CS 370: OPERATING SYSTEMS
[INTRODUCTION]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2026

** Lecture slides created by: SHRIDEEP PALLICKARA

Topics covered in this lecture

- Brief course Overview
- Introduction and reminders about computers

Course Overview

- All course materials will be accessible via the public-facing webpage

  => https://courses.cs.colostate.edu/cs370/ (will be populated this week)

  NOTE: temporarily for this week, see instead https://www.cs.colostate.edu/~cs370/Spring24/

  - Schedule (Lecture slide sets for each lecture)
  - Assignments
  - Syllabus
  - Grading policy
- Grades will be posted on **Canvas**; assignment submissions will be via Canvas
- The course website, MS Teams Channel, and Canvas will all be updated this week, expectedly by Thursday January 22 2026.

Pedagogical Objectives

Upon successful completion of this course students will be able to:

1. Explain basic operating system terminology
2. Explain processes and thread management
3. Distill core concepts in scheduling algorithms and develop tools to assess their performance
4. Synthesize diverse concepts in memory management.
5. Contrast mechanisms for interprocess communications
6. Distill and build upon core concepts in process and task synchronization
7. Design resource management schemes that mitigate deadlocks
8. Explain file systems and storage architecture
9. Contrast virtualization and containers alongside identifying when one approach outperforms the other

Topics Covered in CS370

- Processes and Threads
- Process Synchronization (plus Atomic Transactions)
- CPU Scheduling
- Deadlocks
- UNIX I/O
- Memory Management
- File System interface and management. Unix file system. NTFS.
- Storage Management including SSDs and Flash Memory
- Virtualization and Containers, and modern safety mechanisms in OS

Course Textbook

Operating Systems Concepts, 9th/10th edition

Avi Silberschatz, Peter Galvin, and Greg Gagne Publisher - John Wiley & Sons, Inc.

ISBN-13: 978-1118063330.

Grading Policy

Course Element	Weight
Assignments	45% [5, 5, 5, 10, 10, 10]
Quizzes	10%
Mid Term	20%
Final Exam	25%

- Letter grades will be based on the following standard breakpoints:
  - >= 90 is an A, >= 88 is an A-, >=86 is a B+, >=80 is a B, >=78 is a B-, >=76 is a C+, >=70 is a C, >=60 is a D, and <60 is an F.
  - No cut higher than this, but *may* be cut lower (i.e., higher letter grade than displayed above)

A Word About Me

- **Louis-Noel Pouchet**, Associate Professor in Computer Science, joint appointment in Electrical and Computer Engineering
  - Joined CSU in 2016
  - Teaching semester in SP26: CS370, CS453, CS553.
  - Specialty: compilers, high-performance computing, hardware/software co-design, distributed systems,…
  - Working also with AMD on correctness verification for programs (deadlock detection, etc.)
- To reach me for any direct communication (not seen by TAs, just me): email directly pouchet@colostate.edu with subject line "[CS370] …your subject…"
- In case of emergency my cellphone is +1 614 859 5115
- For all communications related to this class, but which may be addressed by our TA team or I, we will use compsci_cs370@colostate.edu. This email is read by the entire teaching team.
- *Note CS370 in Spring 2026 is nearly exactly the class from Pr. Shrideep Pallicakara, I am only the instructor. He is carefully thanked for sharing all his material! ☺*

OPERATING SYSTEMS

A modern computer is a complex system

- Multiple processors
- Main memory and Disks
- Keyboard, Mouse and Displays
- Network interfaces
- I/O devices

Why do we need Operating Systems?

- If every programmer had to understand how *all* these components work?
  - Software development would be arduous

