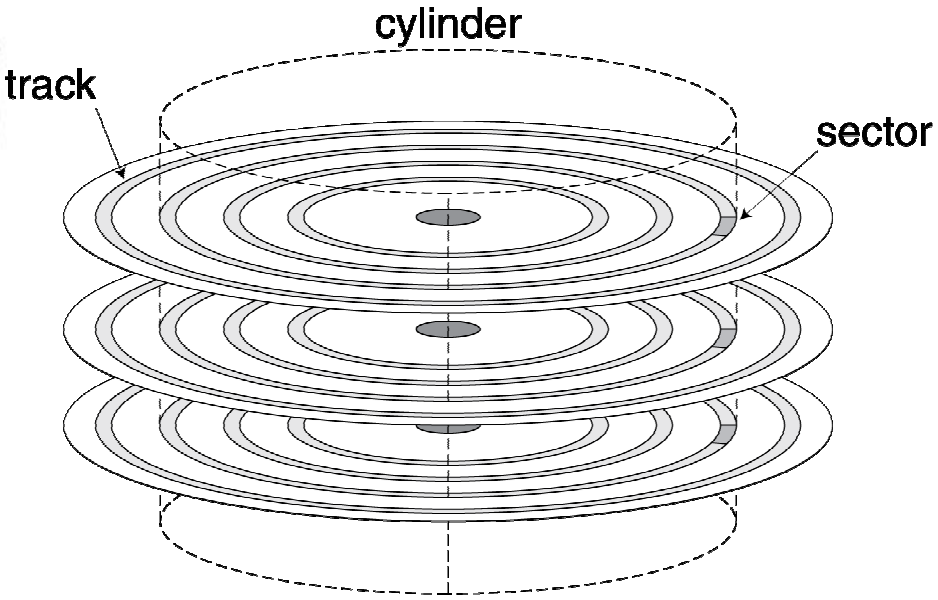


Abstracted
by OS as files

A Conventional Hard Disk (Magnetic) Structure

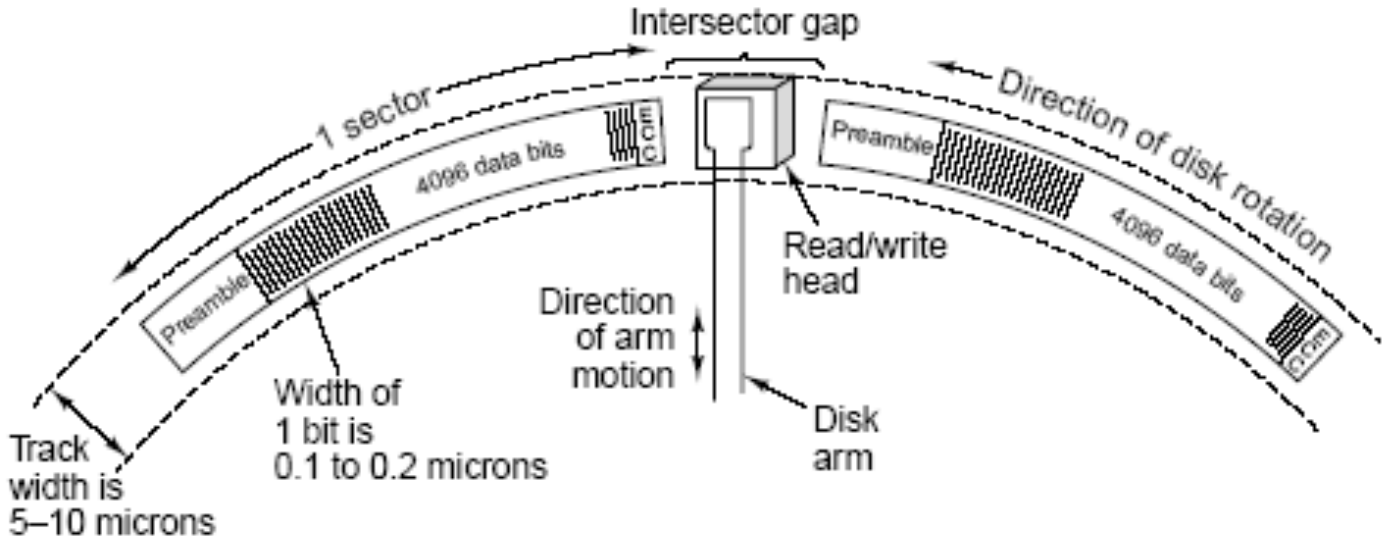





cylinder
track
sector

Hard Disk (Magnetic) Architecture

- **Surface** = group of tracks
- **Track** = group of sectors
- **Sector** = group of bytes
- **Cylinder**: several tracks on corresponding surfaces

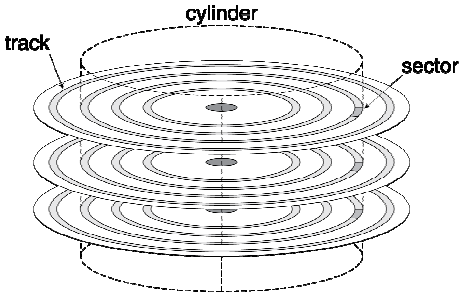


Disk Performance

- 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 bytes
  - Average transfer bandwidth (15-37 Mbytes/sec)
- Performance of transfer 1 Kbytes
  - Seek (5.3 ms) + half rotational delay (3ms) + transfer (0.04 ms)
  - Total time is 8.34ms or 120 Kbytes/sec!
- What block size can get 90% of the disk transfer bandwidth?

Disk Behaviors

- There are more sectors on outer tracks than inner tracks
  - Read outer tracks: 37.4MB/sec
  - Read inner tracks: 22MB/sec
- Seek time and rotational latency dominate the cost of small reads
  - A lot of disk transfer bandwidth is wasted
  - Need algorithms to reduce seek time



Block Size (Kbytes)	% of Disk Transfer Bandwidth
1Kbytes	0.5%
8Kbytes	3.7%
256Kbytes	55%
1Mbytes	83%
2Mbytes	90%

Observations

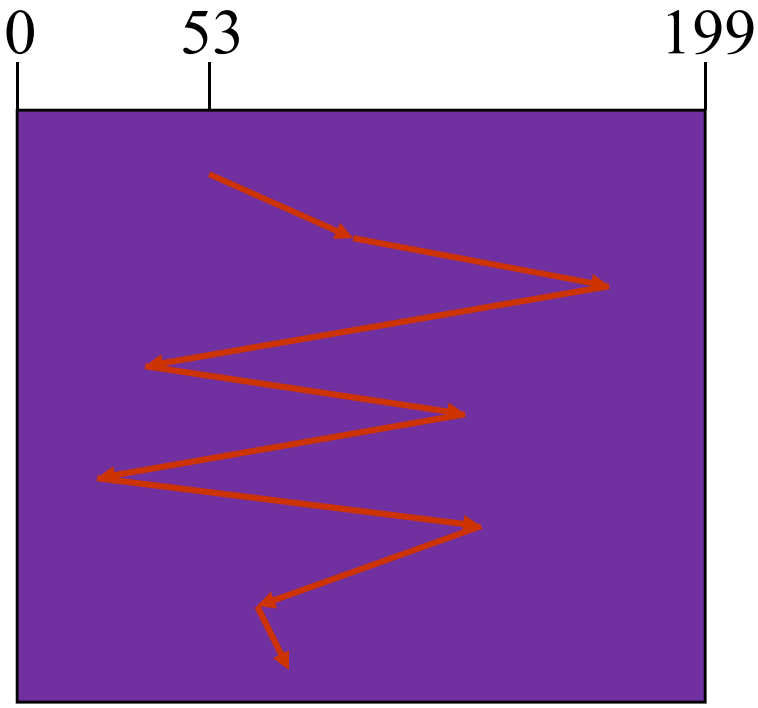
- Getting first byte from disk read is slow
  - high latency
- Peak bandwidth high, but rarely achieved
- Need to mitigate disk performance impact
  - Do extra calculations to speed up disk access
    - Schedule requests to shorten seeks
  - Move some disk data into main memory – file system caching

Disk Scheduling

- Which disk request is serviced first?
  - FCFS
  - Shortest seek time first
  - Elevator (SCAN)
  - LOOK
  - C-SCAN (Circular SCAN)
  - C-LOOK
- Looks familiar?

