Input/Output Systems

Prof. James L. Frankel
Harvard University

Version of 4:05 PM 27-Sep-2022

I/O Overview

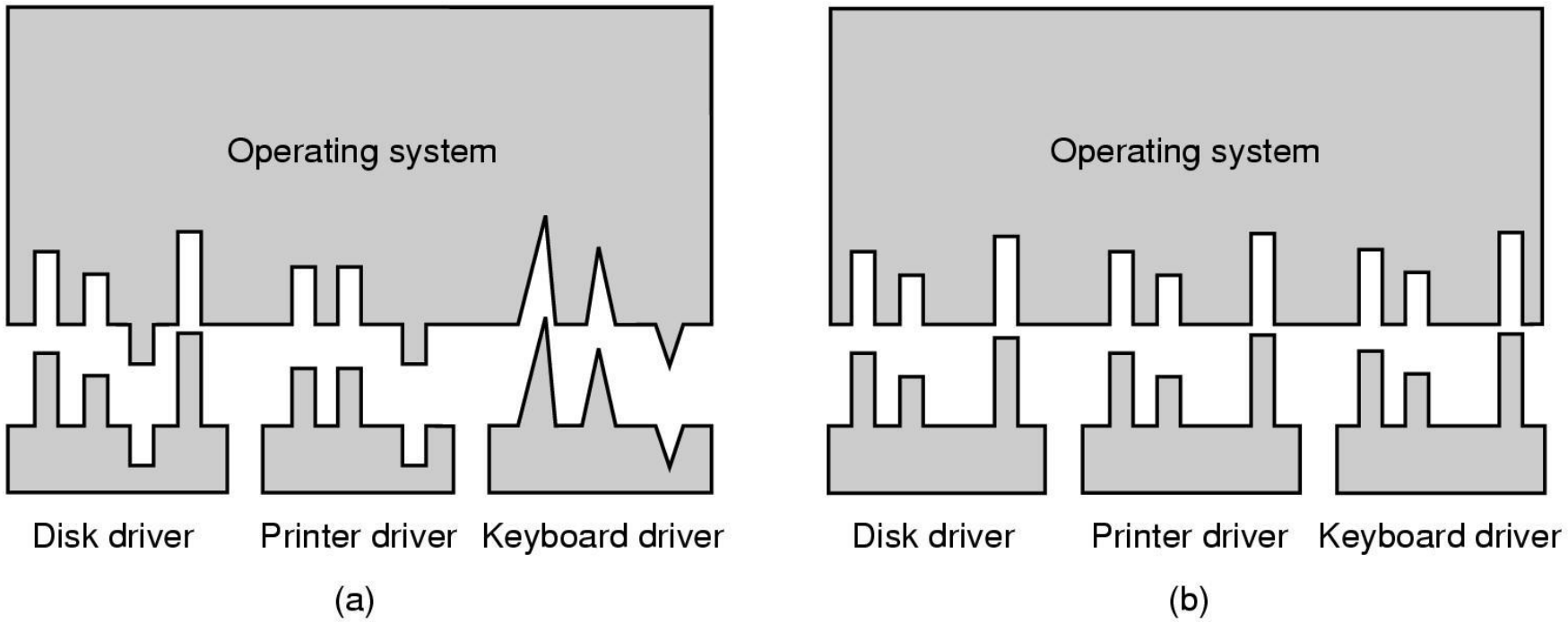
- Different kinds of devices
  - Mass storage
  - Keyboard
  - Pointing devices (e.g. mouse, touchpad, trackball, etc.)
  - Displays
  - Networks
    - Serial
    - USB
    - Ethernet
    - WiFi
    - Etc.

Components of an I/O Device

- Mechanical
  - Is the device itself (e.g. disk platters, arms, heads, servos)
- Electronics
  - Is the **device controller**

