CSE 451: Operating Systems

Spring 2020

Module 1

Course Introduction

John Zahorjan

zahorjan@cs.washington.edu

Lecture Questions

- How will I get essential info and updates for the course?
- Why is explicitly identifying mechanism separately from policy important?
- What are the important roles/functions of an OS?
- What does control flow look like from the hardware point of view?
- How can the OS protect resources at times when it’s not even running?

Today's agenda

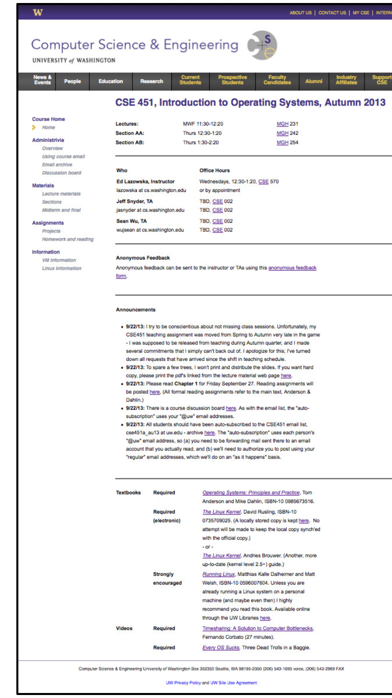
- Administrative Details
  - Course overview
    - course staff
    - general structure
    - the text(s)
    - policies
    - your to-do list
- OS overview
  - Trying to make sense of the topic

Course overview

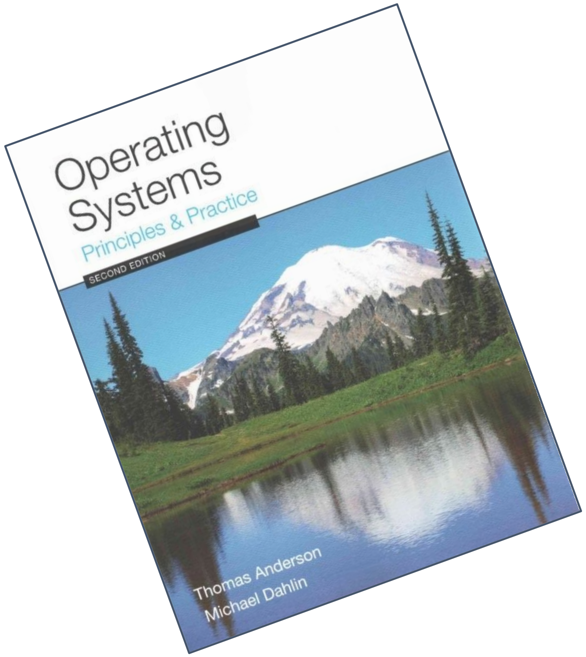
- Operationally, everything you need to know will be on the course web pages:
  http://www.cs.washington.edu/451/ and https://canvas.uw.edu/

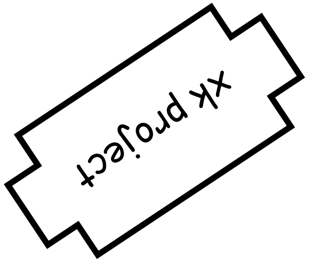
- Or on the course email:
  cse451a_sp20@uw.edu

- Or on the course discussion board:
  https://us.edstem.org/



Course Issue





Policy vs. Mechanism

- Policy is what you're trying to achieve
  - All users should get about the same amount of CPU time per second
    - Warning: that's a crude example, more of a goal than a policy. Actual policies later in the course.
- Mechanism is how you achieve that
  - The OS sets a hardware timer and switches tasks when it expires
- Roughly, class/reading is about policy, the projects are about mechanism

Class Resources

- The text
  - Really outstanding – written by current experts
    - Allows you to actually figure out how things work
  - Think of it as helping you to understand, and dig deeper than, the lecture, section, and project material

- Other resources
  - Many online; some of them are essential

- Policies
  - Collaboration vs. cheating
  - Projects: late policy

The Project(s)

- Start your projects early
- Projects
  - Start them early
  - Four of them
  - Start them early
  - Teams of two. You’re likely to be happier if you form a team on your own than if we form one for you!
  - Do not start your project late. You will regret it.
  - Start them early
  - We’re imagining having to work only remotely is not much of a change
    - but we’ll see...

Late Policy

- There is one
- It's some balancing act of these principles
  - You know better than we do what all your responsibilities are
  - We owe it to you (and your partner!) to provide sufficient incentive to stay on schedule that you don't get yourself in trouble
- Don't get yourself in trouble!

What is an Operating System?

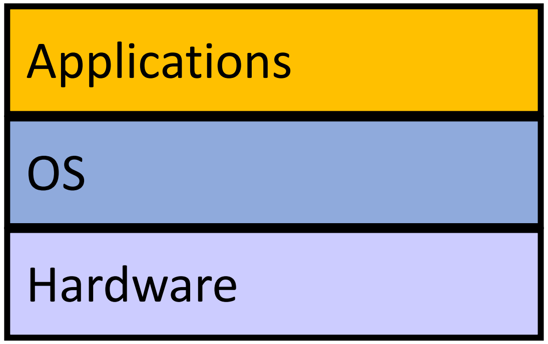
- Answers:
  - I don't know.
  - Nobody knows.
  - The book claims to know – read Chapter 1.
  - They're programs – big hairy programs
    - The Linux source has over 27M lines of C
    - Windows has way, way more…

What is an Operating System?