FIFO (FCFS) order

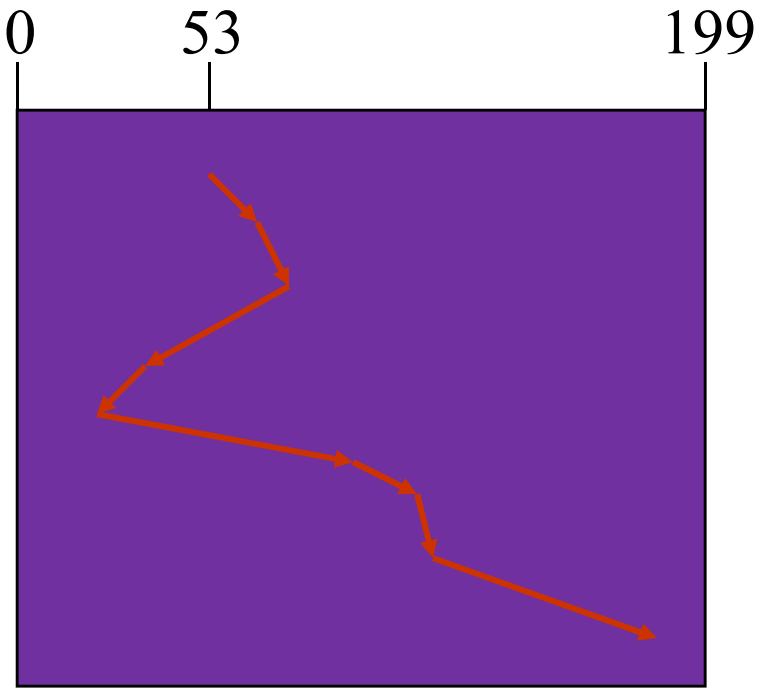
- Method
  - First come first serve
- Pros
  - Fairness among requests
  - In the order applications expect
- Cons
  - Arrival may be on random spots on the disk (long seeks)
  - Wild swing can happen
- Analogy:
  - Can elevator scheduling use FCFS?



98, 183, 37, 122, 14, 124, 65, 67

SSTF (Shortest Seek Time First)

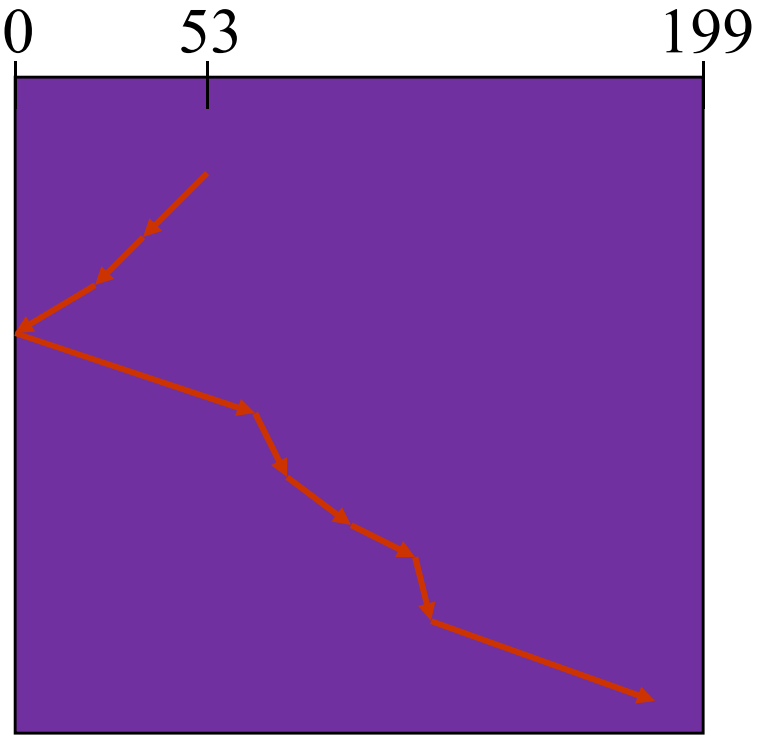
- Method
  - Pick the one closest on disk
  - Rotational delay is in calculation
- Pros
  - Try to minimize seek time
- Cons
  - Starvation
- Question
  - Is SSTF optimal?
  - Can we avoid starvation?
- Analogy: elevator



98, 183, 37, 122, 14, 124, 65, 67
(65, 67, 37, 14, 98, 122, 124, 183)

Elevator (SCAN)

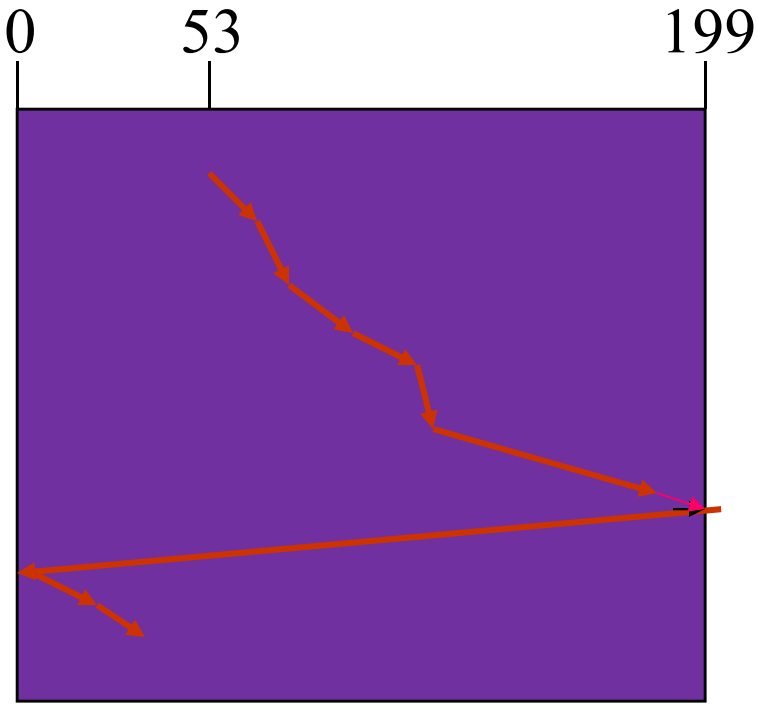
- Method
  - Take the closest request in the direction of travel
  - Real implementations do not go to the end (called LOOK)
- Pros
  - Bounded time for each request
- Cons
  - Request at the other end will take a while



98, 183, 37, 122, 14, 124, 65, 67
(37, 14, 0, 65, 67, 98, 122, 124, 183)

C-SCAN (Circular SCAN)

- Method
  - Like SCAN
  - But, wrap around
  - Real implementation doesn't go to the end (C-LOOK)
- Pros
  - Uniform service time
- Cons
  - Do nothing on the return



98, 183, 37, 122, 14, 124, 65, 67

(65, 67, 98, 122, 124, 183, 199, 0, 14, 37)