- Software in the operating system
  - Is the **device driver**

Device-Independent I/O Software



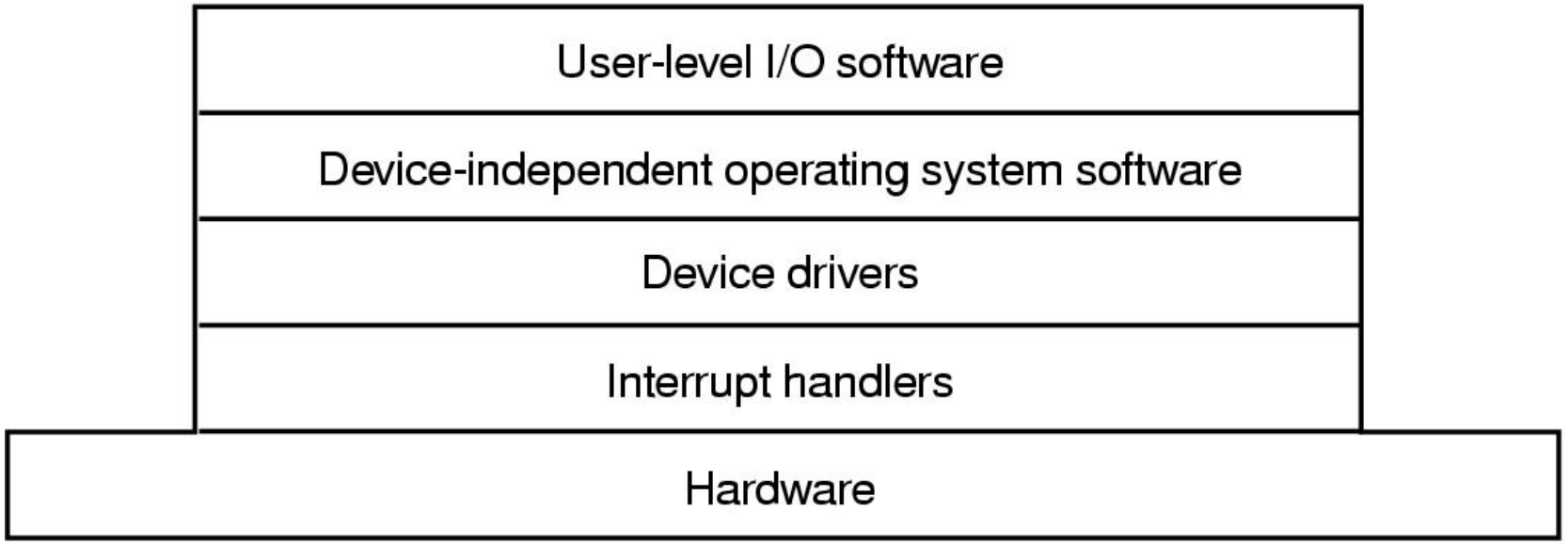
(a) Without a standard driver interface

(b) With a standard driver interface

Organization of Devices

- Are all devices accessed in a uniform way?
- Are all device drivers accessed by the OS in a uniform way?
- Do all devices appear to be part of a uniform file system?

I/O Software Layers



Layers of the I/O Software System

File Systems

- Naming
  - Includes an explicit different drive designator (e.g. C:, etc.)
  - Case sensitive
    - Does case matter for accessing a file?
    - Is case stored in the name?
  - If hierarchical, level separator character (e.g. /, \, etc.)
  - Is the OS aware of the file name extension (e.g. .txt, .bin, .exe, etc.)?
    - Does the extension affect how a file is opened?
  - Length of file name and file name extension
- Structure
  - Flat
  - Tree
  - Mountable file systems

File Contents

- Byte-structured
  - Each file appears as a stream of bytes
- Record/Block-structured
  - Each file appears to be a list of records
  - The records in a file may be of fixed size or variable size

File Types

- Types of files (in a Unix-like system)
  - Regular
  - Directories – hierarchical
  - Character special files
  - Block special files
  - Symbolic links
  - Named pipes (mkfifo)
- Other types of files
  - Binary vs. Text
  - Byte Stream vs. Block-based vs. Object-based
  - Indexed (Key-based)

File Access

- Sequential
  - Each access continues to next sequential unit
- Random
  - Seek
- Memory-mapped

File Links

- Symbolic Links
  - Maps one textual file name to another
- Hard Links
  - Has every file name point directly to the contents of a file
  - More than one file name can point to the same file contents

Operations on Files

- Open (open (low-level, non-buffered, in POSIX), fopen (stream-based, higher-level, buffered, in C library))
- Close (close, fclose)
- Create (creat, mkdir)
- Delete (unlink, rmdir)
- Create a hard link (link)
- Create a symbolic link (symlink)
- Rename
- Read (read, fgetc, getc (macro), fgets, ungetc, getchar (stdin))
- Write (write, fputc, putc (macro), fputs, putchar (stdout), puts (stdout with newline))
  - Does a write in the middle of a file overwrite?
  - Does a write in the middle of a file truncate the old length?
  - Does a write in the middle of a file insert?
- Flush (fflush)
  - Forces buffered output to be actually written
- Check for end-of-file (feof)
- Seek (seek, fseek, ftell, fgetpos, fsetpos, rewind)
- Manipulate attributes (chmod, chgrp, fcntl, fstat)

File Attributes

- Size
- File type
  - Regular vs. directory vs. specific by file name extension
- Device ID
  - Major and minor
- User ID of owner
- Group ID of file
- Protection by owner, by group (not owner), by others (not in group) – a.k.a. file mode
  - Read
  - Write
  - Execute
- Protection via access control lists (ACLs)
  - For granted permissions only
  - For granted and denied permissions
- Date/Time Stamps
  - Change time of the file data
  - Creation of file
  - Last modification
  - Last access
- Link count
- Hidden
- System
- Compressed
- Encrypted
- Indexed
- Archive (modified since last backup)
- Append-only
- Default application that opens the file
- Arbitrary attributes may be added by applications
- Icon to be displayed for this file

Unix Demonstration of File Attributes

I-node Structure

- A directory is a file that contains a list of file names and associated I-node numbers
- An I-node is a system structure that contains:
  - Attributes
  - Fifteen Pointers (12 direct pointers & 3 differing degrees of indirect pointers)
    - Up to twelve **direct pointers** to the first twelve blocks on disk that contain the file contents (first level)
    - A **singly indirect pointer** to a block on disk that contains additional block pointers, if necessary (second level)
    - A **doubly indirect pointer** to a block on disk that contains additional singly indirect block pointers, if necessary (third level)
    - A **triply indirect pointer** to a block on disk that contains additional doubly indirect block pointers, if necessary (fourth level)

Log-structured/Journaled File System

- The file system stores updated versions (or modifications) to files
- Requests for the current state of a file is usually served from a file cache
- The log allows the state of the system to be unrolled to a prior time with previous versions still available
- The log will eventually wrap around and early versions will need to be deleted to make space for new versions
- Allows large files to be stored as incremental changes
  - On a regular basis, the entire files are stored

File System Consistency

- fsck, chkdsk
- Write-through vs. write-back disk cache
  - A write-through cache writes the disk media on all cache updates
    - Advantage: Better consistency of the data on disk
  - A write-back cache delays the writes to disk media until necessary
    - Advantage: Better performance because there are fewer writes
- Removable media
  - Must ensure that all writes have been committed to the disk media before the media is removed from the system – this is why an unmount (umount) or eject system call exists

