Operating Systems: Principles and Practice

Mark Zbikowski
Gary Kimura
(kudos to Tom Anderson)

How This Course Fits in the UW CSE Curriculum

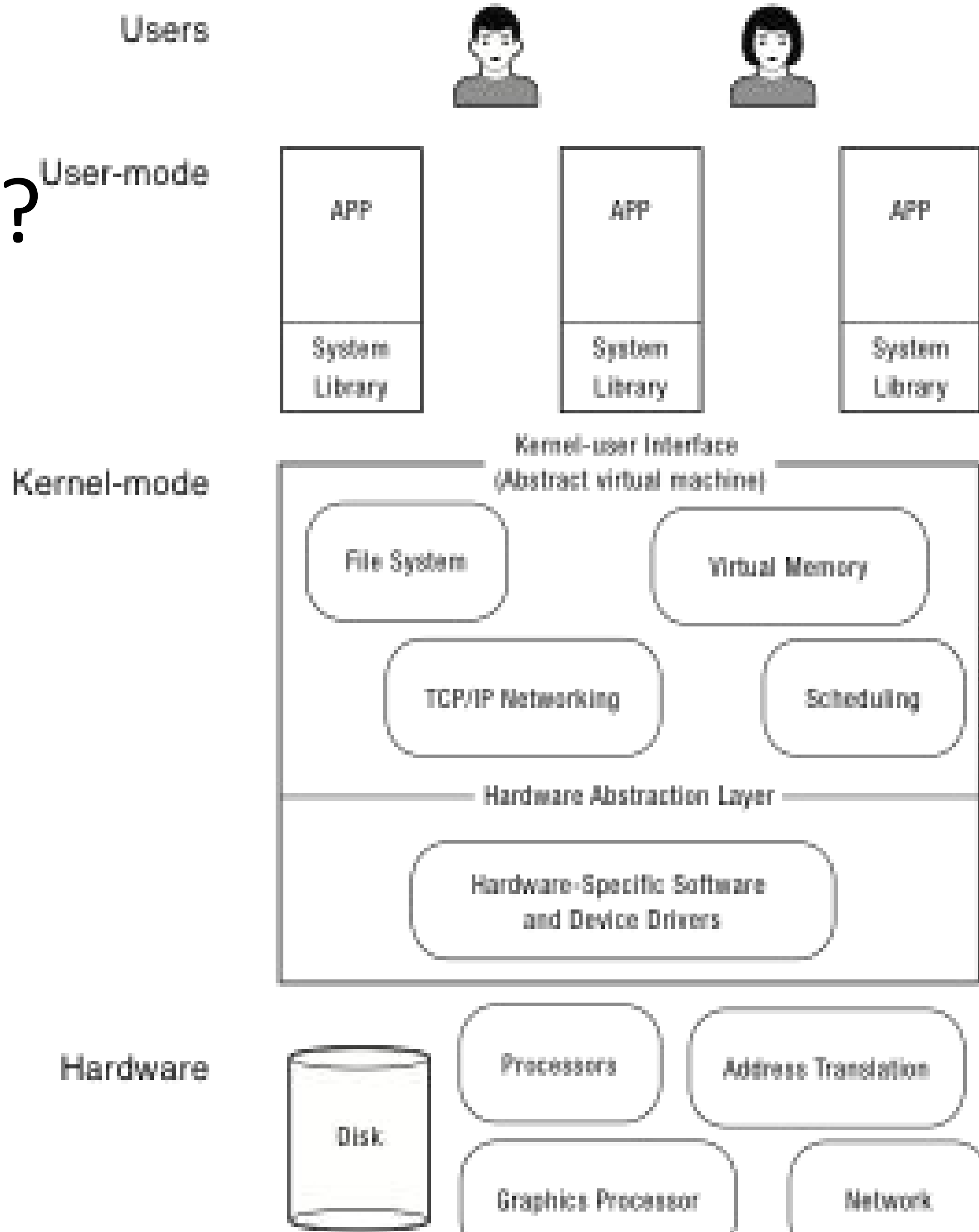
- CSE 333: Systems Programming
  - Project experience in C/C++
  - How to use the operating system interface
- CSE 451: Operating Systems
  - How to make a single computer work reliably
  - How an operating system works internally
- CSE 452: Distributed Systems (spring 2018)
  - How to make a set of computers work reliably, despite failures of some nodes

Main Points (for today)

- Operating system definition
  - Software to manage a computer's resources for its users and applications
- OS challenges
  - Reliability, security, responsiveness, portability, …
- OS history
  - How did we get here?
- How I/O works

What is an operating system?