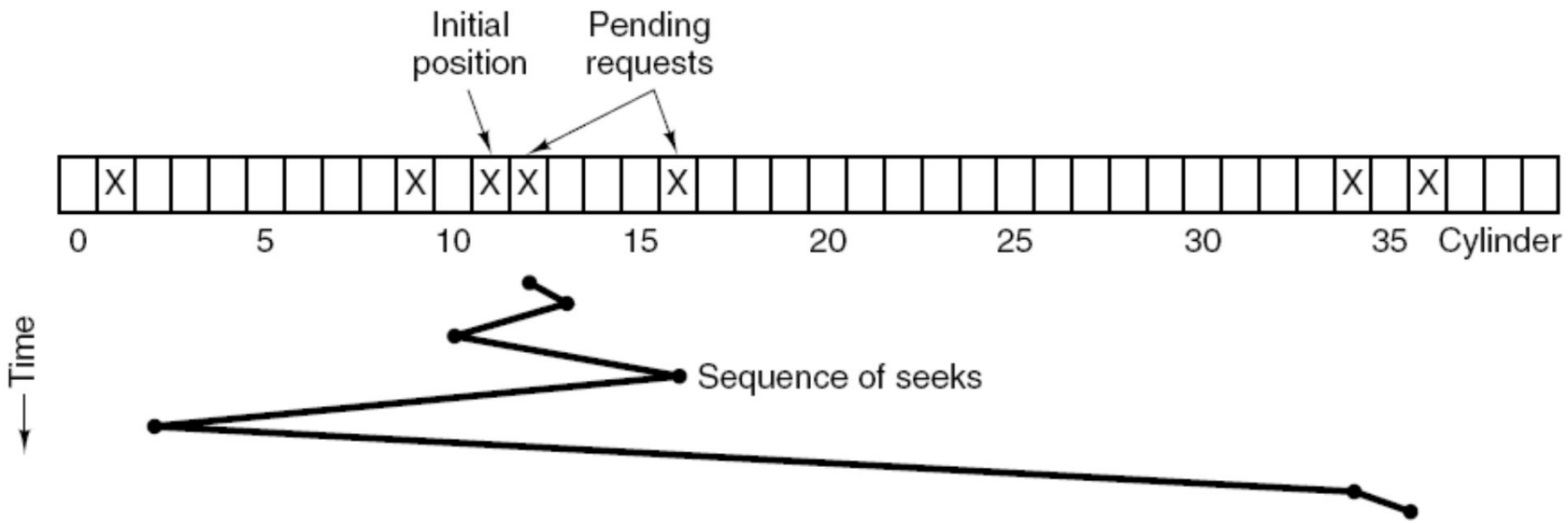
LOOK and C-LOOK

- SCAN and C-SCAN move the disk arm across the full width of the disk
- In practice, neither algorithm is implemented this way
- More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without first going all the way to the end of the disk.
- These versions of SCAN and C-SCAN are called LOOK and C-LOOK

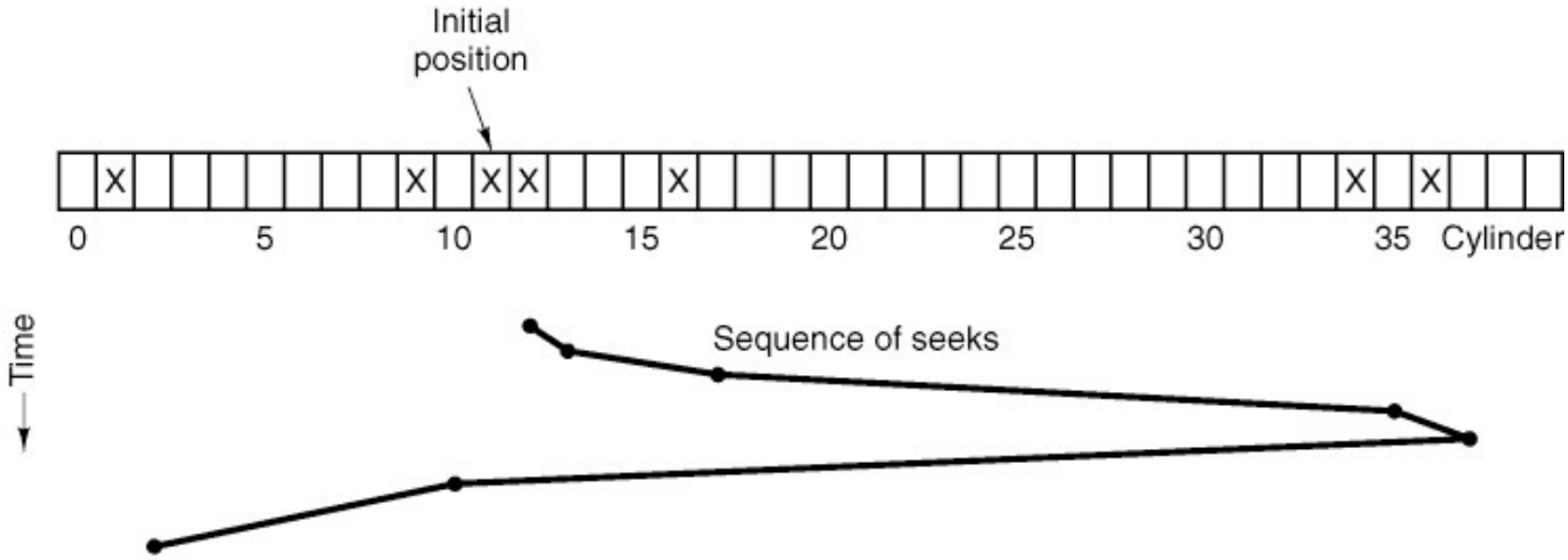
FSCAN

- Like SCAN … but
- Operates with two queues:
- While I/O is ongoing from one queue any new I/O request is entered into the second queue.
- When first queue is fully processes, the second will be processed and any new incoming requests are entered into the first queue and so forth…

Which algorithm is this?



Which algorithm is this?



RAID

RAID

- *Redundant Array of Independent Disks* OR *Redundant Array of Inexpensive Disks*
- Use parallel processing to speed up CPU performance
- Use parallel I/O to improve disk performance, reliability (1988, Patterson)
- Design new class of I/O devices called RAID – Redundant Array of Inexpensive Disks (also Redundant Array of Independent Disks)
- Use the RAID in OS as a SLED (Single Large Expensive Disk), but with better performance and reliability

Common Hard Drive Errors

1. Programming error
   - request for nonexistent sector
2. Transient checksum error
   - caused by dust on the head
3. Permanent checksum error
   - disk block physically damaged
4. Seek error
   - the arm sent to cylinder 6 but it went to 7
5. Controller error
   - controller refuses to accept commands

Hard Disk Driver in Linux

Register	Read Function	Write Function
0	Data	Data
1	Error	Write Precompensation
2	Sector Count	Sector Count
3	Sector Number (0-7)	Sector Number (0-7)
4	Cylinder Low (8-15)	Cylinder Low (8-15)
5	Cylinder High (16-23)	Cylinder High (16-23)
6	Select Drive/Head (24-27)	Select Drive/Head (24-27)
7	Status	Command

(a)

- (a) The control registers of an IDE hard disk controller. The numbers in parentheses are the bits of the logical block address selected by each register in LBA mode.

Hard Disk Driver in Linux

- (b) The fields of the Select Drive/Head register.

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA: 0 = Cylinder/Head/Sector Mode
1 = Logical Block Addressing Mode
D: 0 = master drive
1 = slave drive
HSn: CHS mode: Head select in CHS mode
LBA mode: Block select bits 24 - 27

(b)

RAID

- RAID consists of RAID SCSI controller plus a box of SCSI disks
- Data are divided into strips and distributed over disks for parallel operation
- RAID 0 … RAID 5 levels
- RAID 0 organization writes consecutive strips over the drives in round-robin fashion – operation is called striping
- RAID 1 organization uses striping and duplicates all disks
- RAID 2 uses words, even bytes and stripes across multiple disks; uses error codes, hence very robust scheme
- RAID 3, 4, 5 alterations of the previous ones

Small Computer System Interface (**SCSI**, /ˈskʌzi/ ***SKUZ****-ee*) is a set of standards for physically connecting and transferring data between computers and peripheral devices. The SCSI standards define commands, protocols, electrical and optical interfaces. (source Wikipedia)