- Answers:
  - I don't know.
  - Nobody knows.
  - The book claims to know – read Chapter 1.
  - They’re programs – big hairy programs
    - The Linux source you'll be compiling has over 1.7M lines of C

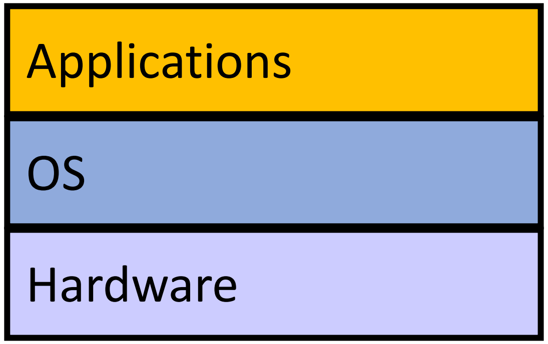
Okay. What are some functions/goals of an OS?

The Traditional Graphic



- This depiction invites you to think of the OS as a library
- (This depiction invites you to think of the hardware as a library...)

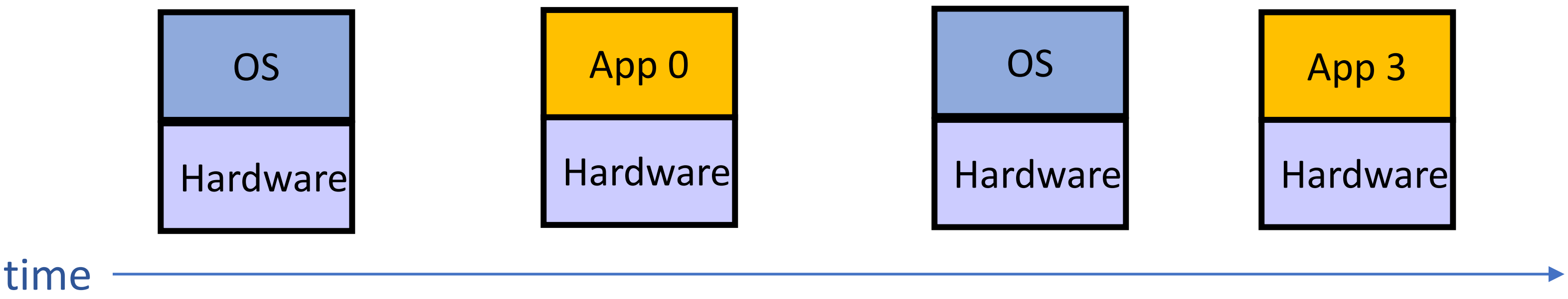
What's Right With That Picture



- The OS *is* between you (the app) and the hardware
  - Your application cannot manipulate the hardware, it has to ask (politely)
  - This is the basis of security

- The OS *is (partly)* a library
  - Sometimes apps explicitly invoke it to request some function
  - Some functionality is in the kernel because it turns out to be fastest to put it there, not because it absolutely has to be there

What’s Wrong With That Picture



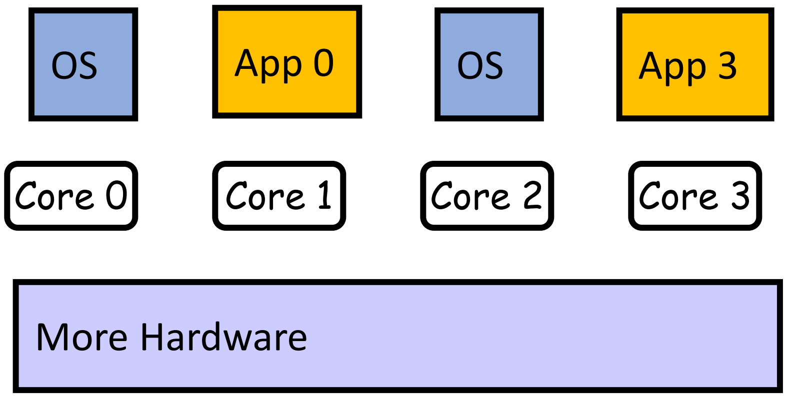
- The OS *isn’t* between you and (some of) the hardware
  - When you’re running, your code uses some of the hardware directly
    - Which hardware?
    - Why (not interpose the OS between you and that hardware)?
    - Why not have separate hardware for the OS and the apps?

- The OS *isn’t* a library
  - You don’t tell it when to run, it tells you when to run
  - You don’t tell it what code to run, it makes up its own mind
  - *(Okay, sometimes you tell it when to run...)*

What's Wrong With That Last Picture

It's from 2004

- why take turns when you can all go at once?



Execution from the Software Point of View

- Memory
  - I have values in local and global variables and you can't touch them.
    - You don't even exist.
- CPU
  - I execute this statement, then I execute the next statement, sometimes I loop, sometimes I call, but it's all me and my code (or library code called by me)
    - My code determines where execution goes
- Shared System
  - Okay, I know there are lots of program running but I don't care because they have no effect on me
    - except maybe for performance impacts
  - *This is one of the primary goals/abstractions of the OS*

Execution from the Hardware POV

- Tick, tick, tick, ...
- Fetch an instruction, execute an instruction
  - Each instruction may modify some machine *state*
    - Update some register or memory value(s)
- From the hardware's point of view, there are no programs
  - No variables, just addresses
  - No statements, just instructions
- The hardware is a state machine
  - there is no OS or application(s)
  - there is just the current state of the machine and the next state
    - the state of the machine is more complicated than what you can see from the application level
    - for example, the state includes the set of page tables currently in use
- *However, the hardware is designed to enable us to write OS's and applications and do other useful things*