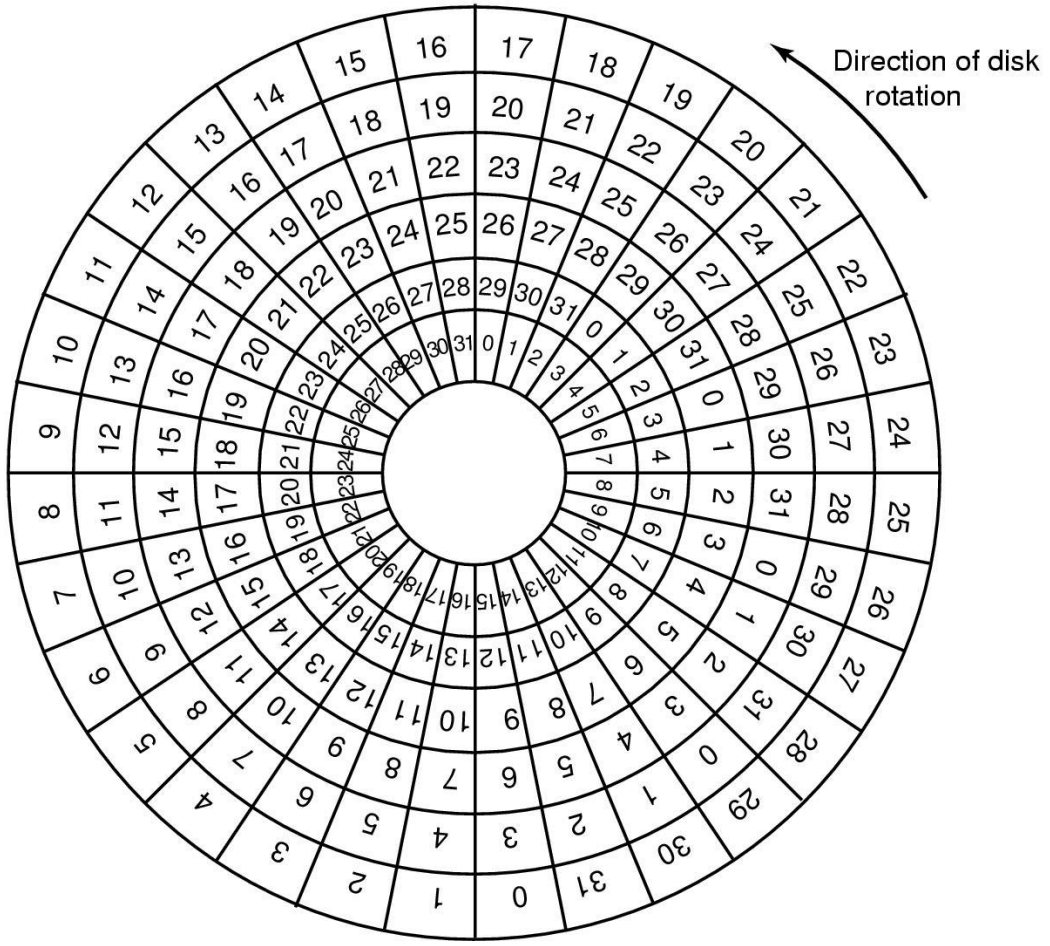
Increasing Disk Performance

- Block read ahead
- Reduce disk arm motion
- Overlap I/O with computation

Disk Formatting (1)

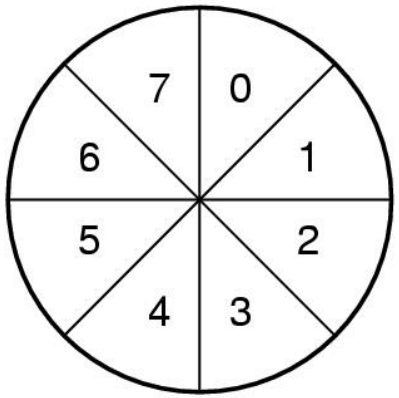
A disk sector

Disk Formatting (2)

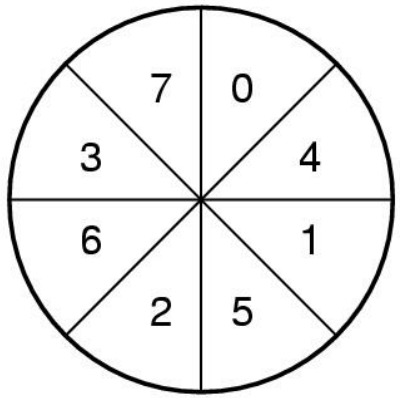


An illustration of cylinder skew

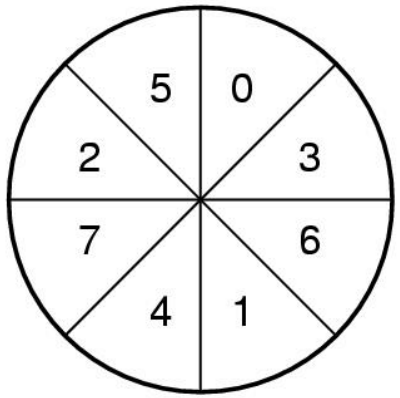
Disk Formatting (3)



(a)



(b)