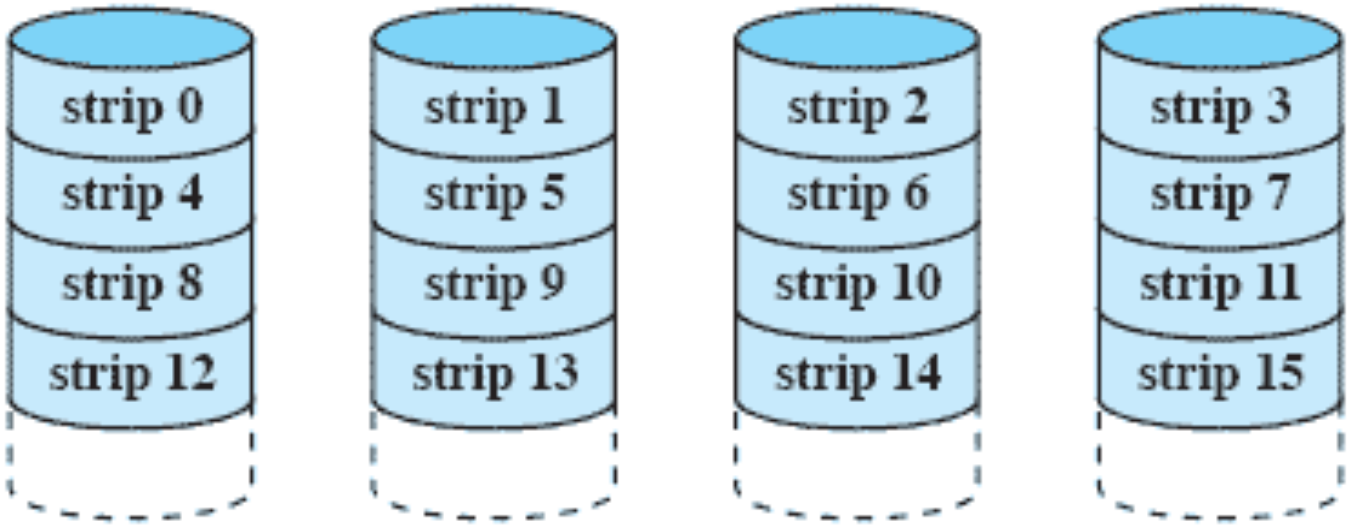
Table 11.4 RAID Levels

Category	Level	Description	Disks required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

N = number of data disks; m proportional to log N

RAID Level 0

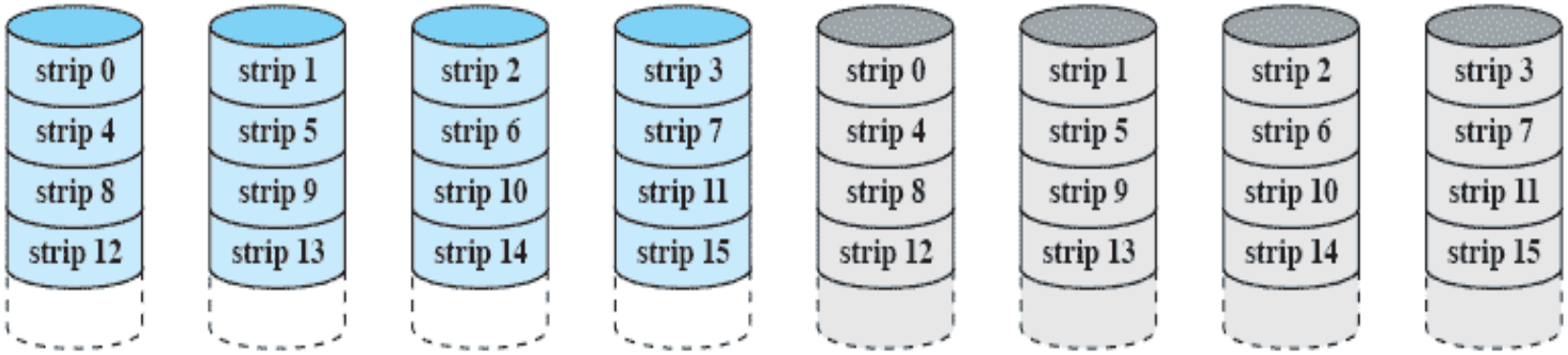
- Not a true RAID because it does not include redundancy to improve performance or provide data protection
- User and system data are distributed across all of the disks in the array
- Logical disk is divided into strips



(a) RAID 0 (non-redundant)

RAID Level 1

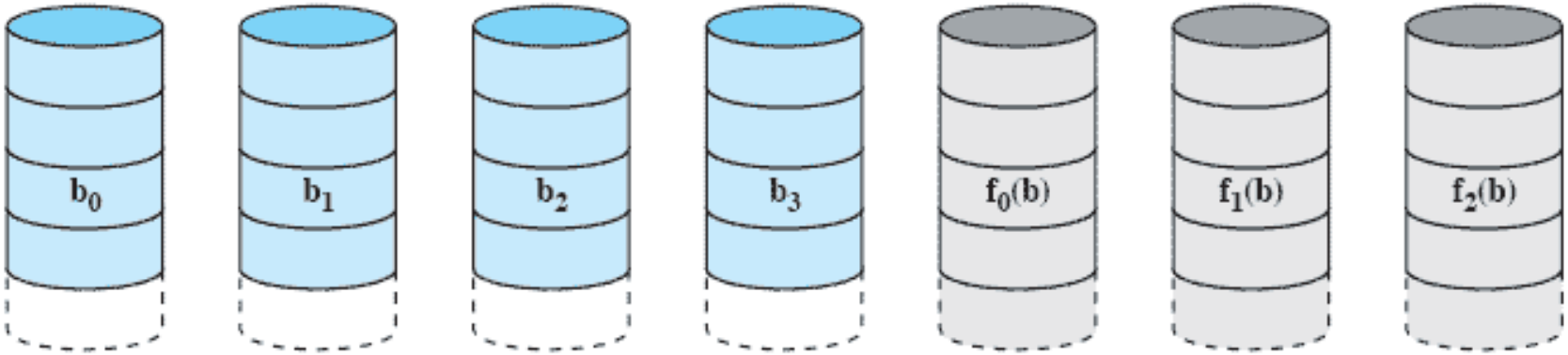
- Redundancy is achieved by the simple expedient of duplicating all the data
- There is no "write penalty"
- When a drive fails the data may still be accessed from the second drive
- Principal disadvantage is the cost



(b) RAID 1 (mirrored)

RAID Level 2

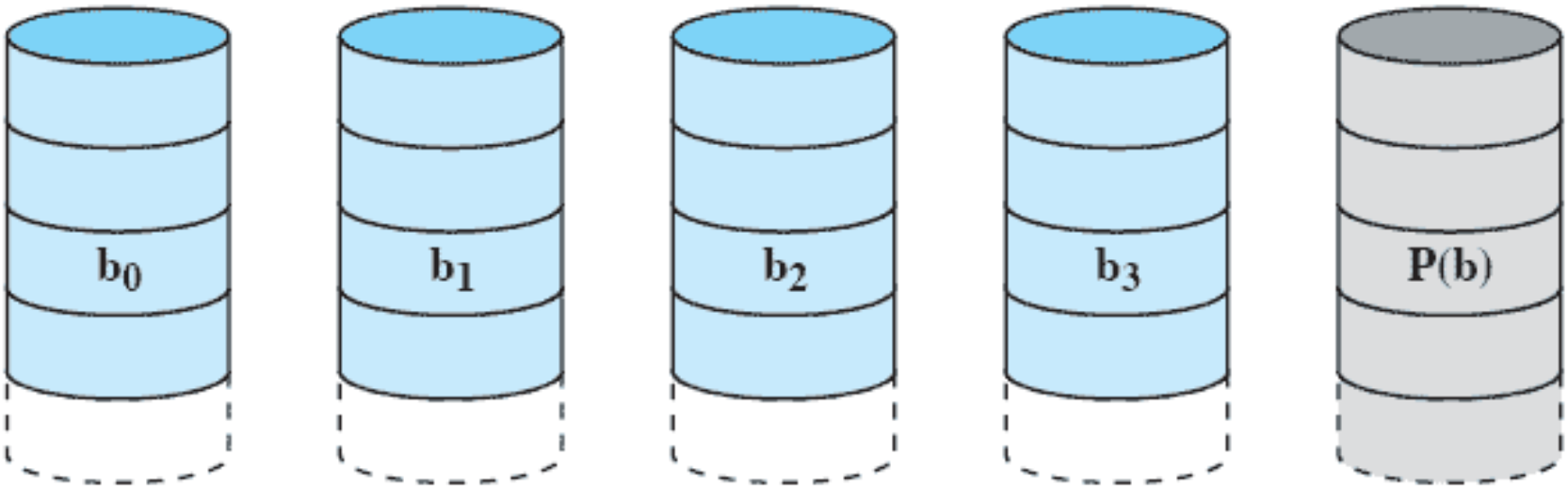
- Makes use of a parallel access technique
- Data striping is used
- Typically a Hamming code is used
- Effective choice in an environment in which many disk errors occur
- Can correct-single bit, detect 2-bit
- On read all disks are read