- Managing all components and using them optimally is a challenge

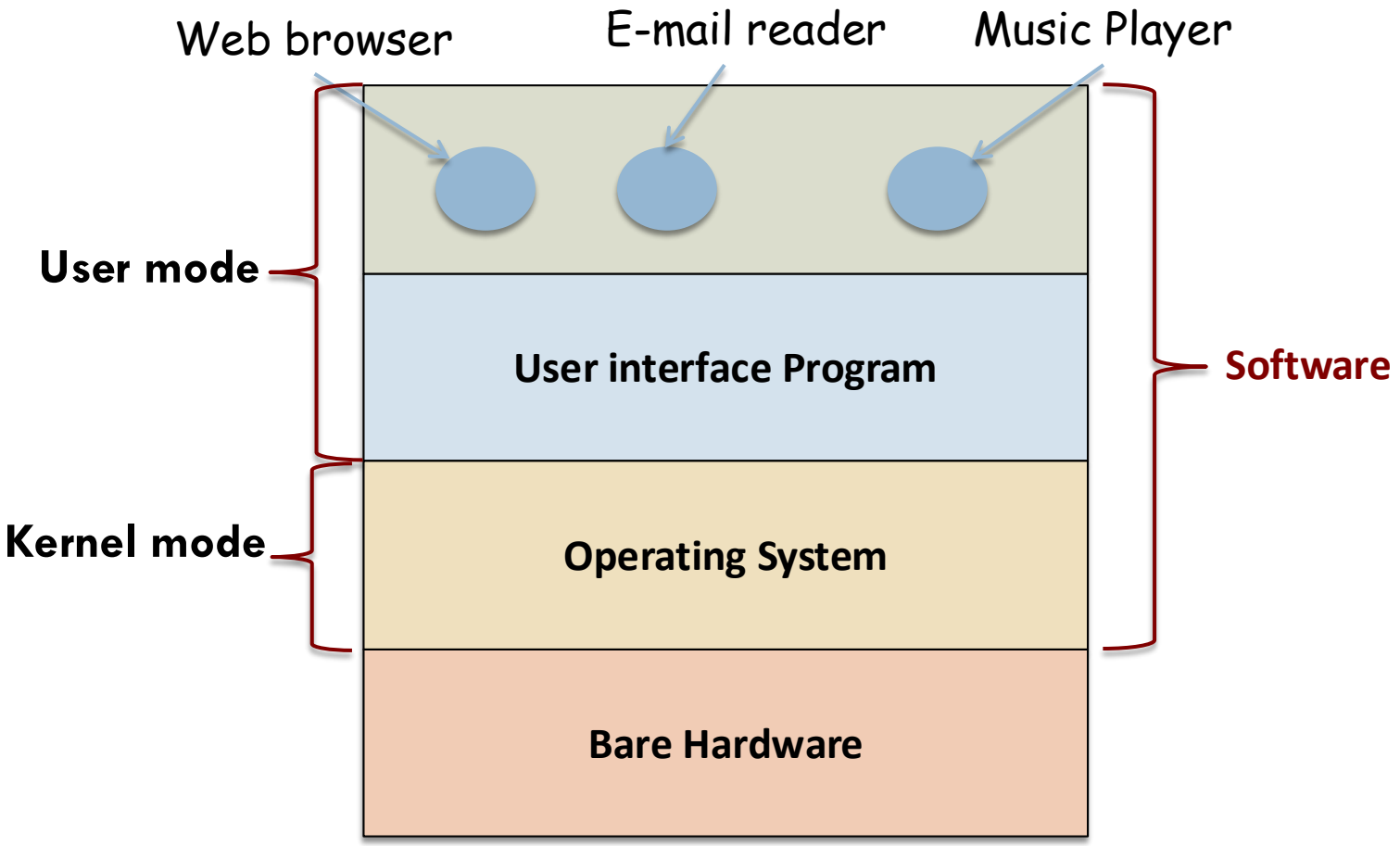
Computers are equipped with a layer of software

- Called the **Operating System**
- Functionality:
  - Provide user programs with a better, simpler, cleaner model of the computer
  - Manage resources efficiently

A common misconception about the OS

- Is it the program that users interact with?
  - Text based: Shell
  - Graphical User Interfaces (GUI) that have icons etc.
    - The look-and-feel if you will
- This is **not actually part** of the OS
  - But it does use the OS to get its work done

