Introduction
CS 111
Operating System Principles

Peter Reiher

Outline

- Administrative materials
- Introduction to the course
  - Why study operating systems?
  - Basics of operating systems

Administrative Issues

- Instructor and TAs
- Load and prerequisites
- Web site, syllabus, reading, and lectures
- Quizzes, exams, homework, projects
- Grading
- Academic honesty

Instructor: Peter Reiher

- UCLA Computer Science department faculty member
- Long history of research in operating systems
- Email: reiher@cs.ucla.edu
- Office: 3532F Boelter Hall
  – Office hours: TTh 1-2
  – Often available at other times

My OS Background

- My Ph.D. dissertation was on the Locus operating system
- Much research on file systems
  - Ficus, Rumor, Truffles, Conquest
- Research on OS security issues
  - Data Tethers

TAs

- Muhammad Mehdi
  - taqi@cs.ucla.edu
- Diyu Zhou
  - zhoudiyu@cs.ucla.edu
- Zhaoxing Bu
  - zbu@cs.ucla.edu
- Jungbeom Lee
  - jungbeol@g.ucla.edu
- Lab sessions:
  - Lab 1A, Fridays 10-12 AM, Boelter 9436
  - Lab 1B, Fridays 12- 2 PM, Humants 169
  - Lab 1C, Fridays 2-4 PM, Rolfe 3134
  - Lab 1D, Fridays 4-6 PM, Bunche 2160
- Office hours to be announced

Instructor/TA Division of Responsibilities

- Instructor handles all lectures, readings, and tests
  - Ask me about issues related to these
- TAs handle projects
  - Ask them about issues related to these
- Generally, instructor won't be involved with project issues
  - So direct those questions to the TAs

Web Site

- Some materials found on CCLE web site
  - Like quizzes
- Most materials here:
  - http://www.lasr.cs.ucla.edu/classes/111_fall16
- What's there:
  - Schedules for reading, lectures, exams, projects
  - Copies of lecture slides (Powerpoint and PDF)
  - Announcements
  - Sample midterm and final problems

Prerequisite Subject Knowledge

- CS 32 programming
  - Objects, data structures, queues, stacks, tables, trees
- CS 33 systems programming
  - Assembly language, registers, memory
  - Linkage conventions, stack frames, register saving
- CS 35L Software Construction Laboratory
  - Useful software tools for systems programming
- If you haven’t taken these classes, expect to have a hard time in 111

Course Format

- Two weekly reading assignments
  - Mostly from the primary text
  - Some supplementary materials available on web
- Two weekly lectures
  - With online quizzes before the lectures
- Four (10-25 hour) individual projects
  - Exploring and exploiting OS features
  - Plus one warm-up project
- A midterm and a final exam

Course Load

- Reputation: THE hardest undergrad CS class
  - Fast pace through much non-trivial material
- Expectations you should have
  - lectures 4-6 hours/week
  - reading 3-6 hours/week
  - projects 3-20 hours/week
  - exam study 5-15 hours (twice)
- Keeping up (week by week) is critical
  - Catching up is extremely difficult

Primary Text for Course

- Remzi and Andrea Arpaci-Dusseau: *Operating Systems: Three Easy Pieces*
  - Freely available on line at http://pages.cs.wisc.edu/~remzi/OSTEP/
- Supplementary readings from Saltzer and Kaashoek: *Principles of Computer Systems Design*
  - Free on line at
    http://www.sciencedirect.com/science/book/9780123749574
    - Only on-campus or through the UCLA VPN
- Supplemented with web-based materials

Course Grading

- Basis for grading:
  - Quizzes 5%
  - 1 midterm exam 20%
  - Final exam 30%
  - Lab 0 5%
  - Other labs 10% each
- I do look at distribution for final grades
  - But don't use a formal curve
- All scores available on MyUCLA
  - Please check them for accuracy

Quizzes

- 3-5 question quizzes on the assigned reading materials
- Must be taken before the lecture
- Not intended to be hard questions
  - IF you've read the assigned materials

Midterm Examination

- When: Second lecture of the 5th week (in class section, Tuesday, October 25)
- Scope: All lectures up to the exam date
  - Approximately 60% lecture, 40% text
- Format:
  - Closed book
  - 10-15 essay questions, most with short answers
- Goals:
  - Test understanding of key concepts
  - Test ability to apply principles to practical problems

Final Exam

- When: Monday, December 5, 11:30 AM-2:30 PM
- Scope: Entire course
- Format:
  - 6-8 hard multi-part essay questions
  - You get to pick a subset of them to answer
- Goals:
  - Test mastery of key concepts
  - Test ability to apply key concepts to real problems
  - Use key concepts to gain insight into new problems