(c) RAID 2 (redundancy through Hamming code)

RAID Level 3

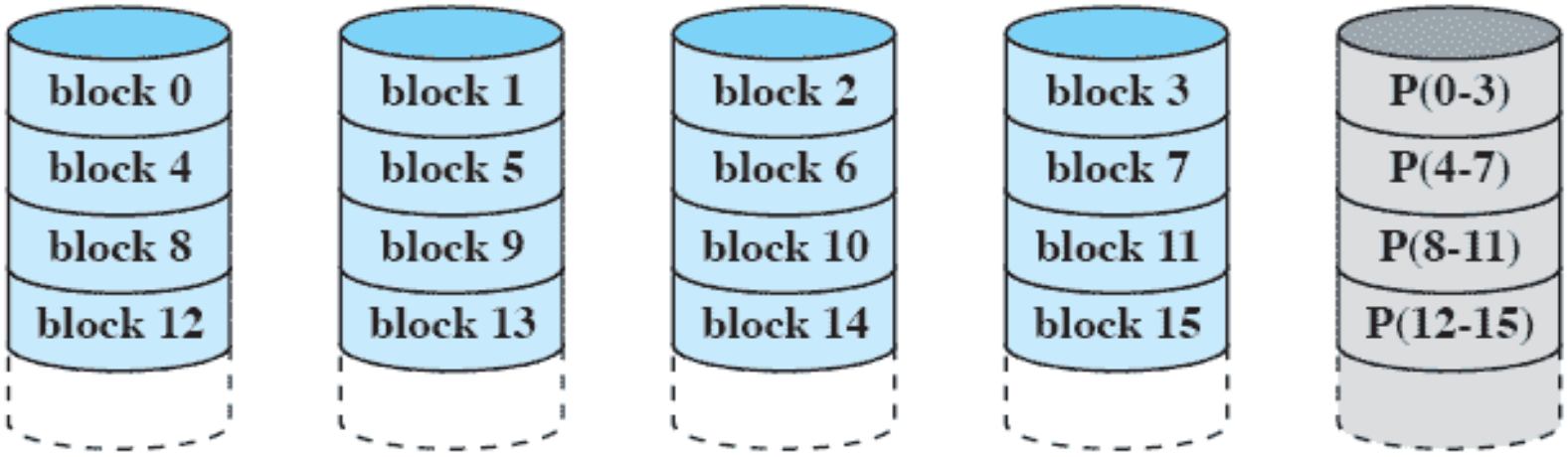
- Requires only a single redundant disk, no matter how large the disk array
- Employs parallel access, with data distributed in small strips
- Can achieve very high data transfer rates



(d) RAID 3 (bit-interleaved parity)

RAID Level 4

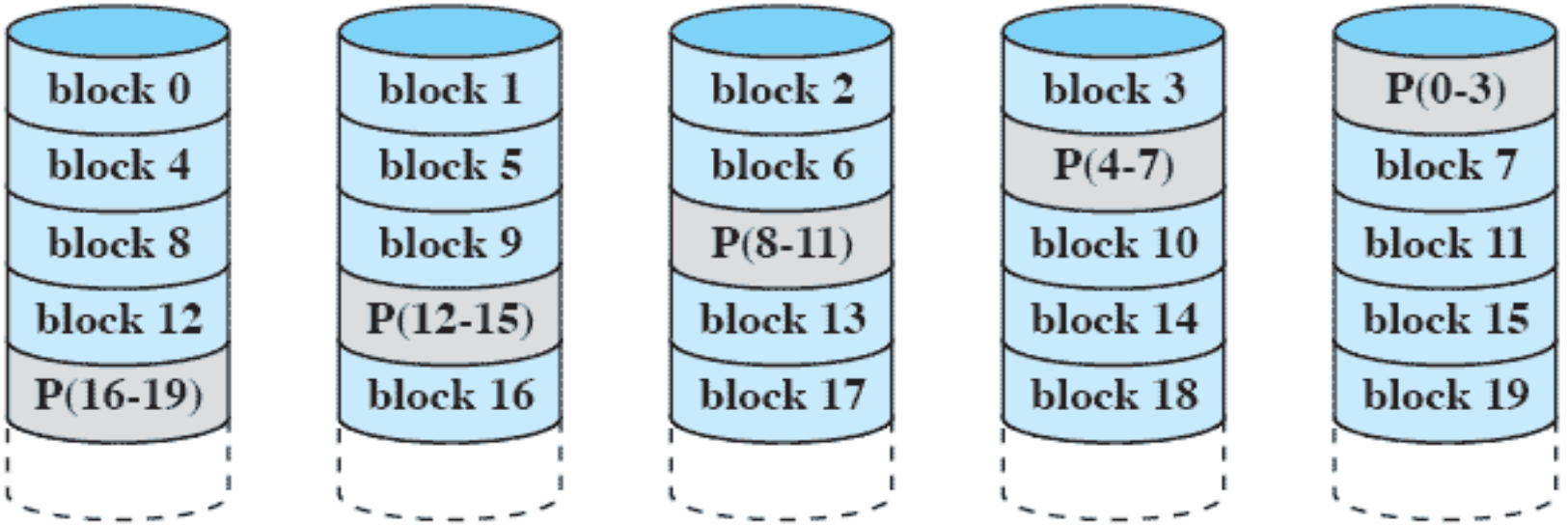
- Makes use of an independent access technique
- A bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk
- Involves a write penalty when an I/O write request of small size is performed



(e) RAID 4 (block-level parity)

RAID Level 5

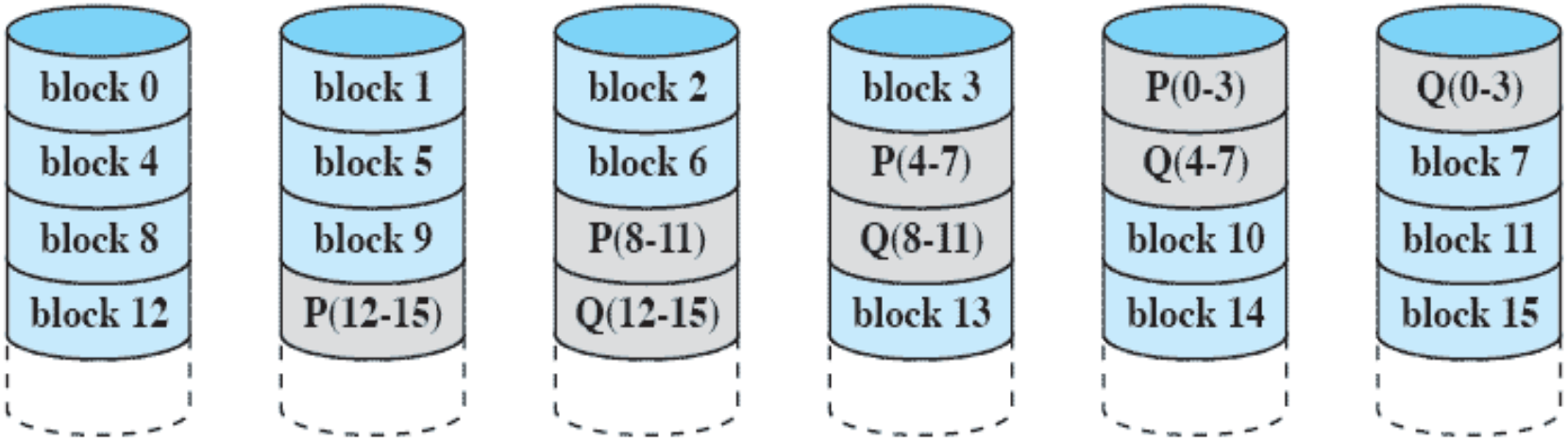
- Similar to RAID-4 but distributes the parity bits across all disks
- Typical allocation is a round-robin scheme
- Has the characteristic that the loss of any one disk does not result in data loss