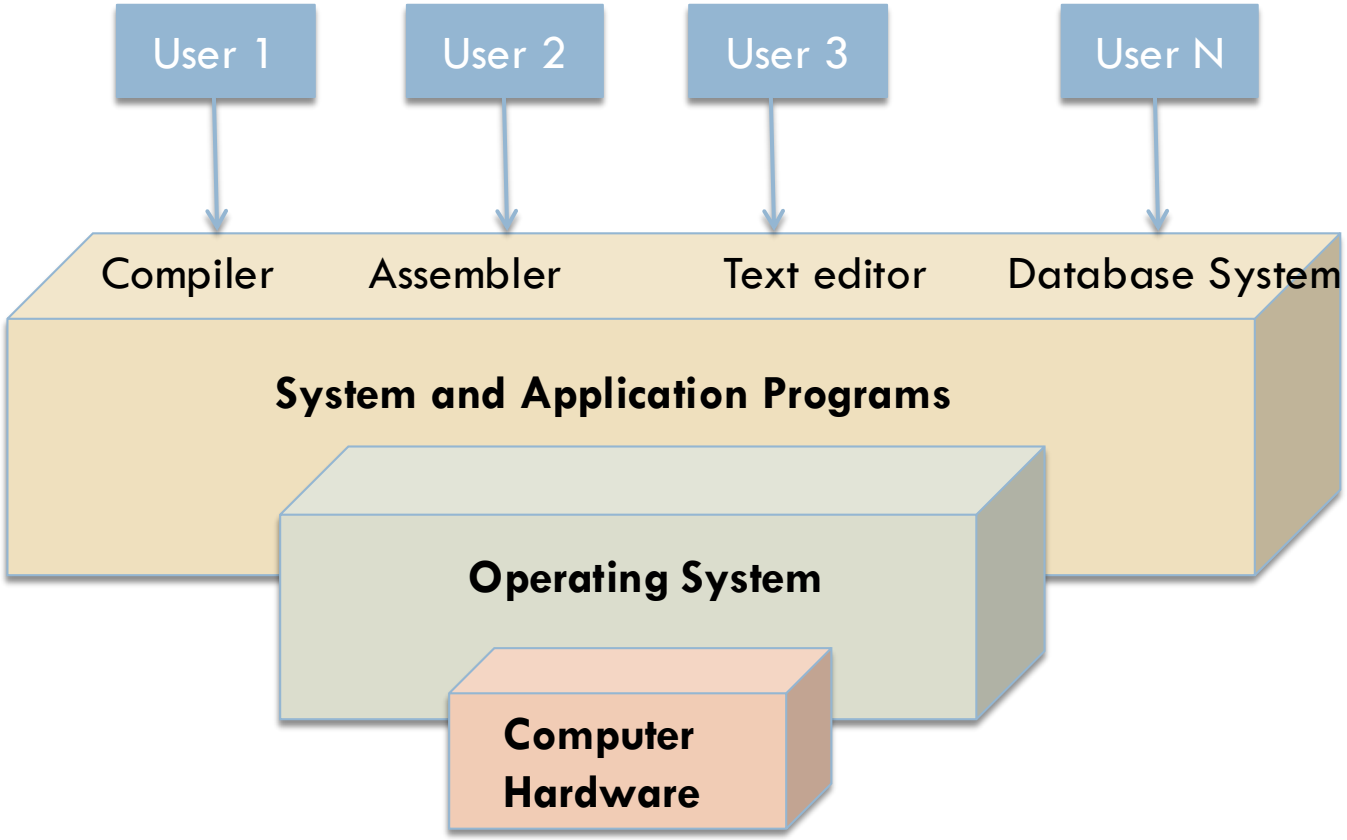
Where the operating system fits in



Where the operating system fits in

- The OS runs on bare hardware in **kernel mode**
  - *Complete access* to all hardware
  - Can execute *any* instruction that the machine is capable of executing
- Provides the base for all software
  - Rest of the software runs in **user-mode**
    - Only a **subset** of machine instructions is available

The OS controls hardware and coordinates its use among various programs



Kernel and user modes

- Everything running in kernel mode is part of the OS
- But some programs running outside it are part of it or at least closely associated with it

Operating systems tend to be huge, complex and long-lived

- Source code of an OS like Linux or Windows?
  - Order of > 5 million lines of code (for kernel)
    - 50 lines page, 1000 pages/volume = 100 volumes
- Application programs such as GUI, libraries and application software?
  - 10-20 times that

Why do operating systems live for a long time?

- Hard to write and folks are loath to throw it out
- Typically **evolve** over long periods of time
  - Windows 95/98/Me is one OS
  - Windows NT/2000/XP/Vista/7/8 is another
  - System V, Solaris, BSD derived from original UNIX
  - Linux is a fresh code base
    - Closely modeled on UNIX and highly compatible with it
  - Apple OS X based on XNU (X is not Unix) which is based on the Mach microkernel and BSD's POSIX API

An operating system performs two unrelated functions

- Providing application programmers a clean **abstract** set of resources
  - Instead of messy hardware ones
- **Managing** hardware resources

The OS as an extended machine

- The **architecture** of a computer includes
  - Instruction set, memory organization, I/O, and bus structure
- The architecture of most computers at the machine language level
  - Primitive and awkward to program especially for I/O

Lets look at an example of floppy disk I/O done using NEC PD765

- The PD765 has 16 commands
  - For reading and write data, moving the disk arm, formatting tracks, etc.
  - Specified by loading 1-9 bytes into the device register

- Most basic commands are for read and write
  - 13 parameters packed into 9 bytes
    - Address of disk block, number of sectors/track, inter-sector gap spacing etc.

But that's not the end of it …

- When the operation is completed
  - Controller returns 23 status and error fields packed into 7 bytes
- You must also check the status of the **motor**
  - If it is off? Turn it on before reading or writing
  - Don't leave the motor on for too long
    - Floppy disk will wear out
  - TRADEOFF: Long start-up delay *Vs* wearing out disk

And what about AI accelerators?