- Software to manage a computer's resources for its users and applications



Operating System Roles

- Referee:
  - Resource allocation among users, applications
  - Isolation of different users, applications from each other
  - Communication between users, applications
- Illusionist
  - Each application appears to have the entire machine to itself
  - Infinite number of processors, (near) infinite amount of memory, reliable storage, reliable network transport
- Glue
  - Libraries, user interface widgets, …

Example: File Systems

- Referee
  - Prevent users from accessing each other's files without permission
  - Even after a file is deleting and its space re-used
- Illusionist
  - Files can grow (nearly) arbitrarily large
  - Files persist even when the machine crashes in the middle of a save
- Glue
  - Named directories, printf, …

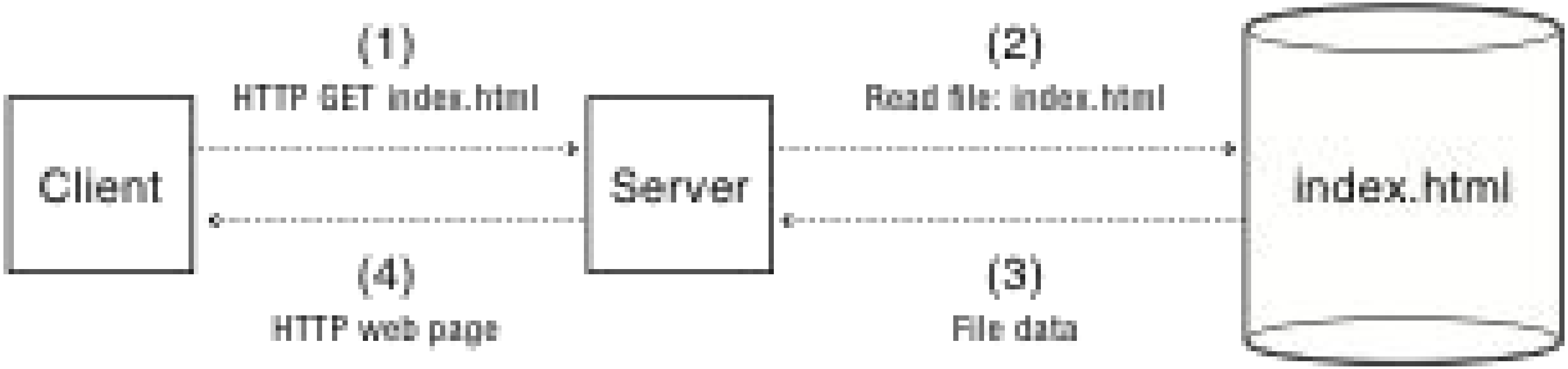
Question

- What (hardware, software) do you need to be able to run an untrustworthy application?

Question

- How should an operating system allocate processing time between competing uses?
  - Give the CPU to the first to arrive?
  - To the one that needs the least resources to complete?   To the one that needs the most resources?