(f) RAID 5 (block-level distributed parity)

RAID Level 6

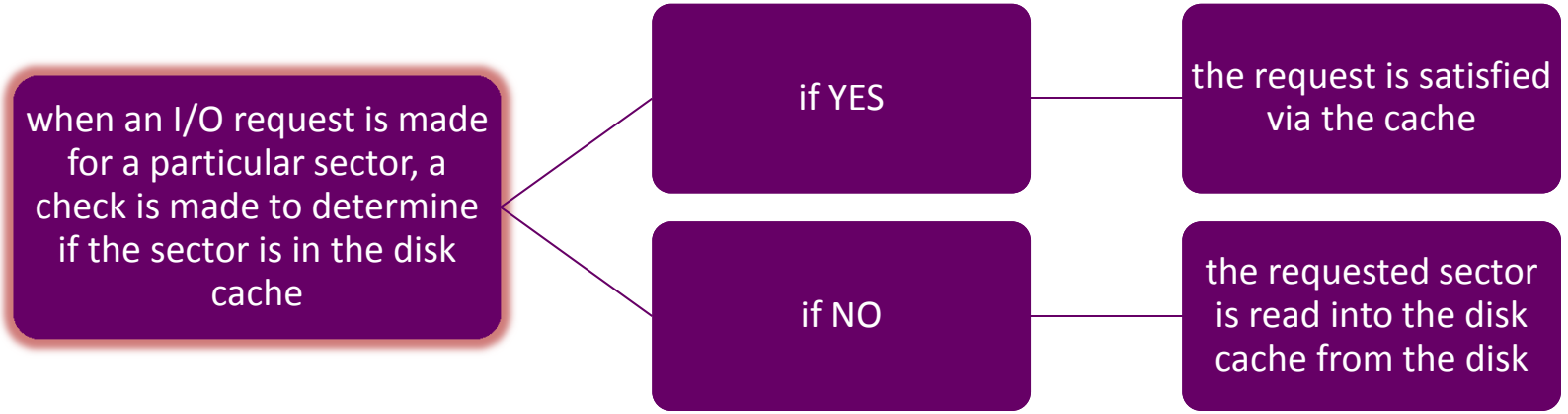
- Two different parity calculations are carried out and stored in separate blocks on different disks
- Provides extremely high data availability
- Incurs a substantial write penalty because each write affects two parity blocks



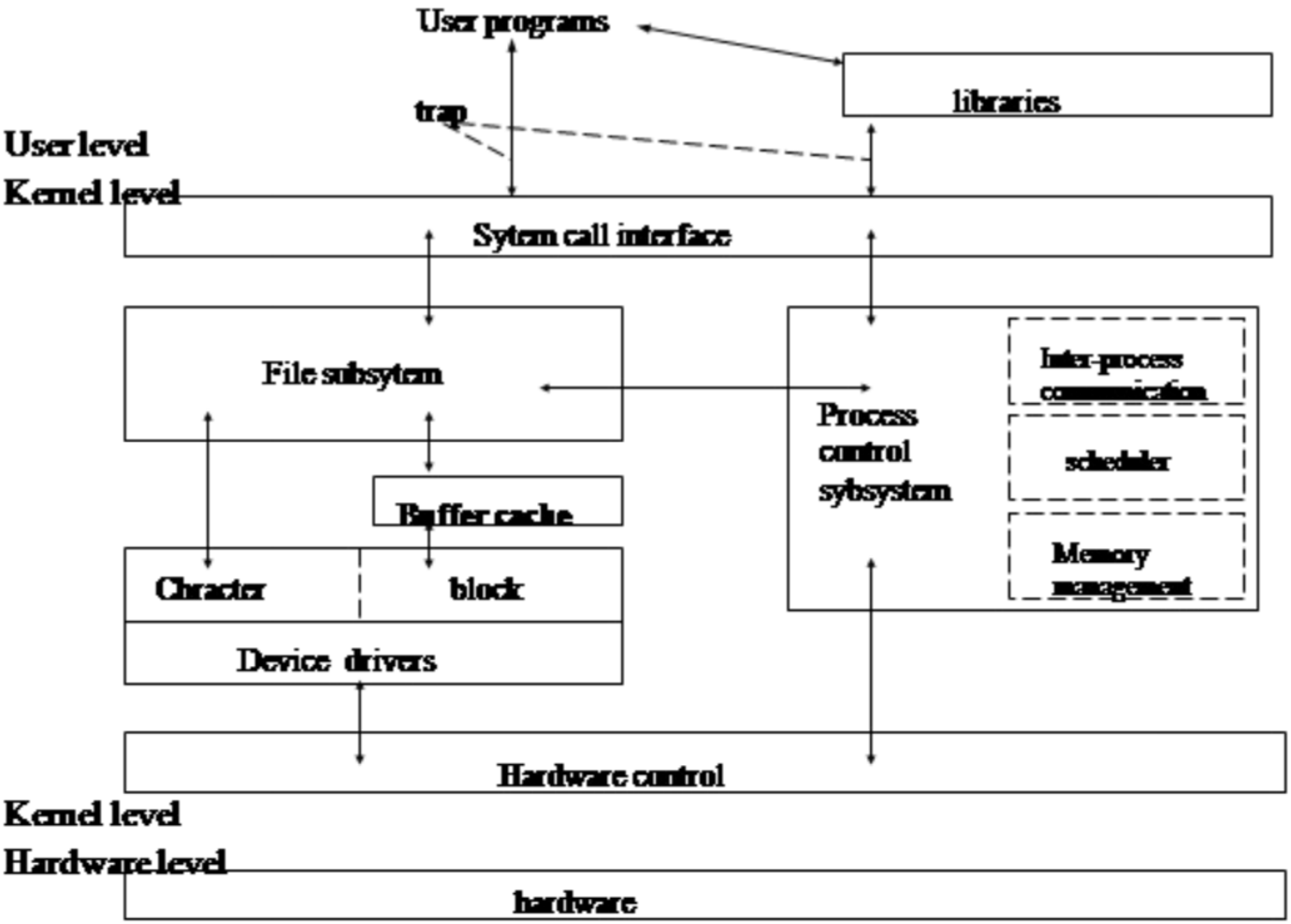
(g) RAID 6 (dual redundancy)

Disk Cache (aka buffer cache)

- *Cache memory* is used to apply to a memory that is smaller and faster than main memory and that is interposed between main memory and the processor
- Reduces average memory access time by exploiting the principle of locality
- *Disk cache* is a buffer in main memory for disk sectors
- Contains a copy of some of the sectors on the disk