Instructions for 32-bit floating-point arithmetic

Two 32-bit FP arithmetic instructions are used to discuss the functionality of MMA. The two instructions that are used to perform a single precision matrix multiplication operation are: **xvf32ger** and **xvf32gerpp**.

The difference between the **ger** instruction and the **gerpp** instruction is as follows:

- The **gerpp** instruction *accumulates* the results in the accumulator register. This instruction requires the accumulator to already have a defined content.
- The **ger** instruction *overwrites* the results in the accumulator register. This instruction defines the content of an accumulator, similar to the **xxmtacc** and **xxsetaccz** instructions.

xvf32gerpp AT,XA,XB, where:

- **AT** refers to any of the eight accumulator registers (ACC0-ACC7).
- **XA** and **XB** refer to VSRs.

For the **xvf32gerpp AT,XA,XB** instruction, assume **AT=1**, **XA= 32**, and **XB=33**. The VSR 32 has four 32-bit single precision values and VSR 33 has four 32-bit single precision values. Each value in VSR 32 is multiplied with each value in VSR 33, generating a 4x4 array of 32-bit results (a total of 512 bits of output). The output is accumulated with the content of ACC1, as shown in Figure 2-2.

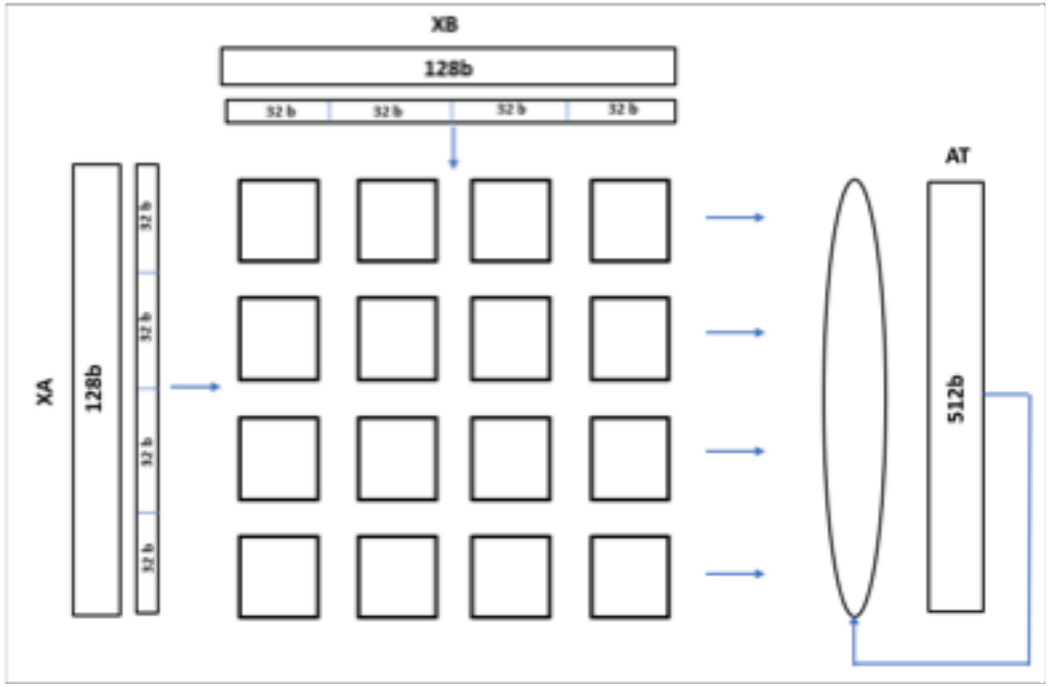


Figure 2-2 MMA xvf32gerpp instruction operation

Source: https://www.redbooks.ibm.com/redpapers/pdfs/redp5612.pdf Matrix-Multiply Assist Best Practices Guide

Of course the average programmer does not want to have any of this

- What they would like is a simple, high-level **abstraction** to deal with
- For a disk this would mean a collection of named **files**
  - Operations include open, read, write, close, etc.
  - BUT NOT
    - Whether the recording should use frequency modulation
    - The state of the motor