Execution from the OS POV

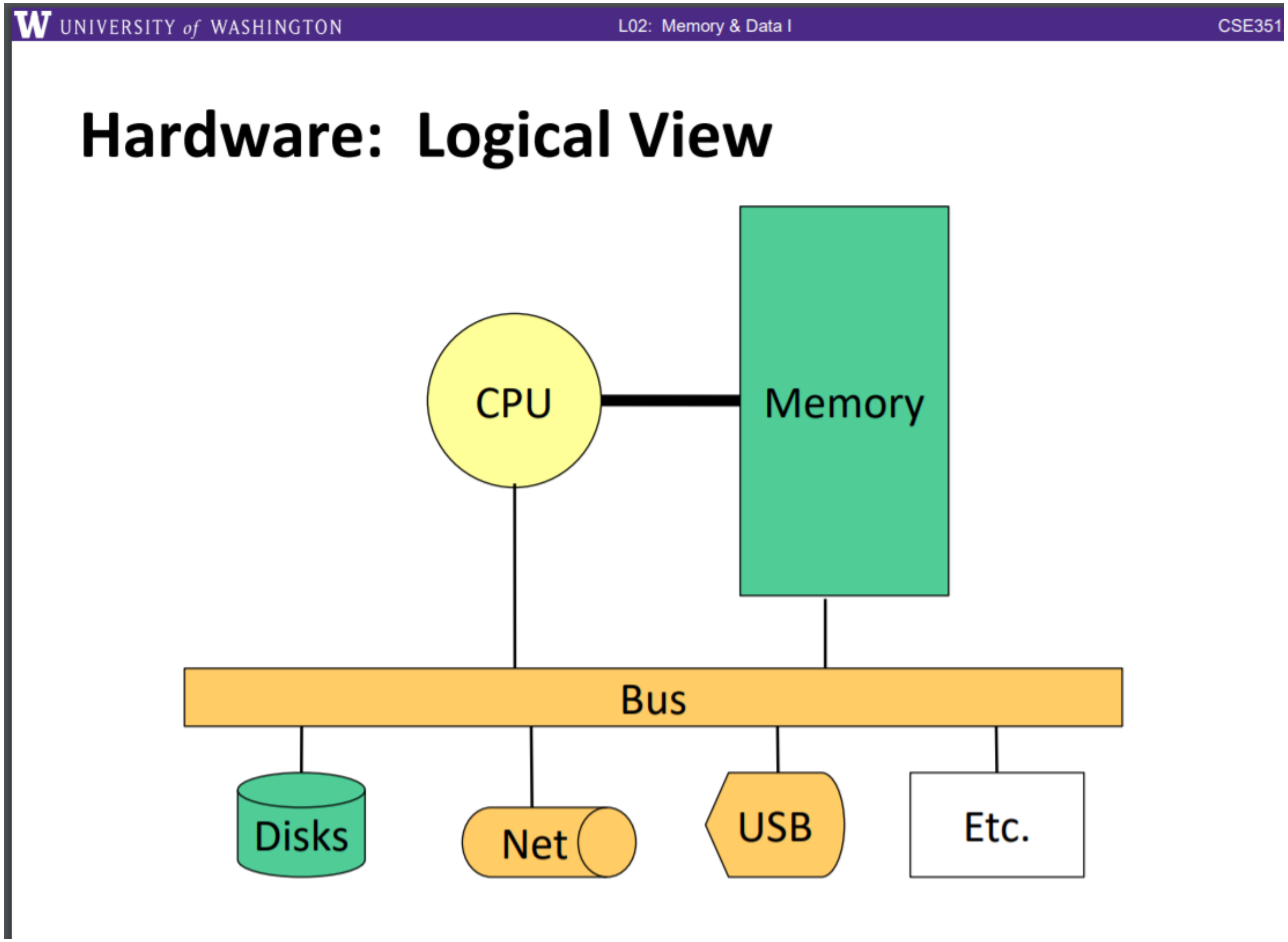
- On the one hand, the OS is the benefactor of the abstractions it creates
  - A lot of its execution feels just like application code
- On the other hand, the OS interfaces with/control/responds to the hardware
  - That code can feel more foreign
- Some features of the hardware are designed to make it possible to write a (reasonable/efficient) OS
  - The OS insulates application programs from ever having to deal directly with many aspects of the hardware
- The OS has to deal with hardware asynchrony
  - The OS provides apps an abstraction of orderly execution, where nothing unexpected happens
  - The hardware doesn't
  - The next instruction to be executed sometimes has nothing to do with the one currently being executed, or its location in memory

Execution from the OS POV

- Suppose the system is running both processes A and B
  - each has its own address space
  - each has its own stack
  - when the OS reallocates a core from A to B it must switch address spaces
    - which address space is it using while it's switching between the two?
    - which stack is it using while it's switching?

- the OS must sometimes deal with physical memory
  - so, an address could mean a value in A's address space, or B's, or physical memory's

The Abstract View of Hardware



The xk View of Hardware (Partial)