Example: web service



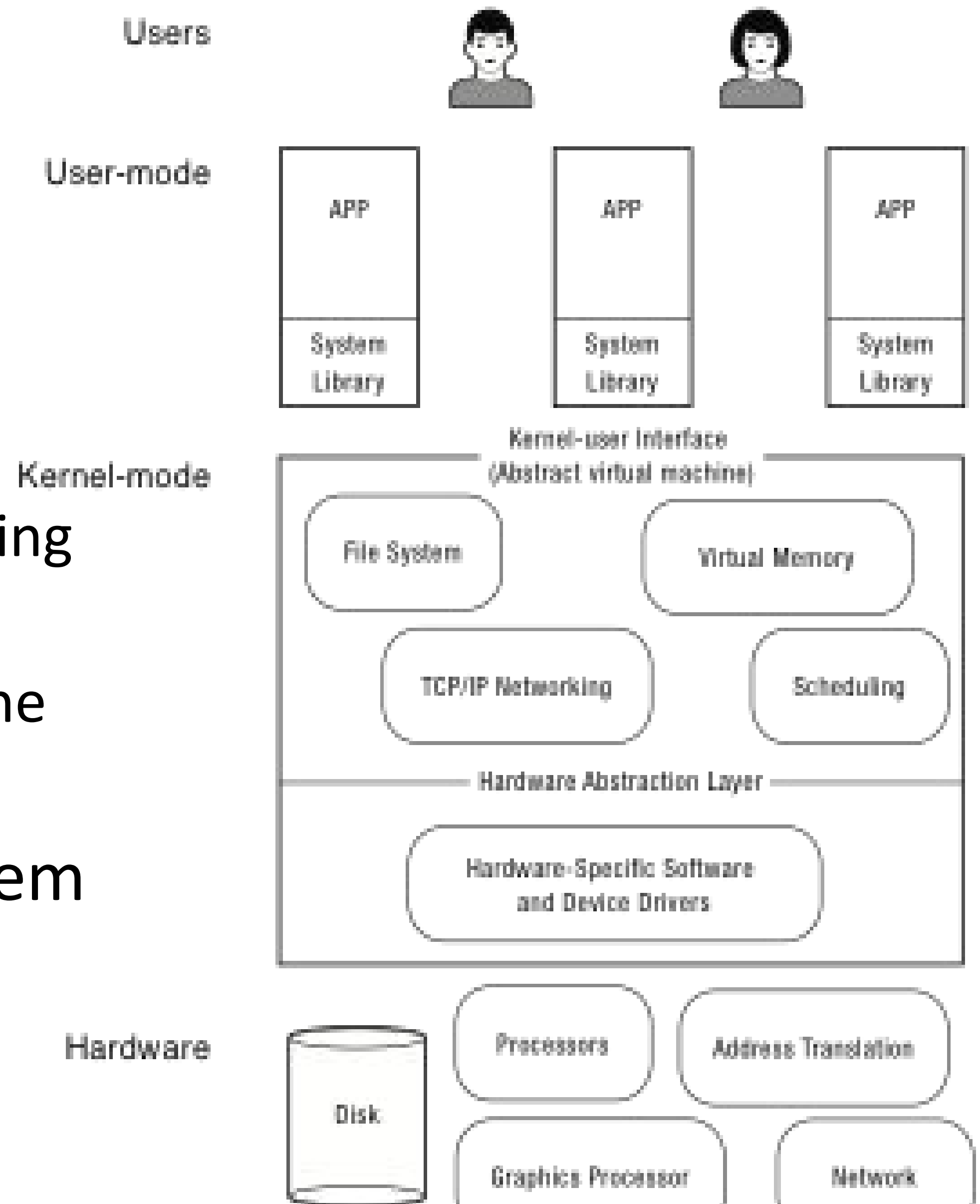
- How does the server manage many simultaneous client requests?
- How do we keep the client safe from spyware embedded in scripts on a web site?
- How do make updates to the web site so that clients always see a consistent view?

OS Challenges

- Reliability
  - Does the system do what it was designed to do?
- Availability
  - What portion of the time is the system working?
  - Mean Time To Failure (MTTF), Mean Time to Repair
- Security
  - Can the system be compromised by an attacker?
- Privacy
  - Data is accessible only to authorized users