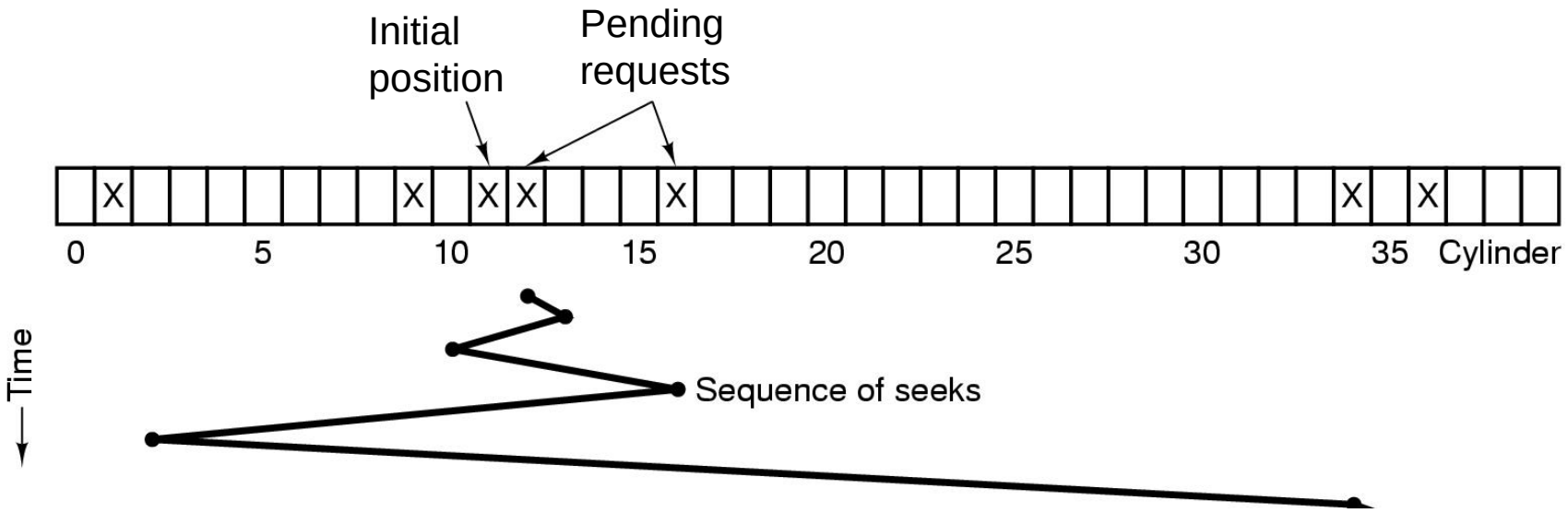
(c)

- No interleaving
- Single interleaving
- Double interleaving

Disk Arm Scheduling Algorithms (1)

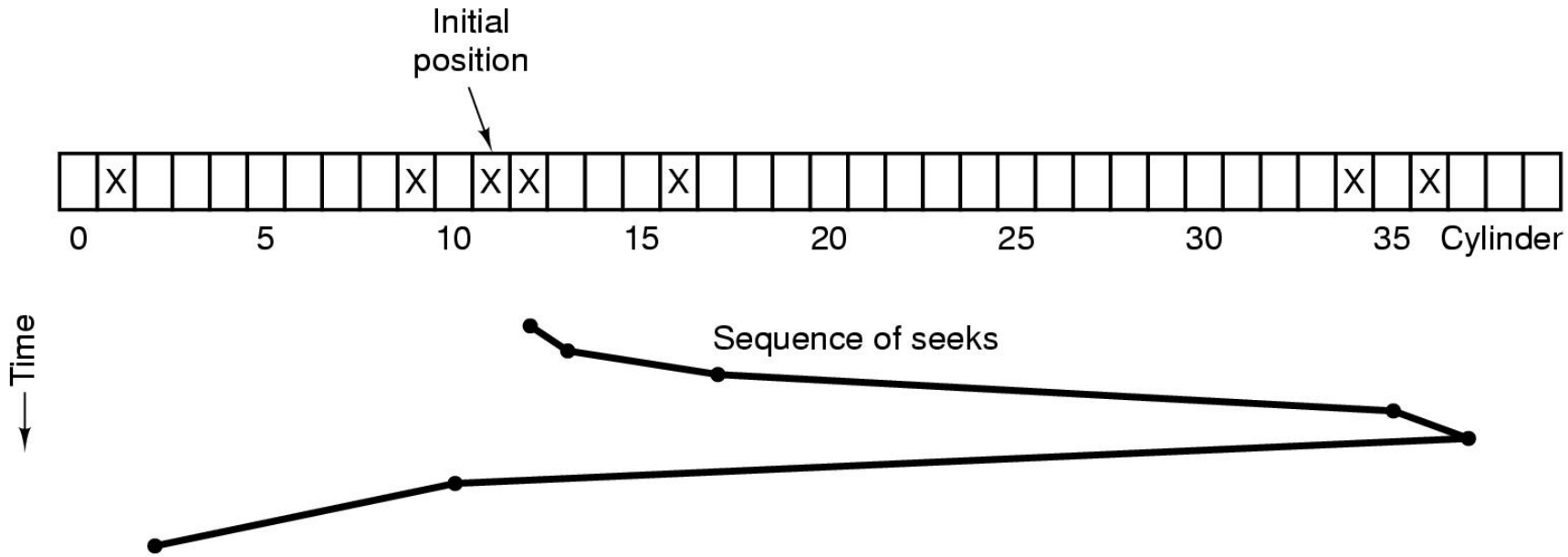
- Time required to read or write a disk block determined by 3 factors
  1. Seek time
  2. Rotational delay
  3. Actual transfer time
- Seek time dominates
- Error checking is done by controllers

Disk Arm Scheduling Algorithms (2)