Disk or Buffer Cache



Solid State Disks and FLASH Memory

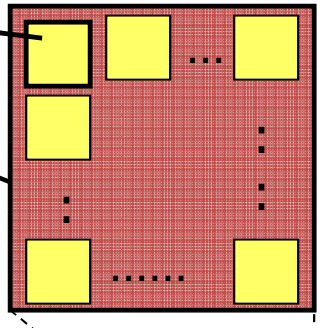
- Solid State Disks
  - Made out of FLASH Memory
  - Genealogy of FLASH
    - RAM, EPROM, EEPROM
      - RAM needs power
      - EPROM needs no power but could be programmed only once
      - EEPROM → erased and programmed, maintain the programmed value without power
      - 'ROM' → Read-Only-Memory: this term is used because although we could read any arbitrary location, entire 'block' needs to be erased at once.
    - Usage of EEPROMs
      - Historically, programs for embedded processors [which needed to be programmed once in a while]
      - Programming an EEPROM was not a time critical event in the past
      - Limitations on the number of times EEPROMs could be programmed ranged from 100s to 1000s and that was not a problem as typical embedded systems were programmed just a dozen times.
    - Evolution of Technology
      - Thousands of EEPROM circuits (called "blocks" in FLASH) are arrayed in order to randomly program and erase blocks

Organization of a Typical FLASH (Single-Layered) Chip

- 1 Page = 2KB

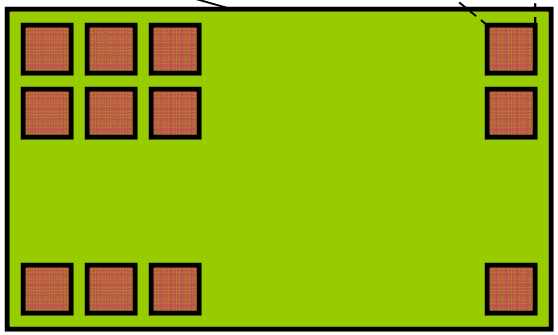
- 1 Block = 128KB

  (64 pages)



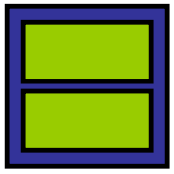
- **1 Plane = 2K Blocks**

  = 256MB



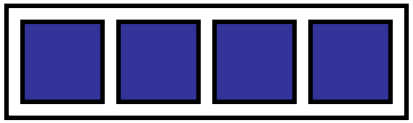
- **1 Die = 2 Planes**

  = 512MB



- **1 Flash Chip =** 2GB

  (4 Dies)



- **Functions supported**
  - Read
  - Erase
  - Program

FLASH Chip: A few details

- Operations on a FLASH Chip
  - 3 main operations: Read/Erase/Program. Write = Erase + Program.
  - Each set of dies in a chip-enable group operate independently as long as the shared data pins are not busy
  - With in a chip-enable group, only one of these operations can take place on a plane
  - Separate operations can take place in separate planes in parallel as long as they are submitted to the chip-enable group at once (this is because once a command is sent to the chip-enable group, it signals busy until the command is complete)
- Types of FLASH Chips
  - SLC: Single Layered Chip
    - Stores 0/1 in each cell [2 voltages]
    - Faster and more reliable
  - MLC: Multi Layered Chip
    - Stores 00/01/10/11 in each cell [4 voltages] ➔ 2 bits per cell
    - Fails 10 times sooner!
    - Cost Advantage of roughly 2 to 1.
  - Manufacturing process almost same for both SLC and MLC.

FLASH Memory is of 2 types : NAND and NOR

- Based on the type of logic gate used in the cell
  - NAND Flash
    - Has faster erase and write times
    - Higher density, and
    - Lower cost per bit than NOR Flash
    - About ten times the endurance.
    - Most NAND Flash manufacturers today claim a write endurance of 1 million write cycles per block

Quick Look at various operations on FLASH

	Read	Erase	Program
Granularity	Page	Block	Set of Pages with in a Block (entire Block need not be programmed at once.)
Access	Random	Random blocks within a Chip	❑ Any block in a chip can be programmed randomly ❑ Set of pages with in a block need to be programmed sequentially
Time	< 0.1ms	1.5ms	>= 0.3ms
'Wear Out'?	No	Yes	No (but you can program only once before erase)
Bit Change	-	Changes all Bits ➔'1'	Only op to change Bits ➔ '0'
Other Notes		❑ 'Stresses' the block and will eventually cause it to fail. SLC Chips: 98% of blocks will last at least 100,000 erase cycles ❑ Large systems have 'wear-leveling' algorithms built in so that 'system write' has high endurance than that of write to each cell	

Example: Sample Write Operation (source: Texas Memory Systems)

- Writing one 8KB random write would require the following steps:

Step 1	Read 128KB (1 block)	**0.1ms + transfer time** **(trsfr time = 128KB/(20MB/s) = 6.25ms)**	6.4ms
Step 2	Erase 128KB Block		1.5ms
Step 3	Program 128KB Block (with 120KB of unchanged data and 8KB of new data)	**0.3ms + 6.25ms + (?)**	6.7ms
Data Transfer rate of this FLASH chip ~ 20MB/s		**Total Time**	**14.6ms**

- **Write Performance (of 14-15ms)**
  - Acceptable in consumer markets line thumb (USB) drives
  - Not Acceptable in Enterprise markets ➔ *one of the reasons for not deploying Flash drives in enterprises*

Difference between SSDs and traditional Hard Drives

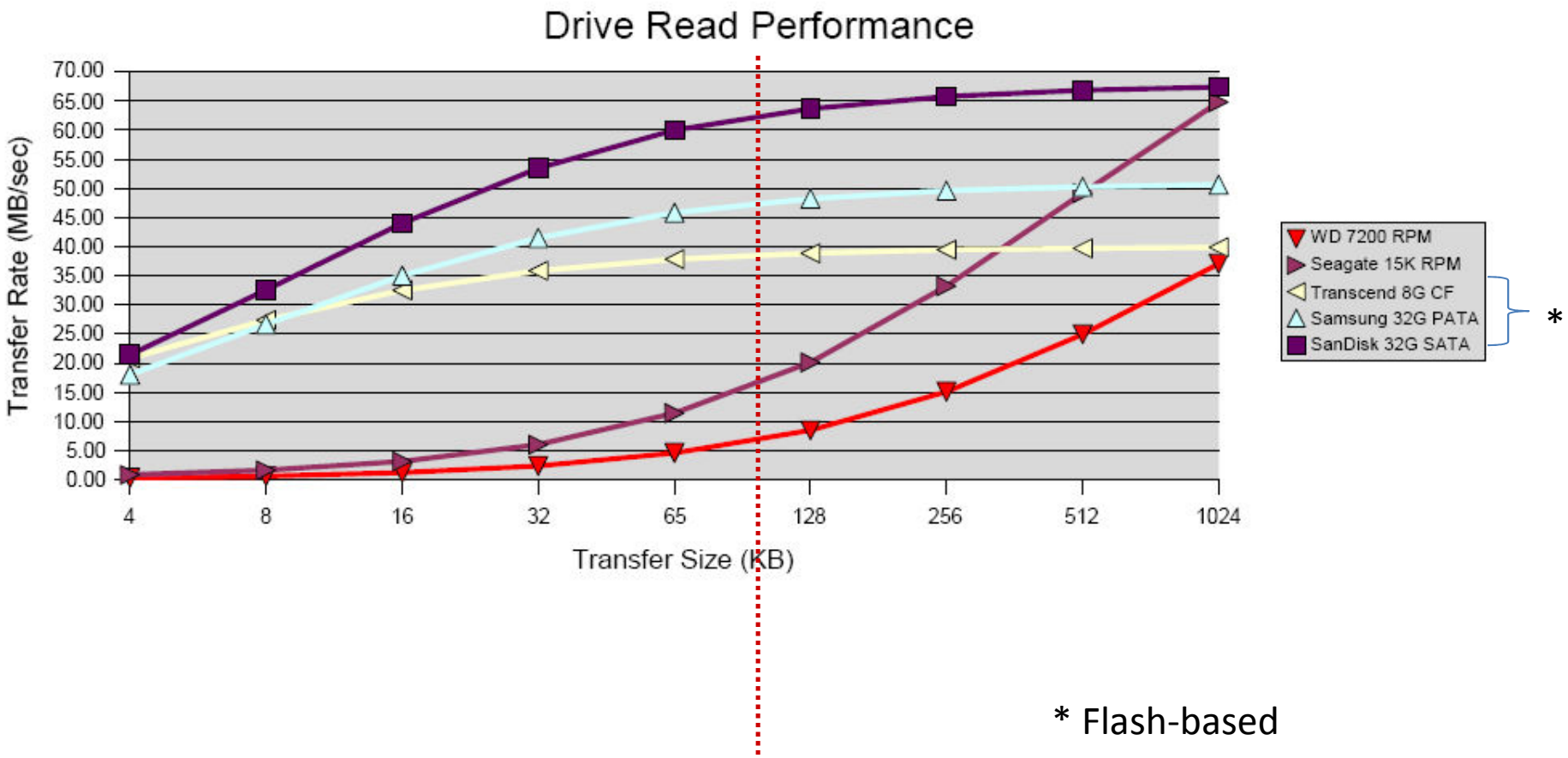
	Hard Drive	Solid State Drive
Construction/ Data Organization	Mechanical Motor/Spindle, Head, Cylinders. Data organized as Tracks and Sectors	FLASH Memory. Constructed using 'blocks'. No spinning. Electronic Reading and Writing.
Reads	❑ Slow Random Reads	Fast Random/Sequential Reads
	❑ Pre-fetching to help Sequential Reads (initiated by OS/present in Disk itself)	——
Writes	Slow Random Writes	Slow Random Writes (slower than current state-of-the-art Hard Disks) → 14-15ms for 128KB.
	Disk Cache/OS Coalescing could help buffer a few writes	Remapping blocks to convert random writes to sequential writes is one trend
Failures	Complete Disk Failure	Block Failure
Recovery/Failure prevention method	Implemented at an array level. RAID is the normal way to re-construct failed disks	Can be done at a disk level or array level ❑ Current methods at a disk level
	——	Wear-leveling algorithms to reduce possible failures of blocks
Striping for arrays	Typically Large Stripe Sizes preferred	Smaller Stripe Sizes preferred