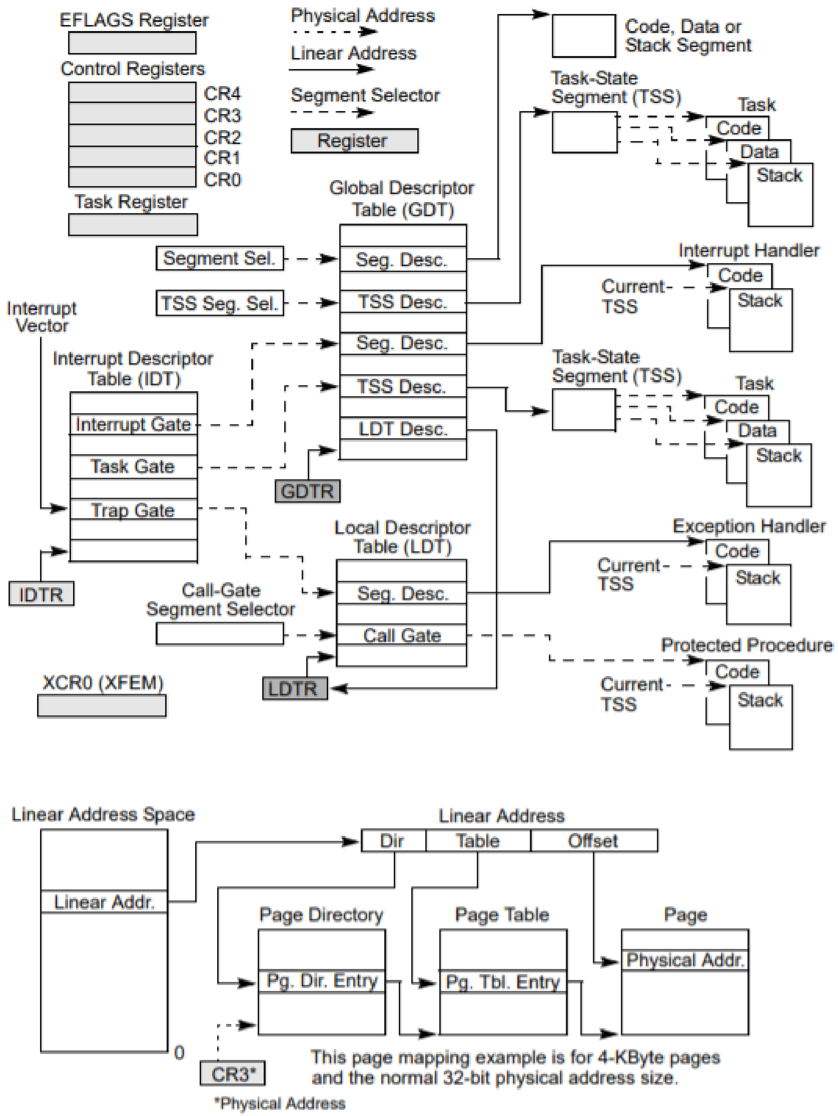


Figure 2-1. IA-32 System-Level Registers and Data Structures

The OS and Hardware

- An OS mediates programs' access to hardware resources (*sharing* and *protection*)
  - computation (CPU)
  - volatile storage (memory) and persistent storage (disk, etc.)
  - network communications (TCP/IP stacks, Ethernet cards, etc.)
  - input/output devices (keyboard, display, sound card, etc.)
- The OS abstracts hardware into logical resources and well-defined interfaces to those resources (*ease of use*)
  - processes/threads (CPU, memory, instruction execution)
  - files (disk)
  - sockets (network)
  - streams (keyboard, display, sound card, etc.)

The text says an OS is ...

- A Referee
  - Mediates resource sharing
- An Illusionist
  - Masks hardware limitations
- Glue
  - Provides common services

Why bother with an OS?

- Application benefits
  - programming simplicity
    - see high-level abstractions (files) instead of low-level hardware
    - abstractions are reusable across many programs
  - portability (across machine configurations or architectures)
    - device independence: 3com card or Intel card?
- User benefits
  - safety
    - program “sees” its own virtual machine, thinks it “owns” the computer
    - OS protects programs from each other
    - OS multiplexes resources across programs
  - efficiency (cost and speed)
    - share one computer across many users
    - concurrent execution of multiple programs

The Major OS Issues

- **structure**: how is the OS organized?
- **sharing**: how are resources shared across users?
- **naming**: how are resources named and what is the scope?
- **protection**: how is one user/process protected from another?
- **security**: how is the integrity of the OS and its resources ensured?
- **performance**: how do we avoid making it all slow?
- **availability**: can you always access the services you need?
- **reliability**: what happens if something goes wrong (either with hardware or with a program)?
- **extensibility**: can we add new features?
- **communication**: how do programs exchange information, including across a network?

More OS Issues…

- **concurrency**: how are simultaneous activities (computation and I/O) created and controlled?
- **scale**: what happens as demands or resources increase?
- **persistence**: how do you make data last longer than program executions?
- **distribution**: how do we allow a computation to span hardware (machine/network) boundaries?
- **accounting**: how do we keep track of resource usage, and perhaps charge for it?
- **auditing**: can we reconstruct who did what to whom?

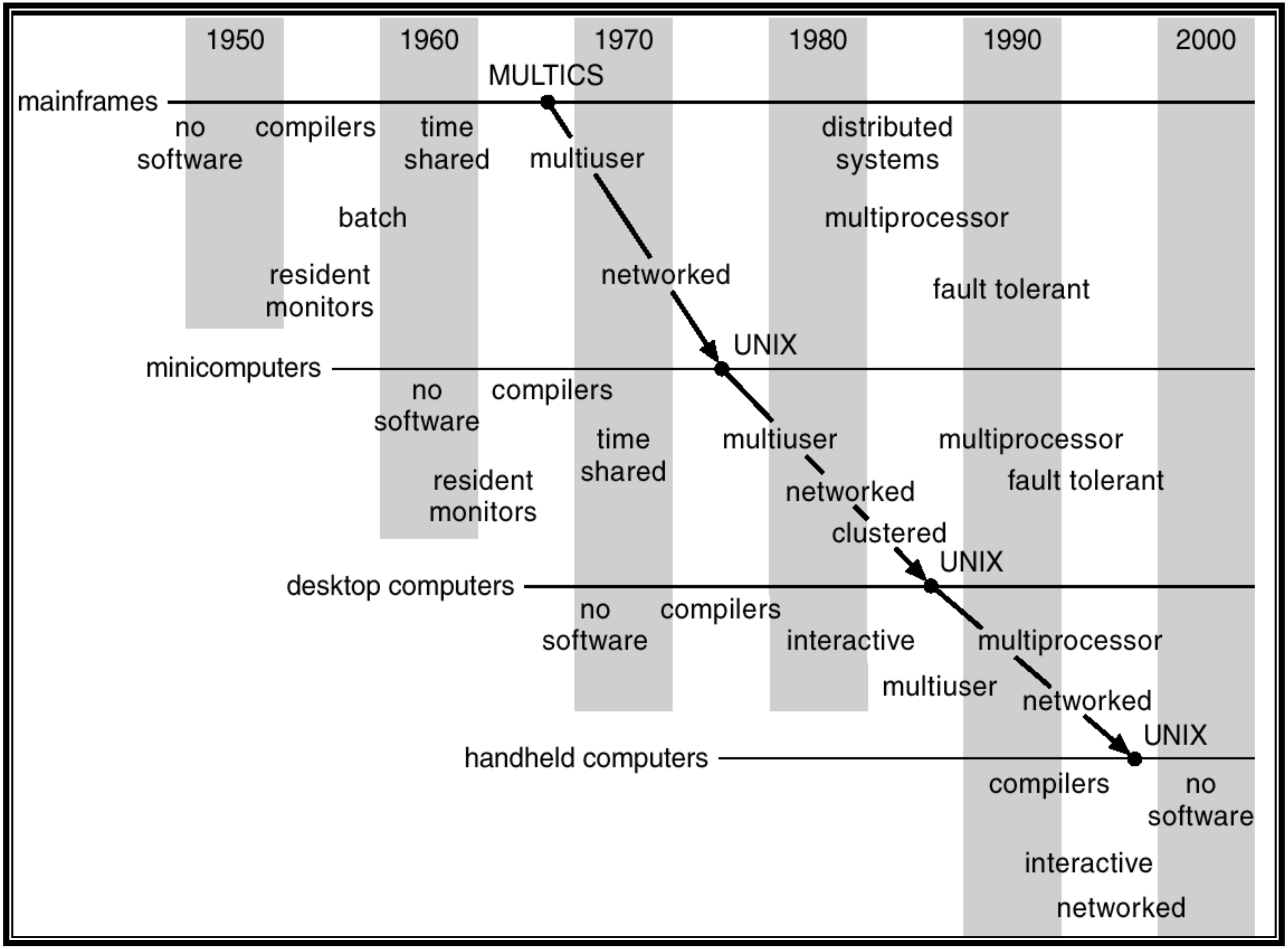
There are tradeoffs – not right and wrong!