OS Challenges

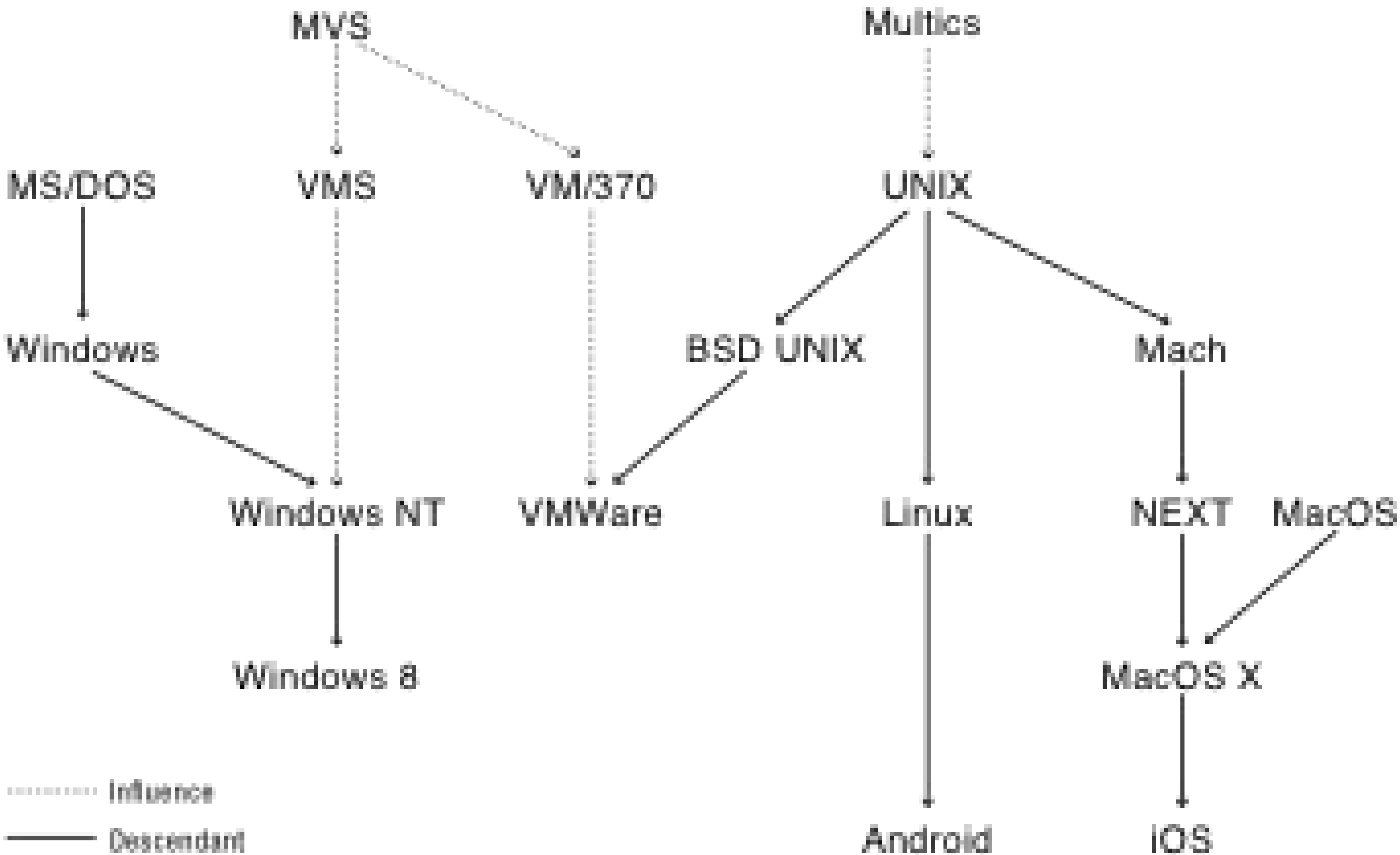
- Portability
  - For programs:
    - Application programming interface (API)
    - Abstract virtual machine (AVM)
  - For the operating system
    - Hardware abstraction layer



OS Challenges

- Performance
  - Latency/response time
    - How long does an operation take to complete?
  - Throughput
    - How many operations can be done per unit of time?
  - Overhead
    - How much extra work is done by the OS?
  - Fairness
    - How equal is the performance received by different users?
  - Predictability
    - How consistent is the performance over time?

