Introduction

CS 4410

Meet the OS

- Software that manages a computer’s resources
- Makes it easier to write the applications you want to write
- Makes you want to use the applications you wrote by running them efficiently

What is an OS?

- An Operating System implements a virtual machine whose interface is more convenient* than the raw hardware interface

OS wears many hats

- Referee
  - Manages shared resources: CPU, memory, disks, networks, displays, cameras...
- Illusionist
  - Look! Infinite memory! Your own private processor!
- Glue
  - Offers a set of common services (e.g., UI routines)
  - Separates apps from I/O devices

* easier to use, simpler to code, more reliable, more secure...

“All the code you did not write”
**OS as Referee**

Resource allocation
- Multiple concurrent tasks, how does OS decide who gets how much?

Isolation
- A faulty app should not disrupt other apps or OS

Communication/Coordination
- Apps need to coordinate and share state

**OS as Illusionist**

Virtualization
- Processor, memory, screen space, disk, network
- The entire computer
  - Fooling the illusionist itself!
  - Eases debugging, portability, isolation

Appearance of resources not physically present

Atomic operations
- HW guarantees atomicity at the word level...
  - What happens during concurrent updates to complex data structures?
  - What is a computer crashes while writing a file block?
- At the hardware level, packets are lost
  - Reliable communication channels
OS as Glue

- Offers standard services to simplify app design and facilitate sharing
  - Send/Receive byte streams
  - Read/Write files
  - Pass messages
  - Share memory
  - UI
- Decouples HW and app development

A Short History of Operating Systems

History of Operating Systems

- Phase 1: Hardware is expensive, humans are cheap
  - User at console: single-user systems
  - Batching systems
  - Multi-programming systems

Hand programmed machines (1945-1955)

- Single user systems
- OS = loader + libraries of common subroutines
- Problem: low utilization of expensive components

\[
\text{utilization} = \frac{\text{time device busy}}{\text{observation interval}} \times 100\%
\]
**Batch Processing (1955-1965)**

Operating system = loader + sequencer + output processor

**Multiprogramming (1965-1980)**

Keep several jobs in memory and multiplex CPU between jobs

```
program P
begin
  Read(var)
end P
```

System call Read()
begin
  StartIO(input device)
  WaitIO(interrupt)
  EndIO(input device)
end Read
History of Operating Systems

- Phase 1: Hardware is expensive, humans are cheap
  - User at console: single-user systems
  - Batching systems
  - Multi-programming systems
- Phase 2: Hardware is cheap, humans are expensive
  - Time sharing: Users use cheap terminals and share servers
- Phase 3: Hardware is very cheap, humans are very expensive
  - Personal computing: One system per user
  - Distributed computing: many systems per user
  - Ubiquitous computing: LOTS of systems per users

Timesharing (1970-)

A timer interrupt is used to multiplex CPU between jobs

Operating Systems for PCs

- Personal computing systems
  - Single user
  - Utilization no longer a concern
  - Emphasis on user interface and API
  - Many services & features not present

Evolution

- Initially: OS as a simple service provider (simple libraries)
- Now: Multi-application systems with support for coordination
Why study Operating Systems?

- To learn how to manage complexity through appropriate abstractions
  - infinite CPU, infinite memory, files, locks, etc.
- To learn about design
  - performance vs. robustness, functionality vs. simplicity, HW vs. SW, etc.
- To learn how computers work
- Because OSs are everywhere!

Where’s the OS?
Las Vegas
Where's the OS?
New York

What will the course be like?

What kind of course?

- Top-down...
  - Start from first principles
  - Re-derive the design of components of a complex system

- ... & Bottom-up
  - Dissect existing systems, to learn
    - what tradeoffs they make
    - what patterns they use
System building is hard!

Painting

- Order
- Design
- Tension
- Balance
- Harmony

Sondheim: Sunday in the Park with George
Therac-25 [1982]

- Computer-controlled radiation therapy machine
- Safety critical system with software interlocks
  - Prevent machine from changing state because of state of another element
- Beam controlled entirely through a custom OS

Therac-25

- Old system used a hardware interlock
  - Lever either in the “electron-beam” or “x-ray” position
- New system was computer controlled.
  - A synchronization failure triggered when competent nurses used back arrow to change the data on the screen “too quickly”
  - Engineers reused software from older models
    - It was buggy, but hardware interlocks masked the bugs
  - The system noted a problem and halted X-beam, displaying “MALFUNCTION” followed by obscure error code 54
    - Technician resumed treatment

Therac-25 Outcome

- Patients received over 100x the recommended dose of radiation
- Three patients died of radiation overdose
- Many cancer patients received inadequate treatment
- People died because a programmer could not write correct code for a concurrent system
- 37 Year Later.... Now what?

System building is hard

- We do not have the necessary technologies and know-how to build robust computer systems
- The world is increasingly dependent on computer systems
  - Connected, networked, interlinked
- There is huge demand for people who deeply understand and can build robust systems (most people don’t and can’t)
What's this course about?

Ostensibly, operating systems
- Architecting complex software
- Identifying needs and priorities
- Separating concerns
- Implementing artifacts with desired properties

In reality, software design principles
- OSes happen to illustrate organizational principles and design patterns

This is a Capstone Course. Get Ready!

What makes a good OS?

The right set of abstractions

A good abstraction:
- is portable and hides implementation details
- has an intuitive and easy-to-use interface
- can be installed many times
- is efficient and reasonably easy to implement

OS: a collection of abstractions

- Processes (abstract CPU and RAM)
- Files (abstract disks)
- Network endpoints (abstract NIC)
- Windows (abstract screens)
- ...

Think of them as objects with state and methods

Issues in OS Design

- Structure: how is the OS organized?
- Concurrency: how are parallel activities created and controlled?
- Sharing: how are resources shared?
- Naming: how are resources named by users?
- Protection: how are distrusting parties protected from each other?
- Security: how to authenticate, authorize, and ensure privacy?
- Performance: how to make it fast?
More Issues in OS Design

- **Reliability**: how do we deal with failures??
- **Portability**: how to write once, run anywhere?
- **Extensibility**: how do we add new features?
- **Communication**: how do we exchange information?
- **Scale**: what happens as demands increase?
- **Persistence**: how do we make information outlast the processes that created it?
- **Accounting**: who pays the bill and how do we control resource usage?