(Ok, some things are clearly wrong, but there is no right.)

Hardware/Software Changes Over Time

- 1960s: mainframe computers (IBM)
- 1970s: minicomputers (DEC)
- 1980s: microprocessors and workstations (SUN), local-area networking, the Internet
- 1990s: PCs (rise of Microsoft, Intel, Dell), the Web
- 2000s:
  - Internet Services / Clusters (Amazon)
  - General Cloud Computing (Google, Amazon, Microsoft)
  - Mobile/ubiquitous/embedded computing (iOS, Android)
- 2010s: sensor networks, “data-intensive computing,” computers and the physical world (“pervasive computing”)
- 2020: it’s up to you!!

Progression of concepts and form factors



Has it all been discovered?

- New challenges constantly arise
  - embedded computing (e.g., home assistants)
  - sensor networks (very low power, memory, etc.)
  - peer-to-peer systems
  - ad hoc networking
  - scalable server farm design and management (e.g., Google)
  - software for utilizing huge clusters (e.g., MapReduce, Bigtable)
  - overlay networks (virtualizing the Internet)
  - finding bugs in system code (e.g., model checking)
  - high value real time systems (autonomous vehicles)
- Old problems constantly re-define themselves
  - the evolution of smart phones recapitulated the evolution of PCs, which had recapitulated the evolution of minicomputers, which had recapitulated the evolution of mainframes

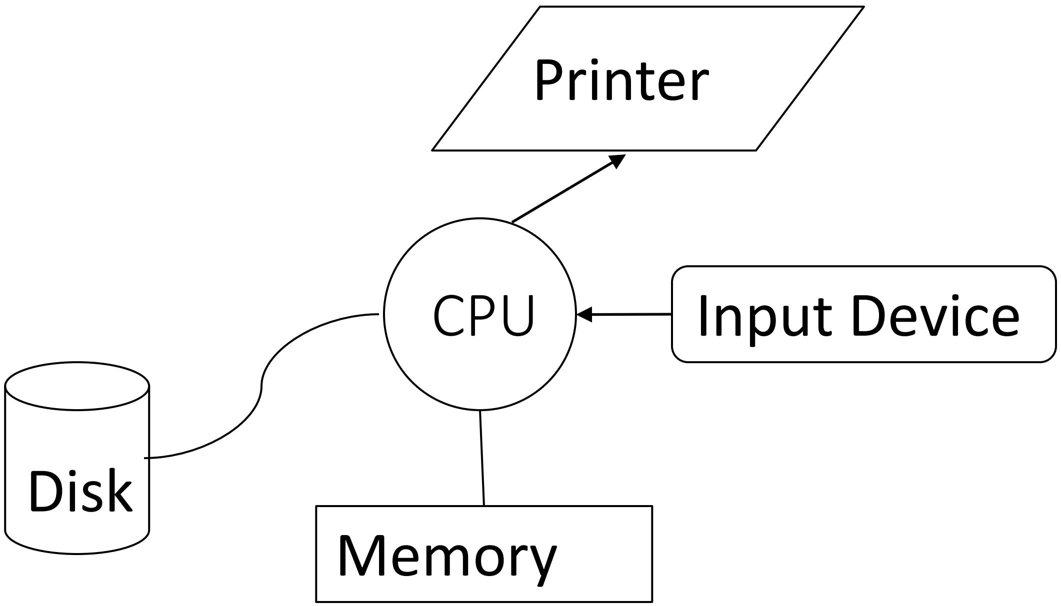
Protection and security as an example

- none
- OS from my program
- your program from my program
- my program from my program
- access by intruding individuals
- access by intruding programs
- denial of service
- distributed denial of service
- spoofing
- spam
- worms
- viruses
- stuff you download and run knowingly (bugs, trojan horses)
- stuff you download and run obliviously (cookies, spyware)
- web tracking

In the Beginning...