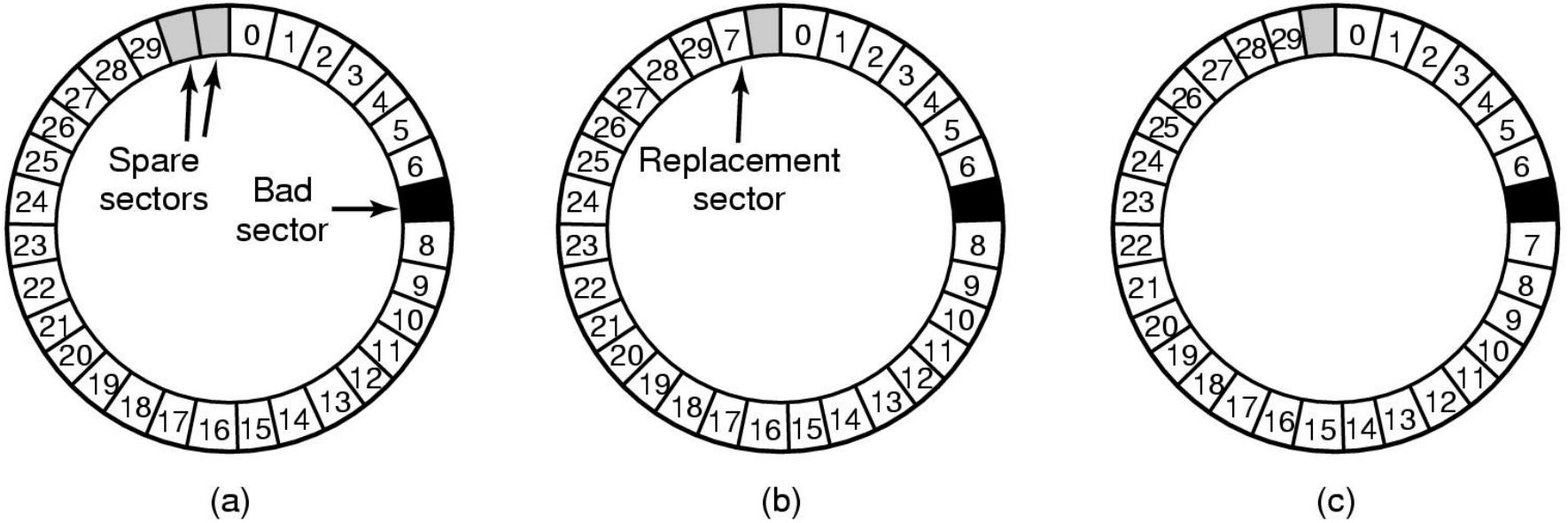
Shortest Seek First (SSF) disk scheduling algorithm

Disk Arm Scheduling Algorithms (3)



The elevator algorithm for scheduling disk requests

Error Handling



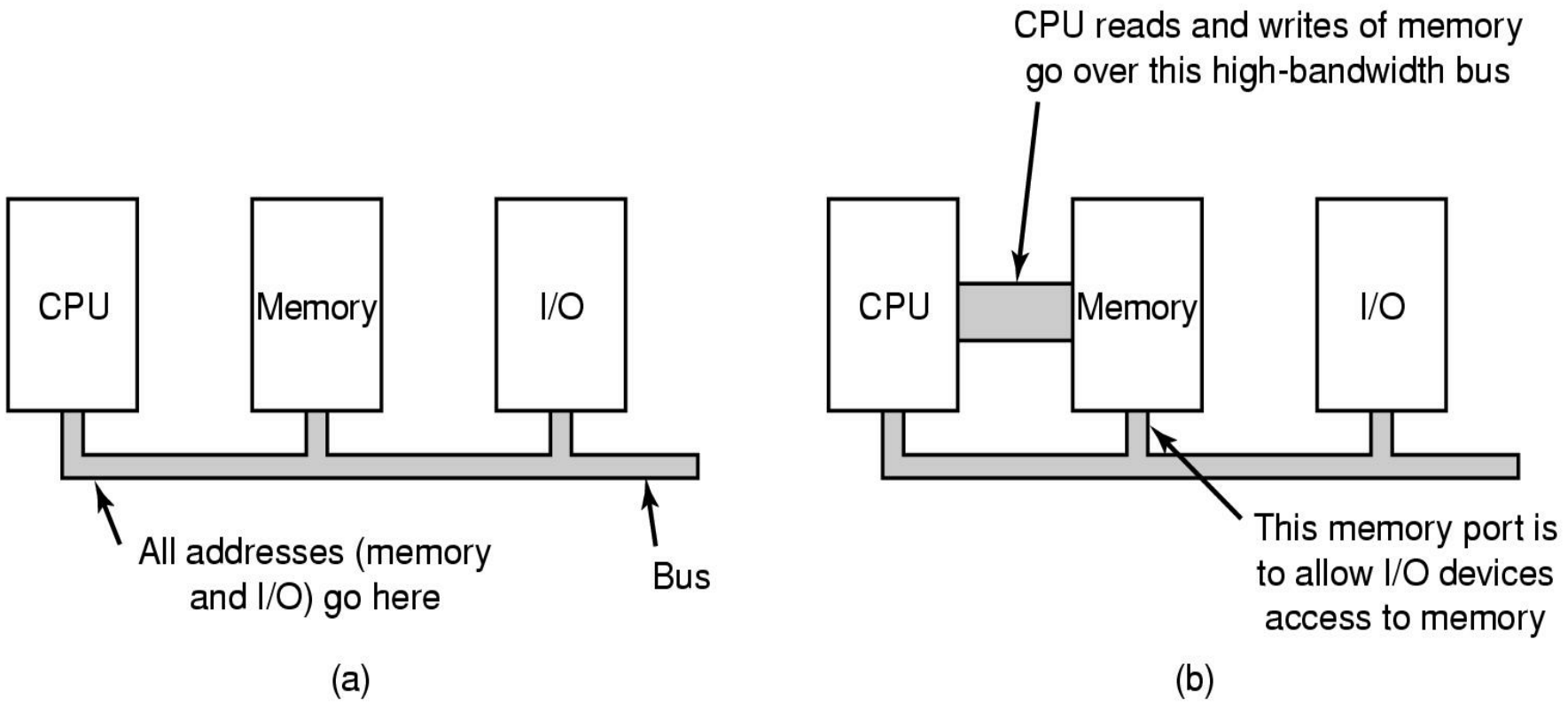
- A disk track with a bad sector
- Substituting a spare for the bad sector
- Shifting all the sectors to bypass the bad one