OS History


MVS
Multics
MS/DOS
VMS
VM/370
UNIX
Windows
BSD UNIX
Mach
Windows NT
VMWare
Linux
NEXT
MacOS
Windows 8
MacOS X
Android
iOS
Influence
Descendant

Computer Performance Over Time

	1981	1997	2014	Factor (2014/1981)
Uniprocessor speed (MIPS)	1	200	2500	2.5K
CPUs per computer	1	1	10+	10+
Processor MIPS/$	$100K	$25	$0.20	500K
DRAM Capacity (MiB)/$	0.002	2	1K	500K
Disk Capacity (GiB)/$	0.003	7	25K	10M
Home Internet	300 bps	256 Kbps	20 Mbps	100K
Machine room network	10 Mbps (shared)	100 Mbps (switched)	10 Gbps (switched)	1000
Ratio of users to computers	100:1	1:1	1:several	100+

Early Operating Systems: Computers Very Expensive

- One application at a time
  - Had complete control of hardware
  - OS was runtime library
  - Users would stand in line to use the computer
- Batch systems
  - Keep CPU busy by having a queue of jobs
  - OS would load next job while current one runs
  - Users would submit jobs, and wait, and wait, and

Time-Sharing Operating Systems: Computers and People Expensive

- Multiple users on computer at same time
  - Multiprogramming: run multiple programs at same time
  - Interactive performance: try to complete everyone’s tasks quickly
  - As computers became cheaper, more important to optimize for user time, not computer time

Today’s Operating Systems: Computers Cheap

- Smartphones
- Embedded systems
- Laptops
- Tablets
- Virtual machines
- Data center servers

Tomorrow's Operating Systems

- Giant-scale data centers
- Increasing numbers of processors per computer
- Increasing numbers of computers per user
- Very large scale storage

Device I/O

- OS kernel needs to communicate with physical devices
  - Netowrk, disk, video, USB, keyboard, mouse, …
- Devices operate asynchronously from the CPU
  - Most have their own microprocessor
  - Ex> Apple Watch OS runs with a laptop keyboard!

Device I/O

- How does the OS communicate with the device?
  - I.O devices assigned a range of memory addresses or "ports"
  - Separate from main RAM
  - CPU instructions to command/read
    - Special I/O-specific instructions (inb/outb)
    - Read/write memory locations

Synchronous I/O

- Polling
  - I/O operations take time: 10^3 instructions to 10^8 instructions (physical limits)
  - OS pokes I/O memory/port to see if I/O is done
  - Device completes and stores data in device buffers
  - Kernel copies data from device into memory
  - Ugh
- Can we do better?

Faster I/O: Interrupts

- Interrupts: let device tell is when it is done
  - OS pokes I/O memory/port to issue request
  - CPU goes back to work on some other task
  - Device completes, stores data in its buffers
  - Triggers CPU interrupt to signal I/O completion
  - CPU copies data to/from device
  - When done, resume previous work
- Can we do better?

Faster I/O: DMA

- ”Programmed I/O”
  - I/O stored in the device
  - Requires CPU to do heavy lifting
  - Each instruction to move data is *uncached*, meaning direct transfers over the I/O bus
- Direct memory access (DMA)
  - Device *reads/writes RAM directly*
  - After I/O interrupt, CPU can access results in memory
- Can we do better?

Faster I/O: Buffer Descriptors

- Buffer descriptor: data structure to specify where to find the *next I/O request*
  - Buffer descriptor itself is DMA'd!
- CPU/Device share a queue of buffer descriptors
- Only interrupt if queue is empty or full

Device Interrupts

- How do device interrupts work?
  - How does the CPU know what code to run?
  - What language is the “interrupt handler” written in?
  - What stack does it use?
  - What about the work the CPU was doing when it was interrupted?
  - How does the CPU know how to resume that work?

Hardware Interrupt Vector

- Table set up by OS kernel
  - Pointers to functions
  - One per interrupt ”type”
  - Indexed and dispatched by the CPU; no software involvement
  - Likely needs a little assembly code help…

Challenge: Saving/Restoring State

- We need to transparently resume execution, e.g., the execution of the interrupt handler is invisible to (almost) all running code.
  - Many interrupts going on
    - I/O completion
    - Periodic timer to share CPU among multiple apps
  - Code must remain unaware that it was interrupted
- Not just instruction pointer
  - Registers
  - Floating point state
  - Condition codes.

Textbook

- Lazowska, Spring 2012: “The text is quite sophisticated. You won't get it all on the first pass. The right approach is to [read each chapter before class and] re-read each chapter once we've covered the corresponding material… more of it will make sense then. *Don't save this re-reading until right before the mid-term or final – keep up.”*