- 1943
  - T.J. Watson (created IBM):
    " I think there is a world market for maybe five computers."
- Fast forward … 1950
  - There are maybe 20 computers in the world
    - They were unbelievably expensive
    - Imagine this: machine time is more valuable than person time!
    - Ergo: *efficient use of the hardware is paramount*
  - Operating systems are born
    - (Why?)
    - They carry with them the vestiges of these ancient forces

The Primordial Computer



The OS as a Linked Library

- In the very beginning…
  - OS was just a library of code that you linked into your program; programs were loaded in their entirety into memory, and executed
    - "OS" had an "API" that let you control the disk, control the printer, etc.
  - Interfaces were literally switches and blinking lights
  - When you were done running your program, you'd leave and turn the computer over to the next person

- *Recapitulation: Paul Allen writing a bootstrap loader for the Altair as the plane was landing in New Mexico*

Seattle Trivia (Wikipedia “Altair Basic”, edited)

Bill Gates recalls that, when he and Paul Allen read about the Altair in the January 1975 issue of Popular Electronics, they understood that the price of computers would soon drop to the point that selling software for them would be a profitable business. They contacted MITS founder Ed Roberts, told him that they were developing an interpreter, and asked whether he would like to see a demonstration. This followed the common engineering industry practice of a trial balloon, an announcement of a non-existent product to gauge interest.

Gates and Allen had neither an interpreter nor even an Altair system on which to develop and test one. However, Allen had written an Intel 8008 emulator for their previous venture. He adapted this emulator based on the Altair programmer guide, and they developed and tested the interpreter on Harvard's PDP-10. Harvard officials were not pleased when they found out, but there was no written policy that covered the use of this computer.

The finished interpreter, including its own I/O system and line editor, fit in only four kilobytes of memory, leaving plenty of room for the interpreted program. In preparation for the demo, they stored the finished interpreter on a punched tape that the Altair could read, and Paul Allen flew to Albuquerque.

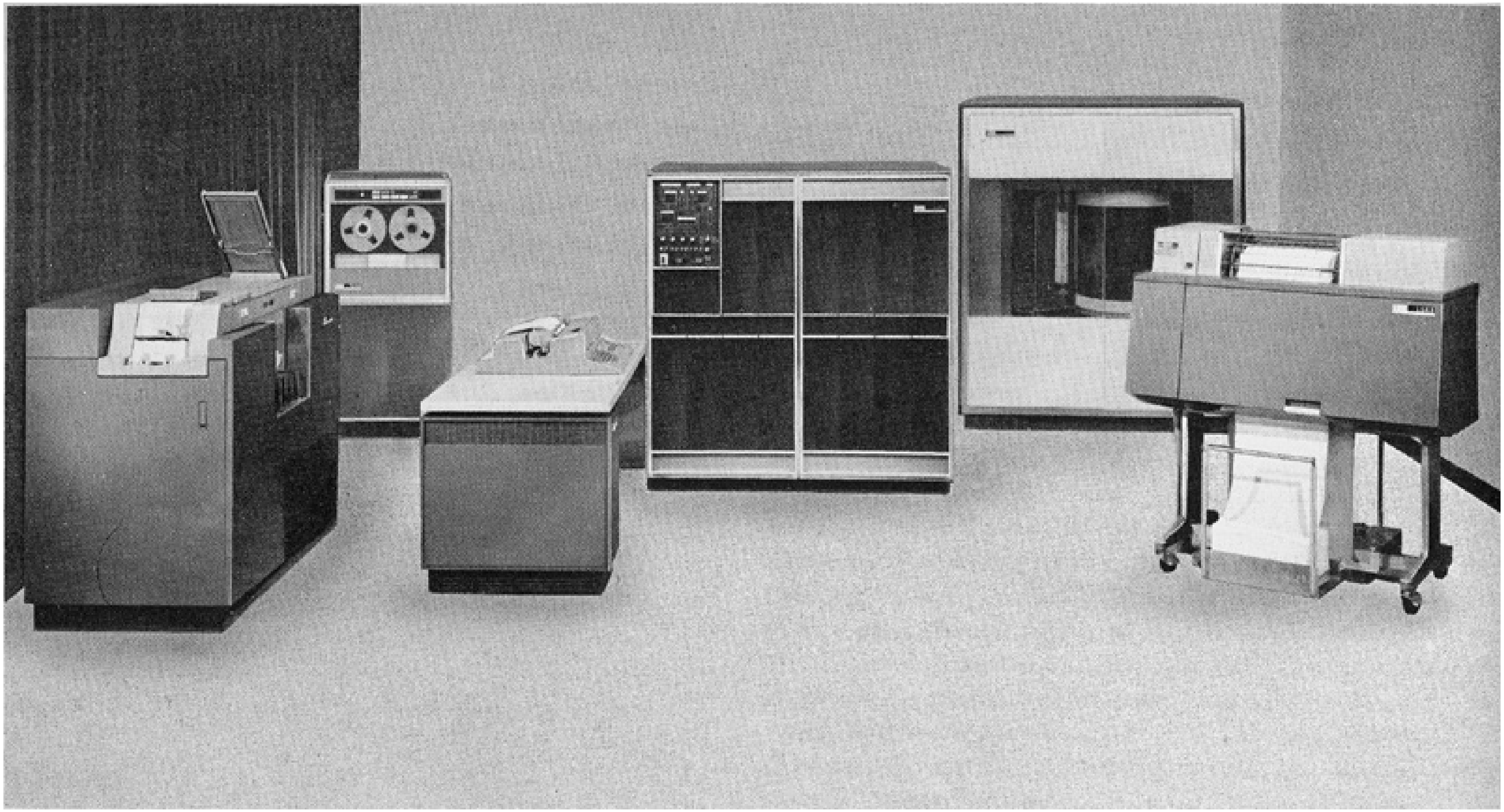
On final approach, Allen realized that they had forgotten to write a bootstrap program to read the tape into memory. Writing in 8080 machine language, Allen finished the program before the plane landed. Only when they loaded the program onto an Altair and saw a prompt asking for the system's memory size did Gates and Allen know that their interpreter worked on the Altair hardware. Later, they made a bet on who could write the shortest bootstrap program. Gates won.