Accessing I/O Devices at a Low-Level

- Non-memory-mapped
  - I/O ports accessed using special instructions
- Memory-mapped
  - I/O ports in special address space
  - I/O ports in usual memory address space

- Direct Memory Access (DMA)
- Interrupts

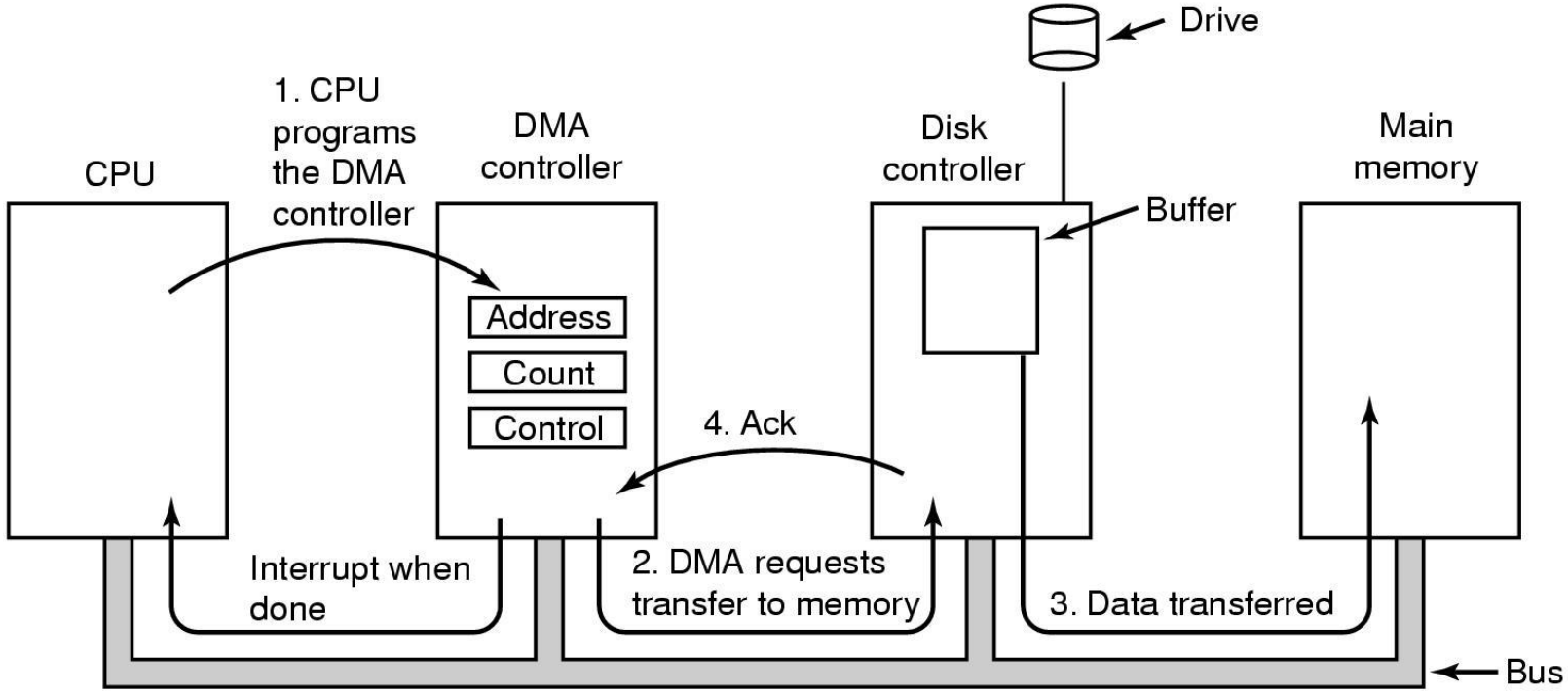
Busses Among CPU, Memory, and I/O



(a) A single-bus architecture

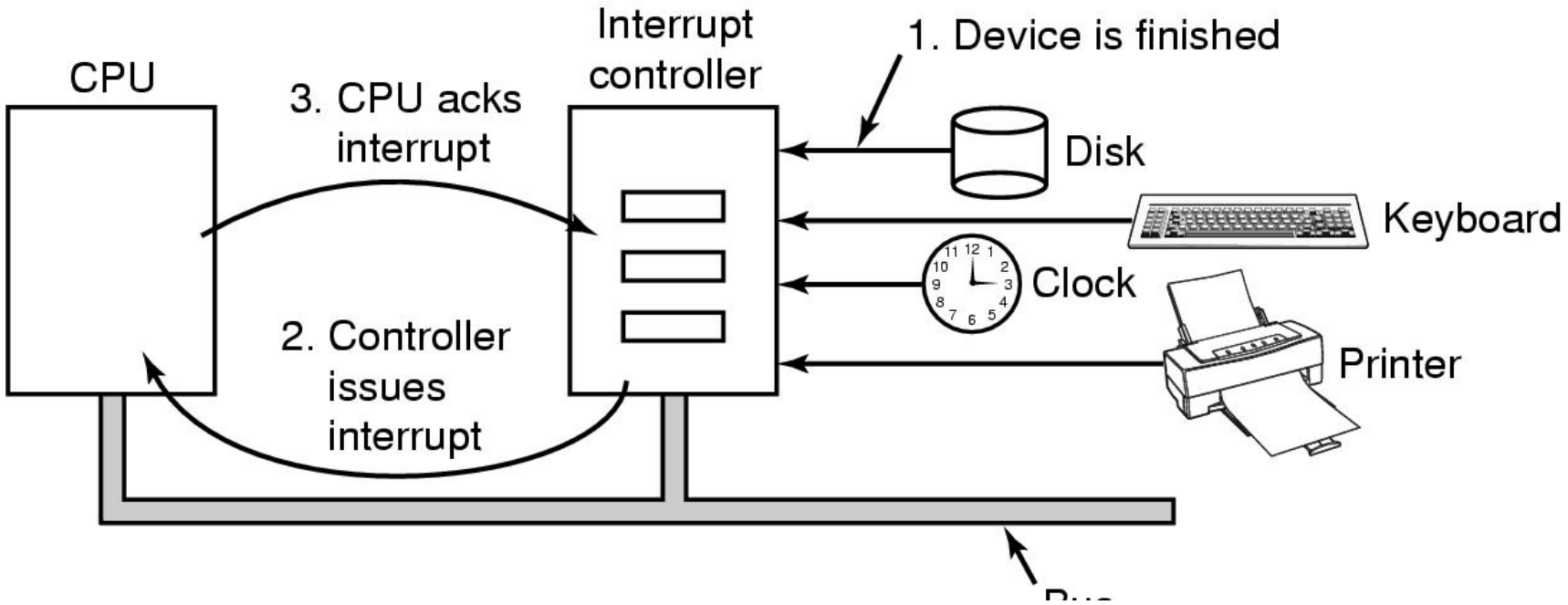
(b) A dual-bus memory architecture

Direct Memory Access (DMA)



Operation of a DMA transfer

Interrupts Illustrated



Interrupts

- How to find the Interrupt Service Routine (ISR) for any particular interrupt
  - Interrupt Vector Table
    - A list of pointers to ISRs for each class of interrupt
- The ISR handles the interrupt and then returns to the code that was running when it was interrupted

- Precise vs. Imprecise Interrupts
- Based on the definition in Wikipedia: An interrupt that leaves the machine in a well-defined state is called a *precise interrupt*. Such an interrupt has four properties
  - The Program Counter (PC) is saved in a known place
  - All instructions before the one pointed to by the PC have fully executed
  - No instruction beyond the one pointed to by the PC has been executed (that is not a prohibition on executing all or part of the instruction beyond that pointed to by the PC, it is just that any changes made to registers or memory must be undone before the interrupt occurs)
  - The execution state of the instruction pointed to by the PC is known
- An interrupt that does not meet these requirements is called an *imprecise interrupt*.

Performing I/O

- Programmed I/O
  - Instructions are executed to transfer data from a device to memory (input) and from memory to a device (output)
  - Program must execute instructions to check for I/O ready/completion
    - Would loop waiting
- Interrupt-driven I/O
  - An interrupt is signaled when I/O is read/has completed
    - The interrupt might wake up a sleeping process(es) waiting for that I/O event