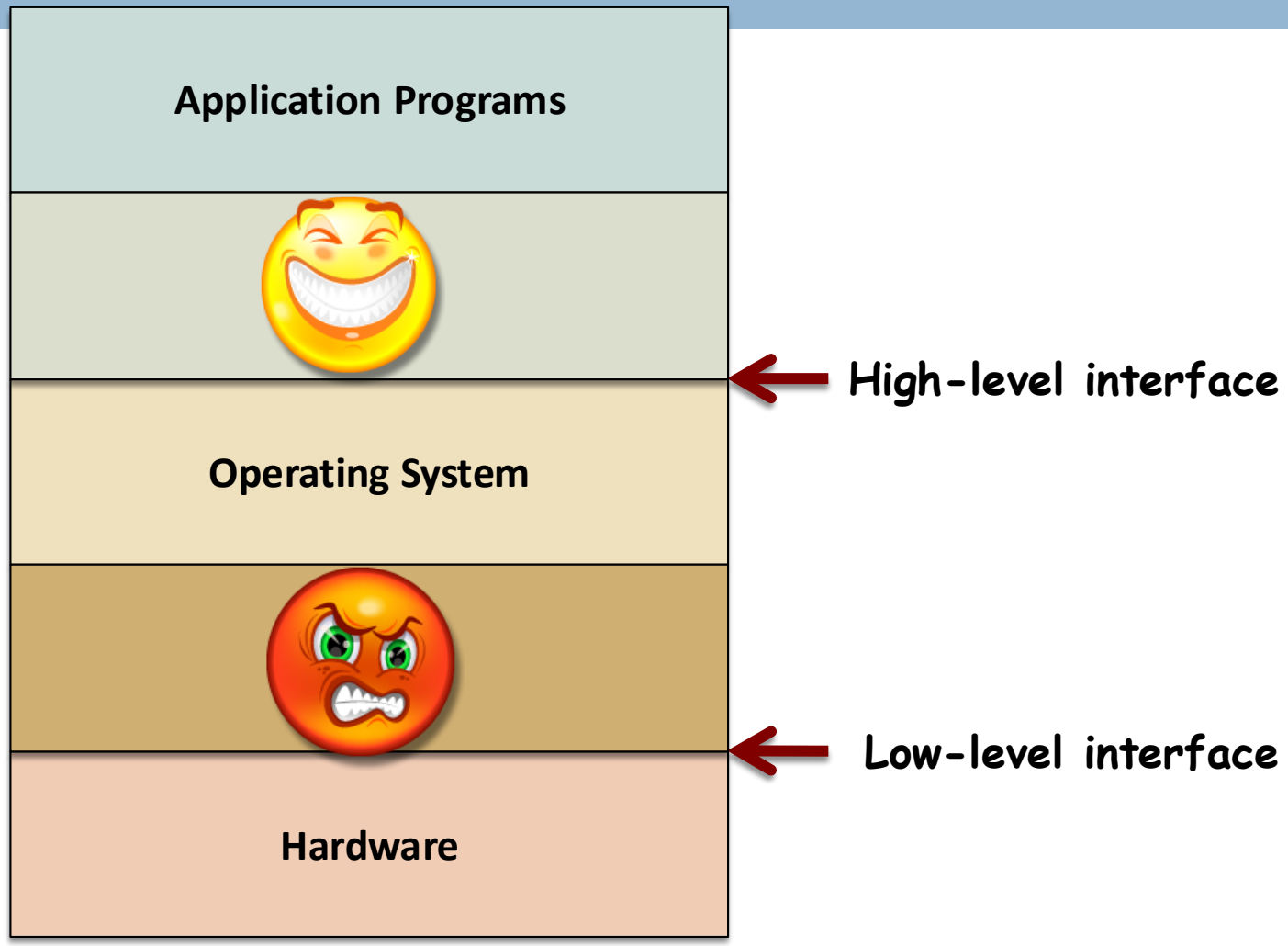
Why do processors, disks, etc. present difficult, awkward, idiosyncratic interfaces ?

- Backward compatibility with older hardware
- Desire to save money
- Sometimes hardware designers don't realize (or care) how much trouble they cause!

Why abstractions are important

- Abstraction is the key to managing **complexity**
- Good abstractions turn a nearly impossible task into two manageable ones
  1. Defining and implementing abstractions
  2. Using abstractions to solve problem
- Example
  - File

Operating systems turn ugly hardware into beautiful interfaces



Two views of the operating system

- Top-down view
  - Providing abstractions to the application programs
- Bottom-up view
  - Manage all pieces of a complex system

The operating system as a resource manager

- Provide **orderly** and **controlled** allocation of resources to programs competing for them
  - Processors, memories, and I/O devices

Operating System Roles: User View

- PC Users: Ease of use
- Mainframe: Maximize resource utilization
- Workstations: Compromise between usability and resource utilization.
- Handheld devices: Ease of use + performance per unit of battery life

The System view of the OS is that of a Resource Allocator

- An OS may receive numerous & conflicting requests for resources
  - Prevent errors and improper use
- Resources are scarce and expensive
- The OS allocates resources to specific programs and users
  - The allocation must be efficient and fair
  - Must increase overall system throughput