Asynchronous I/O

- The disk was really slow
- The CPU was really expensive
- Add hardware so that the disk could operate without tying up the CPU
  - Disk controller
- Hotshot programmers could now write code that:
  - Starts an I/O
  - Goes off and does some computing
  - Checks if the I/O is done at some later time
- Upside
  - Helps increase (expensive) CPU utilization by overlapping CPU and I/O
- Downsides
  - It's hard to get right
  - The benefits are job specific

The OS as a “Resident Monitor”

- Everyone was using the same library of code
- Why not keep it in memory?
- While we’re at it, make it capable of loading Program 4 while running Program 3 and printing the output of Program 2
  - SPOOLing – Simultaneous Peripheral Operations On-Line
- What new requirements does this impose?

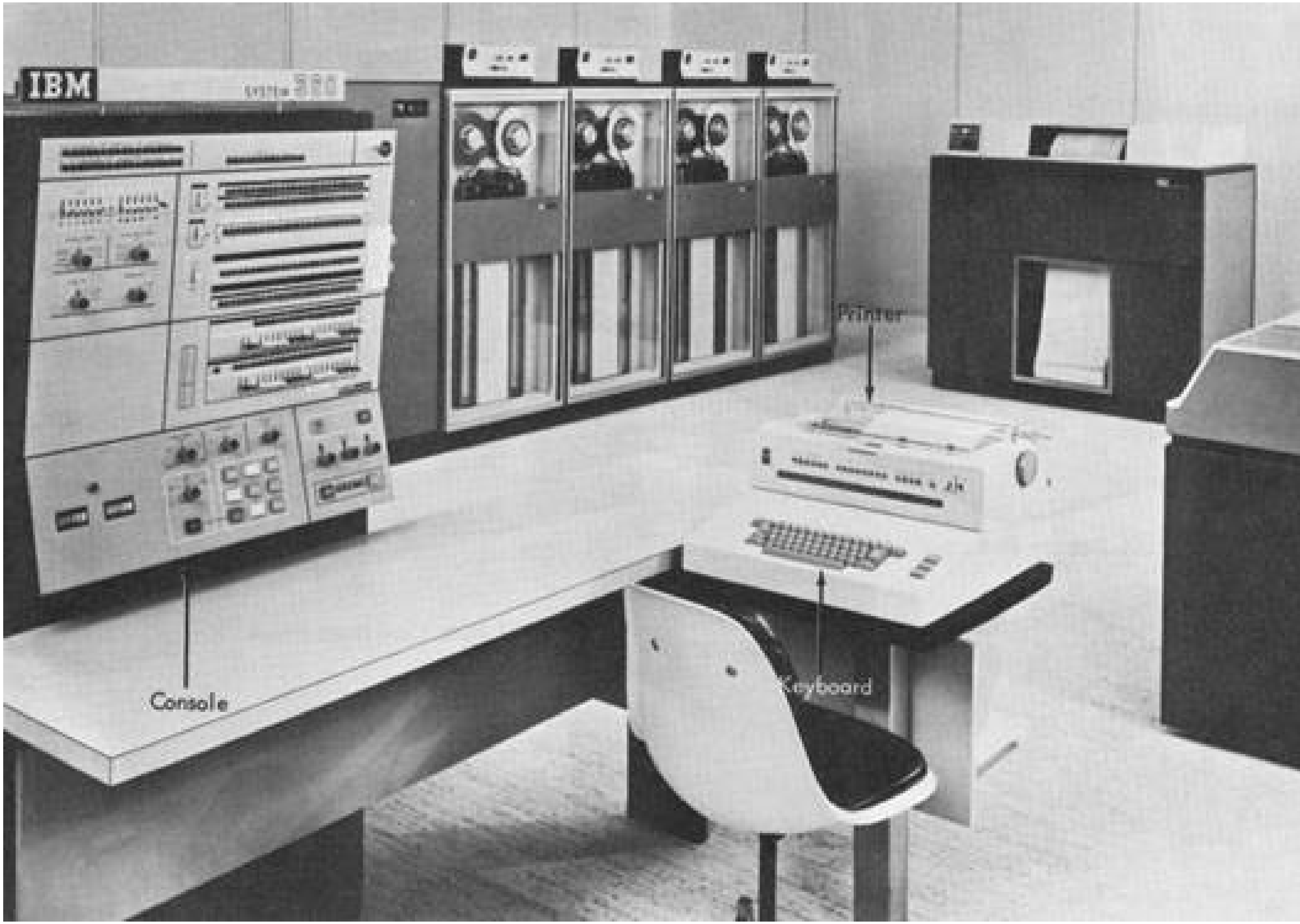


IBM 1401

Multiprogramming

- To further increase system utilization, multiprogramming OSs were invented
  - keeps multiple runnable jobs loaded in memory at once
  - overlaps I/O of one job with compute of another
  - Don't need asynchronous I/O within individual jobs to keep CPU busy
    - Life of application programmer becomes simpler
  - OS *blocks* process waiting for an I/O completion
  - How do we tell when devices are done?
    - Interrupts
    - Polling

- What new requirements does this impose?



IBM System 360

(An aside on protection)