- I/O using DMA
  - The I/O device's controller is able to transfer data to/from memory with requiring the processor's participation
  - An interrupt (or a programmed check) is still required to determine when the I/O operation has completed

RAID (*R*edundant *A*rray of *I*nexpensive *D*isks)

- Utilizes multiple disks to produce a device that can increase disk performance and/or reliability

Reliable Data Communication (1 of 3)

- Determine if data has arrived correctly – with some probability
- Parity
  - Simple technique
  - Add one additional bit – the parity bit – to some unit of bits (i.e., per byte, per word, etc.)
  - That parity bit will be transmitted with a known state determined by the other bits in the unit
    - Even parity – the number of one bits in the unit including the parity bit are even
    - Odd parity – the number of one bits in the unit including the parity bit are odd
  - Will detect any single bit error
  - Does not work well to detect certain errors (e.g. stuck at zero/one)

Reliable Data Communication (2 of 3)

- Probability of detecting an error can be increased
  - Additional computation is required
  - More bits need to be transmitted
    - Longer transmission time
- For certain devices, particular errors may be prevalent
  - Disk drives may have a defect on the surface of the disk
  - Interference may affect communication
  - Cosmic rays may affect the state of a bit in a processor
- CRC (Cyclic Redundancy Check)

Reliable Data Communication (3 of 3)

- Codes can be developed to do more than just detecting an error
- These so-called Hamming codes can also ***correct some errors***

Hamming Codes

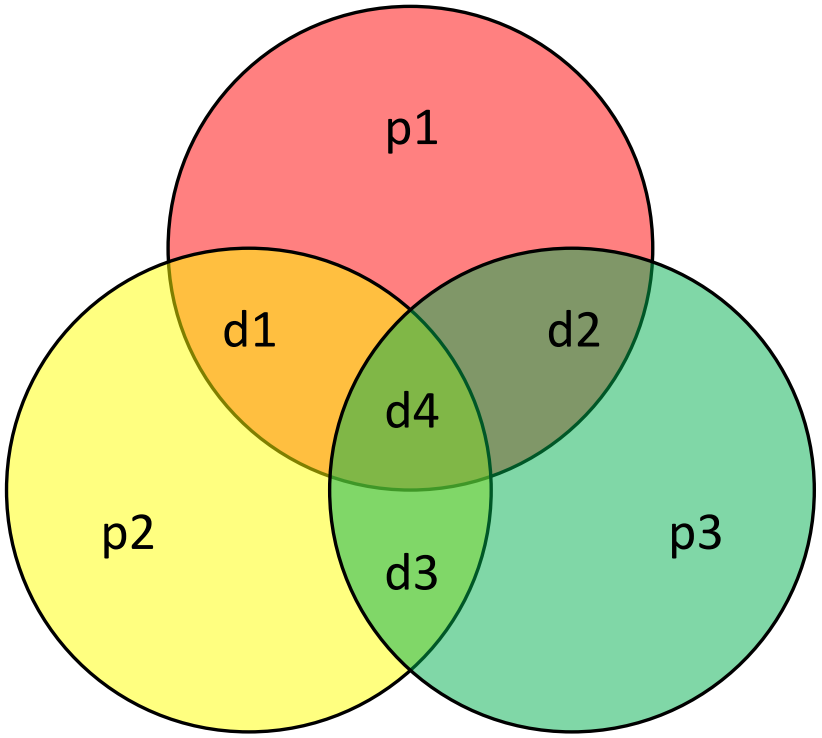
- Let’s look at Hamming(7, 4) as an example
  - 4 bits of data
  - Encoded into 7 bits by adding 3 parity bits
    - We will use even parity
- This code will detect and correct any single-bit error
- This code will detect but not correct any double-bit error