Defining Operating Systems

- Solves the problem of creating a usable computing system
  - Makes solving problems easier
- Control, allocate and mediate access to resources
- It is the one program that is running all the time: **kernel**

A (very) Brief History of Operating Systems

The first mechanical computer was designed by Charles Babbage (1792-1871)

- Spent most of his life and fortune trying to build the analytical engine
- Never got it working properly
  - Purely mechanical
  - Technology of the day could not produce wheels, cogs, gears to the required precision
- Did not have an operating system ;-)

Babbage realized he would need software for his analytical engine

- Hired Ada Lovelace as the worlds first programmer
  - Daughter of British poet Lord Byron
- The programming language Ada® is named after her

History… from CS453 on Compiler Construction

A bit of (modern) history…

- It all started with punch cards
- As early as 19th century
- Picture: IBM machine, 1936
- At start: storage, basic processing
- Programming was hard!
- A good quote (IBM Manual, 1925):

> *All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer.*

The First Generation (1945-55) Vacuum Tubes

- First fully functioning digital computer built at Iowa State University
  - Prof. John Atanasoff and grad student Clifford Berry
- All programming in absolute machine language
  - Also by wiring up electrical circuits
    - Connect 1000s of cables to plugboards to control machine's basic functions
  - Operating Systems were unheard of
- Straightforward numerical calculations
  - Produce tables of sines, cosines, logarithms

The Second Generation (1955-1965): Transistors and Batch Systems

- **Separation** between designers, builders, operators, programmers, and maintenance
- Machines were called **mainframes**
- Write a program on paper, then punch it on cards
  - Give card deck to operator and go drink coffee
  - Operator gives output to programmer

The Third Generation (1965-1980)
ICs and Multiprogramming

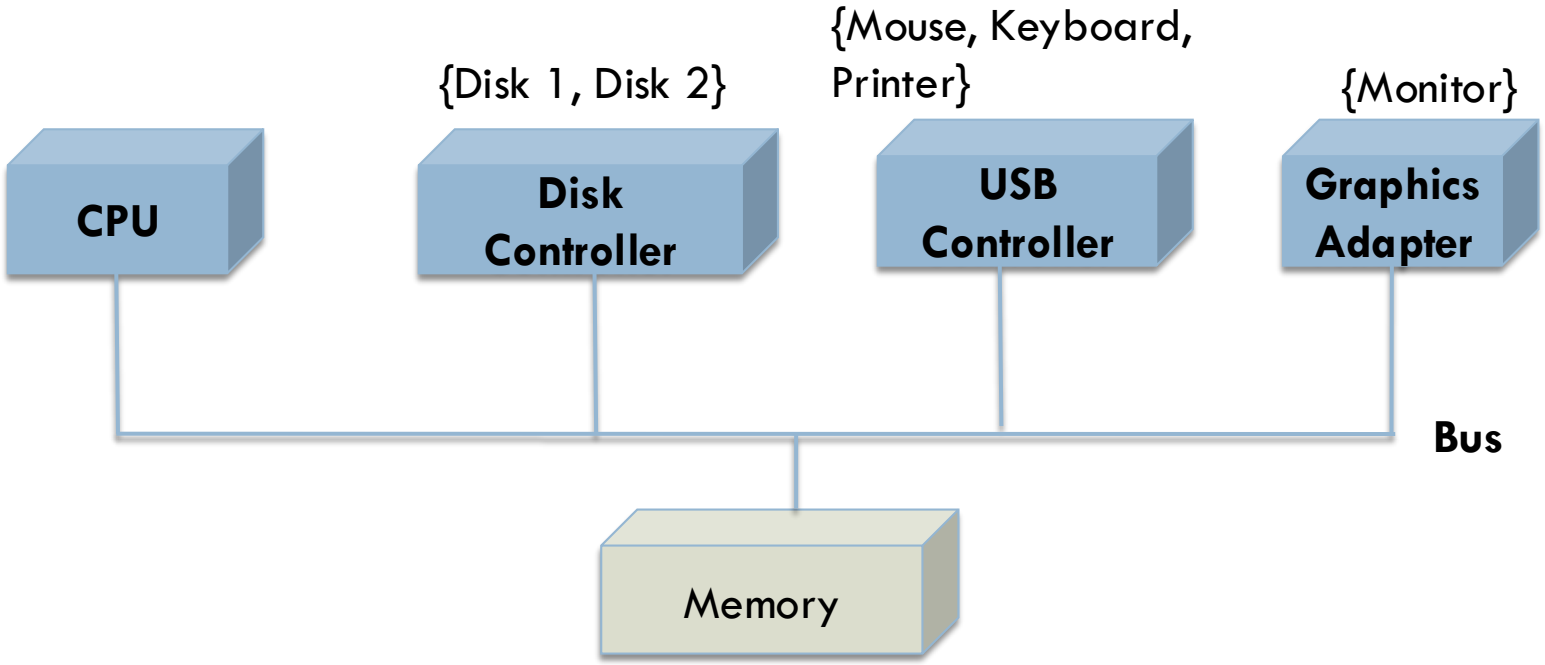
- Managing different product lines was expensive for manufacturers
  - Customers would start with a small machine, and then outgrow it
- IBM introduced the Systems/360
  - Series of **software-compatible** machines
  - All machines had the same instruction set
    - Programs written for one machine could run on all machines

