Today’s lecture

- Why study operating systems?
- What is an OS? What does an OS do?
- History of operating systems
- Principles of operating system design
- Course overview
  - course information
  - schedule, assignments, grading and policy
  - other organization issues
  - see web pages for more information
Why study operating systems?

- Understand how computers work under the hood
  - Magic for "infinite" CPUs, memory devices, network computing
  - Tradeoffs btw. performance & functionality, division of labor btw. HW & SW
  - Combine language, hardware, data structures, and algorithms

- Help you make informed decisions
  - What computer to buy? should I upgrade the HW or the OS?
  - What’s going on with my PC, especially when I have to install something?
  - Linux vs Mac OS X vs Windows 10 …, what should I bet on?

- Give you experience in hacking systems software
  “this system is so slow, can I do anything about it ?”

What’s interesting?

- OS is a key part of a computer system
  - it makes our life better (or worse)
  - it is “magical” and we want to understand how
  - it has “power” and we want to have the power

- OS is complex
  - how many procedures does a key stroke invoke?
  - real OS is huge and insanely expensive to build
    * Windows 8: many years, thousands of people. Still doesn’t work well

- How to deal with complexity?
  - decomposition into many layers of abstraction
  - fail early, fail fast, and learn how to make things work
What is an OS?

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

Operating System

Hardware

User-mode

System Library

File System

TCP/IP Networking

Virtual Memory

Scheduling

Hardware Specific Software and Device Drivers

Kernel-mode

System Library

Kernel-user Interface (Abstract virtual machine)

Hardware Abstraction Layer

Hardware

Disk

Processors

Address Translation

Graphics Processor

Network

System Library

APP

APP

APP
What is an OS?

**Android architecture & system stack**

What is an OS?

Visible software components of the **Linux desktop stack**
From http://en.wikipedia.org/wiki/Linux
**What is an OS?**

**Software stack for HPC clusters**


**What is an OS?**

Cloud computing

![Diagram of cloud computing and multi-user database systems]

**Multi-user database systems**

Other instances: multiplayer games, media players, social networking app, internet, …
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 deleted 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?

What does an OS do?

- OS converts bare HW into nicer abstraction
  - provide coordination: allow multiple applications/users to work together in efficient and fair way (memory protection, concurrency, file systems, networking)
  - provide standard libraries and services (program execution, I/O operations, file system manipulations, communications, resource allocation and accounting)

- For each OS area, you ask
  - what is the hardware interface --- the physical reality ?
  - what is the application interface (API) --- the nicer abstraction?
Example of OS coordination: protection

**Goal:** isolate bad programs and people (security)

**Solutions:**
- **CPU Preemption**
  * give application something, can always take it away (via clock interrupts)
- **Dual mode operation**
  * when in the OS, can do anything (kernel-mode)
  * when in a user program, restricted to only touching that program's memory (user-mode)
- **Interposition**
  * OS between application and "stuff"
  * track all pieces that application allowed to use (in a table)
  * on every access, look in table to check that access legal
- **Memory protection: address translation**

Example: address translation

*Restrict what a program can do by restricting what it can touch!*

◆ **Definitions:**
- Address space: all addresses a program can touch
- Virtual address: addresses in process' address space
- Physical address: address of real memory
- Translation: map virtual to physical addresses

◆ **Virtual memory**
- Translation done using per-process tables (page table)
- done on every load and store, so uses hardware for speed
- protection? If you don’t want process to touch a piece of physical memory, don’t put translation in table.
Computing & communications exponential growth!

- Performance/Price doubles every 18 months
- 100x per decade
- Progress in next 18 months = ALL previous progress
  - New storage = sum of all old storage (ever)
  - New processing = sum of all old processing.
- Aggregate bandwidth doubles in 8 months

15 years ago.

Computer performance over time

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

Figure 1.8: Approximate computer server performance over time, reflecting the most widely used servers of each era: in 1981, a minicomputer; in 1997, a high-end workstation; in 2014, a rack-mounted multicore server. MIPS stands for “millions of instructions per second,” a measure of processor performance. The VAX 11/782 was introduced in 1982; it achieved 1 MIP. DRAM prices are from Hennessey and Patterson, “Computer Architecture: A Quantitative Approach.” Disk drive prices are from John McCallum. The Hayes smartmodem, introduced in 1981, ran at 300bps. The 10 Mbps shared Ethernet standard was also introduced in 1981. One of the authors built his first operating system in 1982, used a VAX at his first job, and owned a Hayes to work from home.

From expensive to cheap devices occurred with telephones over the past hundred years. Initially, telephone lines were very expensive, and a single line was shared among everyone in a neighborhood. Over time, of course, both computers and telephones have become cheap enough to sit idle until we need them.

Despite these changes, operating systems still face the same conceptual challenges as they did fifty years ago. To manage computer resources for applications and users, they must allocate resources among applications, provide fault isolation and communication services, abstract hardware limitations, and so forth. Tremendous progress has been made towards improving the reliability, security, efficiency, and portability of operating systems, but much more is needed. Although we do not know precisely how computing technology or application demand will evolve over the next 10-20 years, it is highly likely that these fundamental operating system challenges will persist.

Early Operating Systems

Computers were expensive; users would wait. The first operating systems were runtime libraries intended to simplify the programming of early computer systems. Rather than the tiny, inexpensive yet massively complex hardware and software systems of today, the first computers often took up an entire floor of a warehouse, cost millions of
OS history

Challenges in writing OS

- Concurrent programming is hard
- Hard to use high-level programming languages
  - device drivers are inherently low-level
  - real-time requirement (garbage collection? probably not)
  - lack of debugging support (use simulation)
- Tension between functionality and performance
- Portability and backward compatibility
  - many APIs are already fixed (e.g., GUI, networking)
  - OS design tradeoffs change as HW changes!
Challenges in writing OS (cont’d)

- **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

Main techniques & design principles

- **Keep things simple!**

- **Use abstraction**
  - hide implementation complexity behind simple interface

- **Use modularity**
  - decompose system into isolated pieces

- **But what about performance**
  - find bottlenecks --- the 80-20 rule
  - use prediction and exploits locality (cache)

- **What about security and reliability?**

  *More research is necessary!*
Course information

Required textbook:


Information, assignments, & lecture notes are available on-line
we won’t use much paper

Official URL: http://flint.cs.yale.edu/cs422

For help, go to the piazza site:

https://piazza.com/yale/fall2015/cpsc422522

Course information (cont’d)

- **13 week lectures on OS fundamentals**
  - class participation is strongly recommended
- **Course requirements**
  - 70% on assignments (as1 - as6)
  - 25% open-book, in-class midterm (Tuesday, November 10th)
  - 5% class participation
  - Contrary to the OCI entry for 422, there is no final exam
- **Assignments (as1-as6) and course policies**
  - build a small but real OS kernel, bootable on real PCs.
  - extensive hacking (in C & x86 assembly) but highly rewarding
  - 2 persons / team (one person team is OK too).
  - 5 free late days (3 day late max per assignment).
Programming assignments

- Assignment topics (tentative)
  - Bootup & memory management
  - Virtual memory
  - Thread and process management
  - IPC and trap handling
  - Virtualization
  - File system

- How
  - Each assignment takes two weeks
  - All assignments due Tuesdays 11:59pm

- The Lab
  - Linux cluster in ZOO
  - You can setup your own machine to do projects

Programming assignments (cont’d)

Based on mCertiKOS (Yale FLINT) & JOS (from MIT)
Programming assignments (cont’d)

- Break kernel interdependency by insisting on careful layer decomposition
- With the right methodology, every CS major should be able to write an OS kernel from scratch