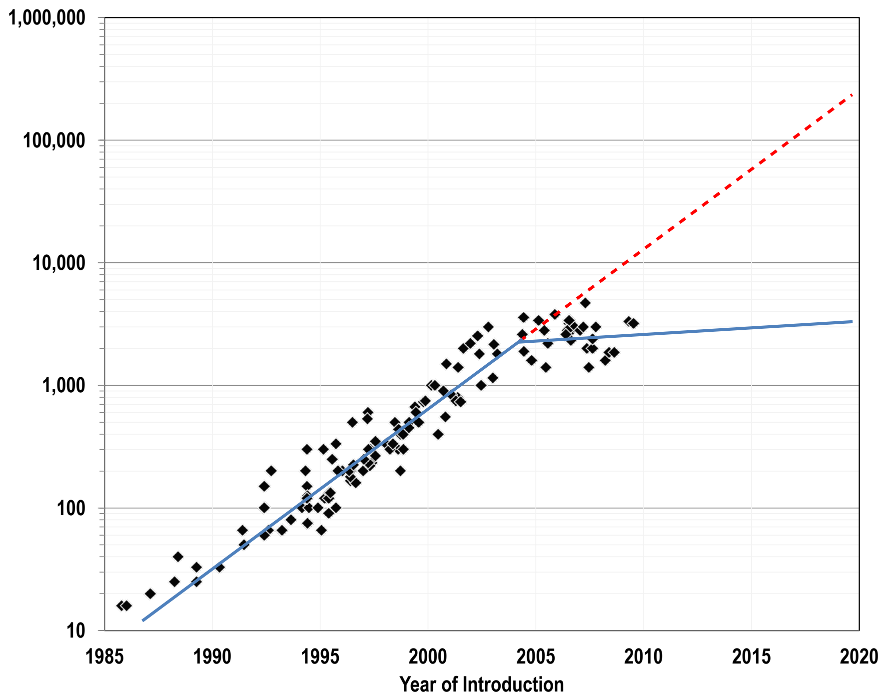
- Applications/programs/jobs execute directly on the CPU, but cannot touch anything except “their own memory” without OS intervention

- Applications can’t be allowed direct access to hardware, e.g., disk
  - Why not?

- Applications are allowed constrained direct use of CPU
  - Why?
  - Constrained how?

(An aside on concurrency)

- Transistor density continues to increase (Moore’s Law), but individual cores aren’t getting faster very fast – instead, we’re getting more of them (the number doubles on roughly the old 18-month cycle)



- The burden is on the programmer to use an ever increasing number of cores

- A lot of this course is about concurrency
  - It used to be a bit esoteric

Timesharing

- To support interactive use, create a timesharing OS:
  - multiple terminals into one machine
  - each user has illusion of entire machine to him/herself
  - optimize *response time*, perhaps at the cost of *throughput*

- Timeslicing
  - divide CPU (equally) among the users
  - if job is truly interactive (e.g., editor), then can jump between programs and users faster than users can generate load
  - permits users to interactively view, edit, debug running programs

- MIT CTSS system (operational 1961) was among the first timesharing systems
  - only one user memory-resident at a time (32KB memory!)

- MIT Multics system (operational 1968) was the first large timeshared system
  - a huge number of fundamental OS concepts can be traced back to Multics!
  - “second system syndrome”

- In early 1980s, a *single* timeshared VAX-11/780 (like the one in the Allen Center atrium) ran computing for *all of* CSE.

- A typical VAX-11/780 was 1 MIPS (1 MHz) and had 1MB of RAM and 100MB of disk.
  - Compare that to your phone

Parallel systems

- Some applications can be written as multiple parallel threads or processes
  - can speed up the execution by running multiple threads/processes simultaneously on multiple CPUs [Burroughs D825, 1962]
  - need OS and language primitives for dividing program into multiple parallel activities
  - need OS primitives for fast communication among activities
    - degree of speedup dictated by communication/computation ratio
  - many flavors of parallel computers today
    - SMPs (symmetric multi-processors)
    - MPPs (massively parallel processors)
    - NOWs (networks of workstations)
    - Massive clusters (Google, Amazon.com, Microsoft)
    - Computational grid (SETI @home)

Personal computing

- Primary goal was to enable new kinds of applications
- Bit mapped display [Xerox Alto,1973]
  - new classes of applications
  - new input device (the mouse)
- Move computing near the display
  - why?
- Window systems
  - the display as a managed resource
- Local area networks [Ethernet]
  - why?
- Effect on OS?

Distributed OS

- Distributed systems to facilitate use of geographically distributed resources
  - workstations on a LAN
  - servers across the Internet
- Supports communications among processes
  - interprocess communication
    - message passing, shared memory
  - networking stacks
- Sharing of distributed resources (hardware, software)
  - load balancing, authentication and access control, …
- Speedup isn’t the issue
  - access to diversity of resources is goal

Client/server computing

- Mail service
- File service
- Calendar service
- Compute server
- Game service
- Web service
- Streaming service
- Backup/Sync service
- etc.

The Major OS Issues

- **structure**: how is the OS organized?
- **sharing**: how are resources shared across users?
- **naming**: how are resources named (by users, by programs)?
- **protection**: how is one user/program protected from another?
- **security**: how is the integrity of the OS and its resources ensured?
- **performance**: how do we make it all go fast?
- **availability**: can you always access the services you need?
- **reliability**: what happens if something goes wrong (either with hardware or with a program)?
- **extensibility**: can we add new features?
- **communication**: how do programs exchange information, including across a network?

More OS Issues…

- **concurrency**: how are parallel activities (computation and I/O) created and controlled?
- **scale**: what happens as demands or resources increase?
- **persistence**: how do you make data last longer than program executions?
- **distribution**: how do multiple computers interact with each other?
- **accounting**: how do we keep track of resource usage, and perhaps charge for it?
- **auditing**: can we reconstruct who did what to whom?

Lecture Question Answers

- How will I get essential info and updates for the course?
  - Course discussion board, which you find on the
  - Course web
- Why is explicitly identifying mechanism separately from policy important?
  - Policy is what we want to achieve, mechanism is how we achieve it
  - Defining interfaces (e.g., the mechanism layer) enables experimentation with policy and with different ways to provide the mechanism
- What are the important roles/functions of an OS?
  - Abstracts the hardware into more convenient interfaces that both facilitate efficient implementation of application software and efficient execution of that software
  - Provides abstractions that can overcome hardware limitations
  - Provides isolation and protection
  - Provides communication and sharing
  - Allocates resources, both as a mechanism and as a policy

Lecture Question Answers

- What does control flow look like from the hardware point of view?
  - The hardware's view horizon is the next instruction to be executed
  - Any notion of OS or running program is at a level above the hardware (created by the OS)
- How can the OS protect resources at times when it's not even running?
  - The hardware implements features designed as mechanism that allows OS's to achieve their policy goals

Next time

- Architectural support for OS