The Fourth Generation (1980-Present) Personal Computers

- Large Scale Integration circuits (LSI)
  - Thousands of transistors on a square centimeter of silicon
- 1974: Intel came out with the 8080
  - General purpose 8-bit CPU
- Early 1980s IBM designed the IBM PC
  - Looked for an OS to run on the PC
  - Microsoft purchased Disk Operating System and went back to IBM with MS-DOS

COMPONENTS OF A COMPUTER

Components of a simple personal computer



Processors

- **Brain** of the computer
- Each CPU has a specific set of instructions that it can execute
  - Pentium cannot execute SPARC and vice versa

Rationale for registers inside the CPU

- Accessing memory to get instruction or data
  - **Often much longer** than executing the instruction
- Registers hold any data processed by the CPU:
  - Key variables
  - Temporary results

What the instruction set looks like

- Load a word from memory into register
  - And, from register into memory
- Combine two operands from register, memory, or both into a result
  - E.g. add two words and store result in a register or in memory

```
L6:
        vmovss      (%rax), %xmm1
        addq        $4, %rax
        vfmadd231ss  (%rdx), %xmm1, %xmm0
        addq        %rbx, %rdx
        cmpq        %rax, %rsi
        jne   L6
        vmovss      %xmm0, (%rdi,%r9,4)
```

Besides the registers to hold variable and temporary results there are special registers

- **Program Counter**
  - Contains the memory address of the program instructions
- **Stack pointer**
  - Points to the top of the current stack in memory, to help manage local memory
- Program Status Word
  - Stores condition code bits and other control code bits
  - Plays an important role in system calls and I/O