- For our purposes, we will number the bits from 1 to 7

Hamming(7, 4) Bit Designations

Bit #	1	2	3	4	5	6	7
Bit Name	p1	p2	d1	p3	d2	d3	d4
p1	Yes	No	Yes	No	Yes	No	Yes
p2	No	Yes	Yes	No	No	Yes	Yes
p3	No	No	No	Yes	Yes	Yes	Yes

Hamming(7, 4) Venn Diagram



Hamming(7, 4) Example (1 of 4)

- We want to transmit d1, d2, d3, d4 as 0110
- We generate p1 from d1, d2, and d4
  - 010 -> even parity is 1
- We generate p2 from d1, d3, and d4
  - 010 -> even parity is 1
- We generate p3 from d2, d3, and d4
  - 110 -> even parity is 0
- We would transmit p1, p2, d1, p3, d2, d3, d4 as 1100110

Hamming(7, 4) Example (2 of 4)

- We transmitted p1, p2, d1, p3, d2, d3, d4 as 1100110
- If we receive p1, p2, d1, p3, d2, d3, d4 as 1100010, then
  - We recompute p1, p2, and p3 including the respective parity bit
  - If the received data is correct, the recomputed parity bits should all be zero
- p1 from p1, d1, d2, and d4
  - 1000 -> even parity is 1
- p2 from p2, d1, d3, and d4
  - 1010 -> even parity is 0
- p3 from p3, d2, d3, and d4
  - 0010 -> even parity is 1
- Because all three recomputed parity bits are not zero, we know that an error has occurred
- In our example, p1 and p3 indicate errors

Hamming(7, 4) Example (3 of 4)

- Each parity bit indicates that a single-bit error occurred within their set
- By looking at the intersection of the one to three parity bits that indicate an error, we can determine the source of the error
- For example, if bits p1 and p3 indicate an error, then bit d2 is incorrect and needs to be inverted
  - Thus, the data bits d1, d2, d3, d4 received as 0010 would be corrected by inverting bit d2 and thus yielding 0110

Hamming(7, 4) Error Source

- If bit p1 indicates an error, then bit p1 (bit 1) is incorrect
- If bit p2 indicates an error, then bit p2 (bit 2) is incorrect
- If bit p3 indicates an error, then bit p3 (bit 4) is incorrect
- If bits p1 and p2 indicate an error, then bit d1 (bit 3) is incorrect
- If bits p1 and p3 indicate an error, then bit d2 (bit 5) is incorrect
- If bits p2 and p3 indicate an error, then bit d3 (bit 6) is incorrect
- If bits p1, p2, and p3 indicate an error, then bit d4 (bit 7) is incorrect

Hamming(7, 4) Error Bit Designation

- If bit p1 indicates an error, then bit p1 (bit 1) is incorrect
- If bit p2 indicates an error, then bit p2 (bit 2) is incorrect
- If bit p3 indicates an error, then bit p3 (bit 4) is incorrect
- If bits p1 and p2 indicate an error, then bit d1 (bit 3) is incorrect
- If bits p1 and p3 indicate an error, then bit d2 (bit 5) is incorrect
- If bits p2 and p3 indicate an error, then bit d3 (bit 6) is incorrect
- If bits p1, p2, and p3 indicate an error, then bit d4 (bit 7) is incorrect

- Because of our clever positioning of the parity bits, if we take the value of p3 (MSB), p2, p1 (LSB) as an integer, it will be the number of the incorrect bit!

Hamming(7, 4) Example (4 of 4)

- Looking at the received and recomputed parity bits p3, p2, p1, they are 101
- As an integer, that is the value 5
- Bit 5 is d2!
- Thus, we invert the d2 bit for the correct data word

Returning to RAID

- We can utilize Hamming Codes to implement RAID

- Imagine that we store data on disk drives as 4 bits words
- We could make our write or read speed four times faster by storing a single bit onto each of 4 disk drives
- If we compute the Hamming(7, 4) code on every read or write and utilize 7 disk drives instead of 4, we can detect and correct for any one drive failure

- Once, we detect a drive failure, we can recreate the contents of that drive
  - We replace the broken disk drive with a working drive
  - We read from every word and write back that word
  - When we complete reading and writing back every word, the contents of the broken drive have been recreated

Network I/O

- Data sent over a network needs to be meaningful at the receiving side
- First Issue
  - Is data sent in text or binary?

Network I/O: Text

- Character set
  - ASCII
  - EBCDIC (IBM's standard)
  - Unicode
    - International character set uses more than a byte per character
    - UTF-8
      - ASCII is unchanged
      - Extended Unicode characters are up to four bytes in length
    - UTF-16
      - Characters are either two or four bytes in length

Network I/O: Formatted Text

- HTML
- YAML
- XML
- JSON (JavaScript Object Notation)
  - Usually used between a browser and the server to which it is communicating

Network I/O: Binary

- Format of numbers (see Numeric Encoding slides)
  - Are floating-point numbers represented the same way?
    - IEEE 754 has made this much more likely
  - In which order are bytes sent?
  - What is the byte ordering of each computer?
    - **Big endian** (also known of as **network byte order**)
      - The Most-Significant Byte of a word is stored in the lowest-numbered memory location
      - Includes Motorola, IBM, and IP protocols
    - **Little endian**
      - The Least-Significant Byte of a word is stored in the lowest-numbered memory location
      - Includes Intel architectures, Z80, MOS Technology 6502, DEC VAX