Read Performance
(source: Understanding FLASH SSD Performance, EasyCo LLC)

	Drive Model	Description	Seek Time			Latency	Read XFR Rate		Write XFR Rate	
			Track to Track	Average	Full Stroke		Outer Tracks	Inner Tracks	Outer Tracks	Inner Tracks
Hard Drives	Western Digital WD7500AYYS	7200 RPM 3.5" SATA	0.6 ms	8.9 ms	12.0 ms	4.2 ms	85 MB/sec	60 MB/sec*	85 MB/sec	60 MB/sec*
	Seagate ST936751SS	15K RPM 2.5" SAS	0.2 ms	2.9 ms	5.0 ms*	2.0 ms	112 MB/sec	79 MB/sec	112 MB/sec	79 MB/sec
Flash SSDs	Transcend TS8GCF266	8GB 266x CF Card	0.09ms				40 MB/sec		32 MB/sec	
	Samsung MCAQE32G5APP	32G 2.5" PATA	0.14ms				51 MB/sec		28 MB/sec	
	Sandisk SATA5000	32G 2.5" SATA	0.125ms				68 MB/sec		40 MB/sec	

* Figure is an estimate

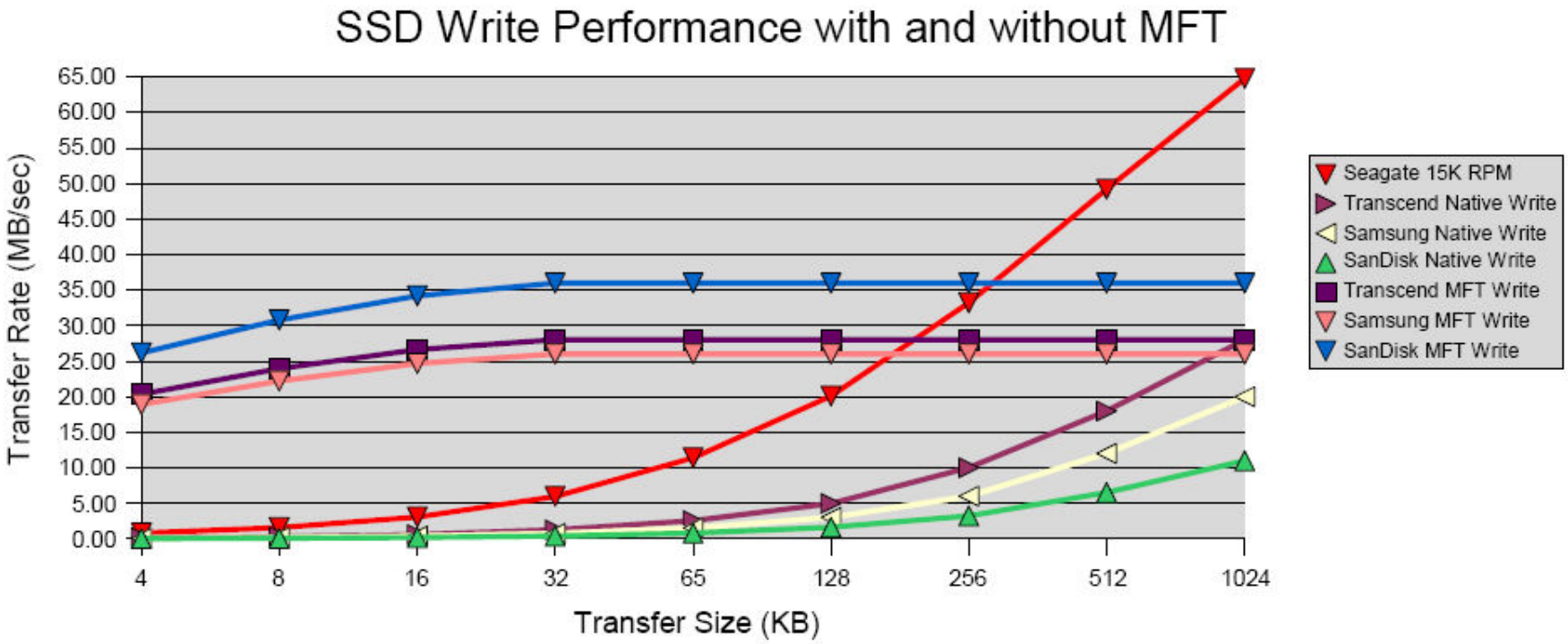
Read Performance
(source: Understanding FLASH SSD Performance, EasyCo LLC)



* Flash-based

Write Performance
(source: Understanding FLASH SSD Performance, EasyCo LLC)

MFT: Managed Flash Technology
technology that linearizes random writes to fix the random write problem of Flash



Endurance and Enterprise Database Workloads

- Parameters
  - Let I/O Block Size from the OS = OS_B (=64K)
  - No of I/Os per Day = N_{IO} (=50 million)
    Size of Flash chip = S (=2GB).
  - No of FLASH Chips = N_{chips} (=512).
  - Let SSD Block Size = SSD_B (=128K)
  - Total FLASH Memory = S*N (= 1TB)
  - Max possible writes per block = W_{max} (= 100K)
  - Total physical blocks in FLASH system = N_{Blk} = (N_{chips}* S)/ SSD_B = (512*2048*1024)/128 = 512*16K = 8192K blocks
- Assuming cyclic wear-leveling
  - Max possible writes for the system = N_{Blk} * W_{max} = 8192K*100K = 819200M writes
  - No of days SSD can last = $819200*10^6/50*10^6$ = 81920/5 = 16384 days = 45 years!
- If we assume striping of 128K across 8 chips for faster reads, then Write Endurance = 45/8 ~ 6 years

Current Industry Trends…

- Increasing Performance (Read/Write)
  - Array of FLASH memories to utilize the parallel bandwidth and reduce the latencies
  - A cache (DDR RAM) in front of the array to further minimize the latencies
  - Optimum Stripe Sizes (transfer rate is important)
- Increasing Write Performance
  - Erase as a parallel background operation
- Increasing Endurance
  - A cache (DDR RAM) in front of the array to buffer writes (write-back cahe)
  - Remapping for writes
  - Wear-leveling algorithms
  - Disks contain more space than advertised (of the order 20%)