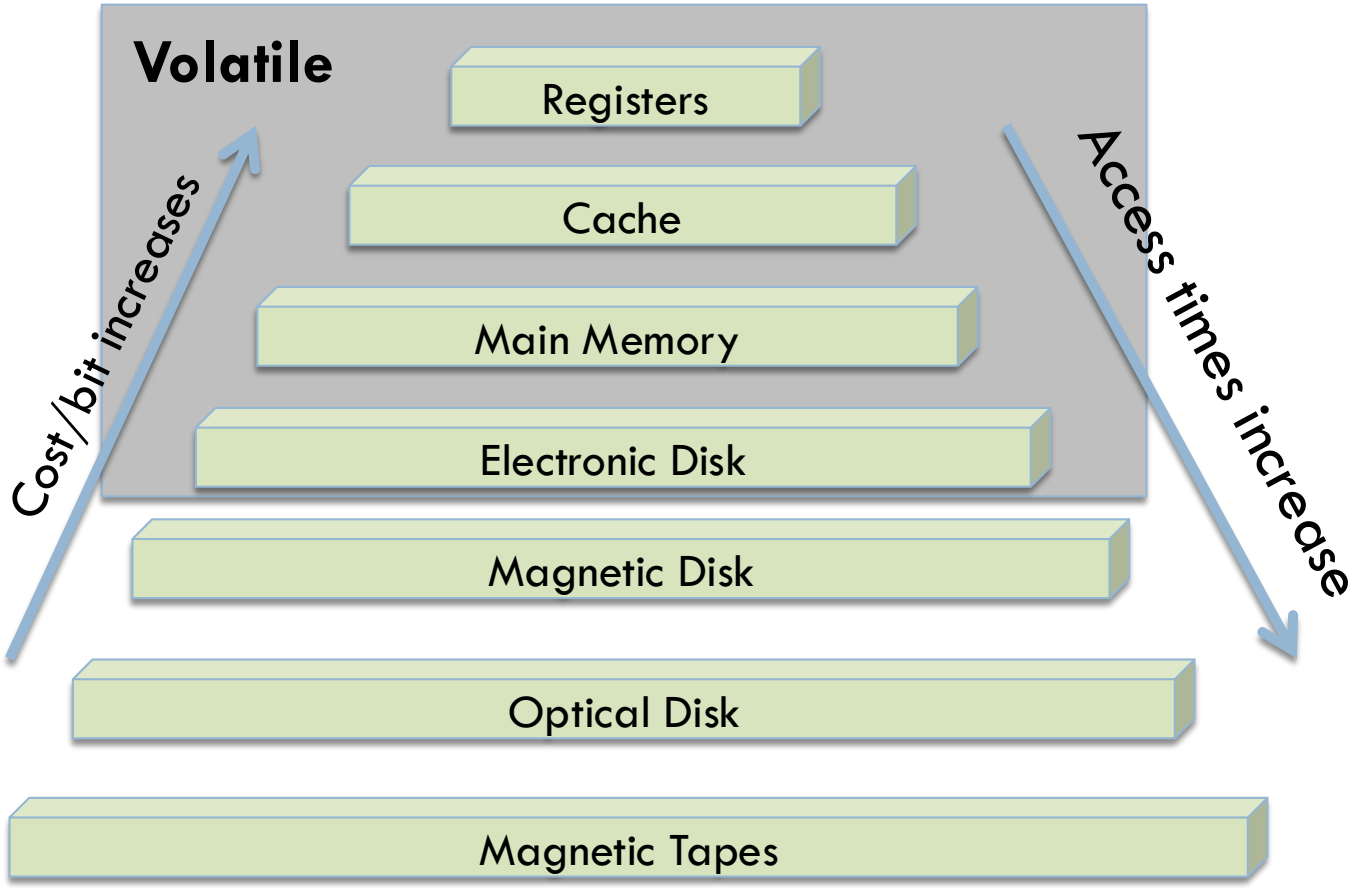
MEMORY

Memory

- Ideally the memory should be
  - Extremely **fast**: Faster than executing an instruction
    - CPU should not be held up by the memory
  - Abundantly **large**
  - Dirt **cheap**

- No current technology satisfies all these goals: they are contradictory!
  - It is all a matter of trade-off and calibration based on typical expected use of the CPU/memory/machine

Storage system hierarchy



Memory Hierarchy: Registers internal to the CPU

- Made of **same material** as the CPU
  - Just as fast as the CPU
- Storage capacity is typically very small: a handful / a few tens of virtual registers
  - May have more physical registers
- Programs explicitly address registers in **software**
  - **Compilers** map the (possibly infinite) set of variables in a program to a finite set of physical memory locations (registers), and deal with backup/restore code as needed

Memory hierarchy: Cache memory

- Mostly *controlled by hardware*
  - *But can be controlled by software*
- Trade off: slower speed than registers, but more capacity
  - Think about a temporary storage for "more" registers but at a "higher" cost of access (restoring a value from cache to its register)
- Typically pre-populated with the next memory location to be accessed: data pre-fetching (hardware or soft.)

When a program needs to read a memory word

- Start at L1: cache hardware checks if the needed line is in the cache
- If it is, that's a **cache hit**
  - Request satisfied from cache in about *1-10 clock cycles*
  - No memory access needed
- If needed line is not present in cache
  - **Cache miss in L1**, which translates into a read in L2. Repeat
  - **If no cache holds the data, read to memory: VERY long latency, 1000 of cycles possible**
    - Do not believe the sequential bandwidth numbers advertised as true in practical scenarios!

Caching is a powerful concept used elsewhere too. Let's see when …

① Large resource *can be divided* into pieces

② Some pieces *used more heavily* than others

- OS caching examples:
  - Pieces of heavily used files in main memory
    - Reduce disk accesses
  - Conversion of file names to disk addresses
  - Addresses of Web pages (URLs) as hosts

Main Memory

- Usually called **RAM** (Random Access Memory)
- Cache misses go to the main memory
- **Volatile**
  - Contents lost when power is turned off
- Memory size is of the order of several GB in most modern desktops

Loading and storing of memory addresses is the precursor to processing

- **`load()`** moves word from main memory to an internal register
- **`store()`** moves content from register to main memory
- CPU automatically loads instructions from main memory

The instruction execution cycle

- Instruction **fetch**ed from memory and stored in instruction register
- Instruction is **decode**d, and operands fetched from memory and stored in some register
- Instruction on operands is **execute**d next
- Result **store**d back in memory

Computers run most of their programs from (rewriteable) main memory

- Typically implemented in a technology called DRAM (dynamic random access memory)
- Ideal Scenario: Programs and data reside permanently in main memory. BUT …
  - Space is *limited*
  - Main memory is *volatile* storage

The contents of this slide-set are based on the following references

- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620*
- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 1]
- *Kay Robbins & Steve Robbins. Unix Systems Programming, 2nd edition, Prentice Hall ISBN-13: 978-0-13-042411-2.* [Chapter 1]