Lab Projects

- Format:
  - 1 warm-up project
  - 4 regular projects
  - Done individually
- Goals:
  - Develop ability to exploit OS features
  - Develop programming/problem solving ability
  - Practice software project skills
- Lab and lecture are fairly distinct
  - Ask instructor about tests and lectures
  - Ask TAs about labs

Late Assignments & Make-ups

- Quizzes
  - No late quizzes accepted, no make-ups
- Labs
  - Due dates set by TAs
  - TAs also sets policy on late assignments
  - The TAs will handle all issues related to labs
    - Ask them, not me
    - Don't expect me to overrule their decisions
- Exams
  - Alternate times or make-ups only possible with prior consent of the instructor
  - If you miss a test, too bad

Academic Honesty

- It is OK to study with friends
  - Discussing problems helps you to understand them
- It is OK to do independent research on a subject
  - There are many excellent treatments out there
- But all work you submit must be your own
  - Do not write your lab answers with a friend
  - Do not copy another student's work
  - Do not turn in solutions from off the web
  - If you do research on a problem, cite your sources
- I decide when two assignments are too similar
  - And I forward them immediately to the Dean
- If you need help, ask the instructor

Academic Honesty – Projects

- Do your own projects
  - If you need additional help, ask the TA
- You must design and write <u>all</u> your own code
  - Do not ask others how they solved the problem
  - Do not copy solutions from the web, files or listings
  - Cite any research sources you use
- Protect yourself
  - Do not show other people your solutions
  - Be careful with old listings

Academic Honesty and the Internet

- You might be able to find existing answers to some of the assignments on line
- Remember, if you can find it, so can we
  - And we have, before
- It IS NOT OK to copy the answers from other people's old assignments
  - People who tried that have been caught and referred to the Office of the Dean of Students
- ANYTHING you get off the Internet must be treated as reference material
  - If you use it, quote it and reference it

Introduction to the Course

- Purpose of course and relationships to other courses
- Why study operating systems?
- Major themes & lessons in this course

What Will CS 111 Do?

- Build on concepts from other courses
  - Data structures, programming languages, assembly language programming, computer architectures, ...
- Prepare you for advanced courses
  - Data bases and distributed computing
  - Security, fault-tolerance, high availability
  - Network protocols, computer system modeling, queueing theory
- Provide you with foundation concepts
  - Processes, threads, virtual address space, files
  - Capabilities, synchronization, leases, deadlock

Why Study Operating Systems?

- Few of you will actually build OSs
- But many of you will:
  - Set up, configure, manage computer systems
  - Write programs that exploit OS features
  - Work with complex, distributed, parallel software
  - Work with abstracted services and resources
- Many hard problems have been solved in OS context
  - Synchronization, security, integrity, protocols, distributed computing, dynamic resource management, ...
  - In this class, we study these problems and their solutions
  - These approaches can be applied to other areas

Why Are Operating Systems Interesting?

- They are extremely complex
  - But try to appear simple enough for everyone to use
- They are very demanding
  - They require vision, imagination, and insight
  - They must have elegance and generality
  - They demand meticulous attention to detail
- They are held to very high standards
  - Performance, correctness, robustness,
  - Scalability, extensibility, reusability
- They are the base we all work from

Recurring OS Themes

- View services as objects and operations
  - Behind every object there is a data structure
- Separate policy from mechanism
  - Policy determines what can/should be done
  - Mechanism implements basic operations to do it
  - Mechanisms shouldn't dictate or limit policies
  - Policies must be changeable without changing mechanisms
- Parallelism and asynchrony are powerful and vital
  - But dangerous when used carelessly
- Performance and correctness are often at odds

More Recurring Themes

- An interface specification is a contract
  - Specifies responsibilities of producers & consumers
  - Basis for product/release interoperability
- Interface vs. implementation
  - An implementation is not a specification
  - Many compliant implementations are possible
  - Inappropriate dependencies cause problems
- Modularity and functional encapsulation
  - Complexity hiding and appropriate abstraction

Life Lessons From Studying Operating Systems

- There Ain't No Such Thing As A Free Lunch! (TANSTAAFL)
  - Everything has a cost, there are always trade-offs
  - But there are bad, expensive lunches . . .
- Keep It Simple, Stupid!
  - Avoid complex solutions, and being overly clever
  - Both usually create more problems than they solve
- Be very clear what your goals are
  - Make the right trade-offs, focus on the right problems
- Responsible and sustainable living
  - Understand the consequences of your actions
  - Nothing must be lost, everything must be recycled
  - It is all in the details

Moving on To Operating Systems . . .

- What is an operating system?
- What does an OS do?
- How does an OS appear to its clients?
  - Abstracted resources
    - Simplifying, generalizing
    - Serially reusable, partitioned, sharable

What Is An Operating System?

- Many possible definitions
- One is:
  - It is low level software . . .
  - That provides better, more usable abstractions of the hardware below it
  - To allow easy, safe, fair use and sharing of those resources

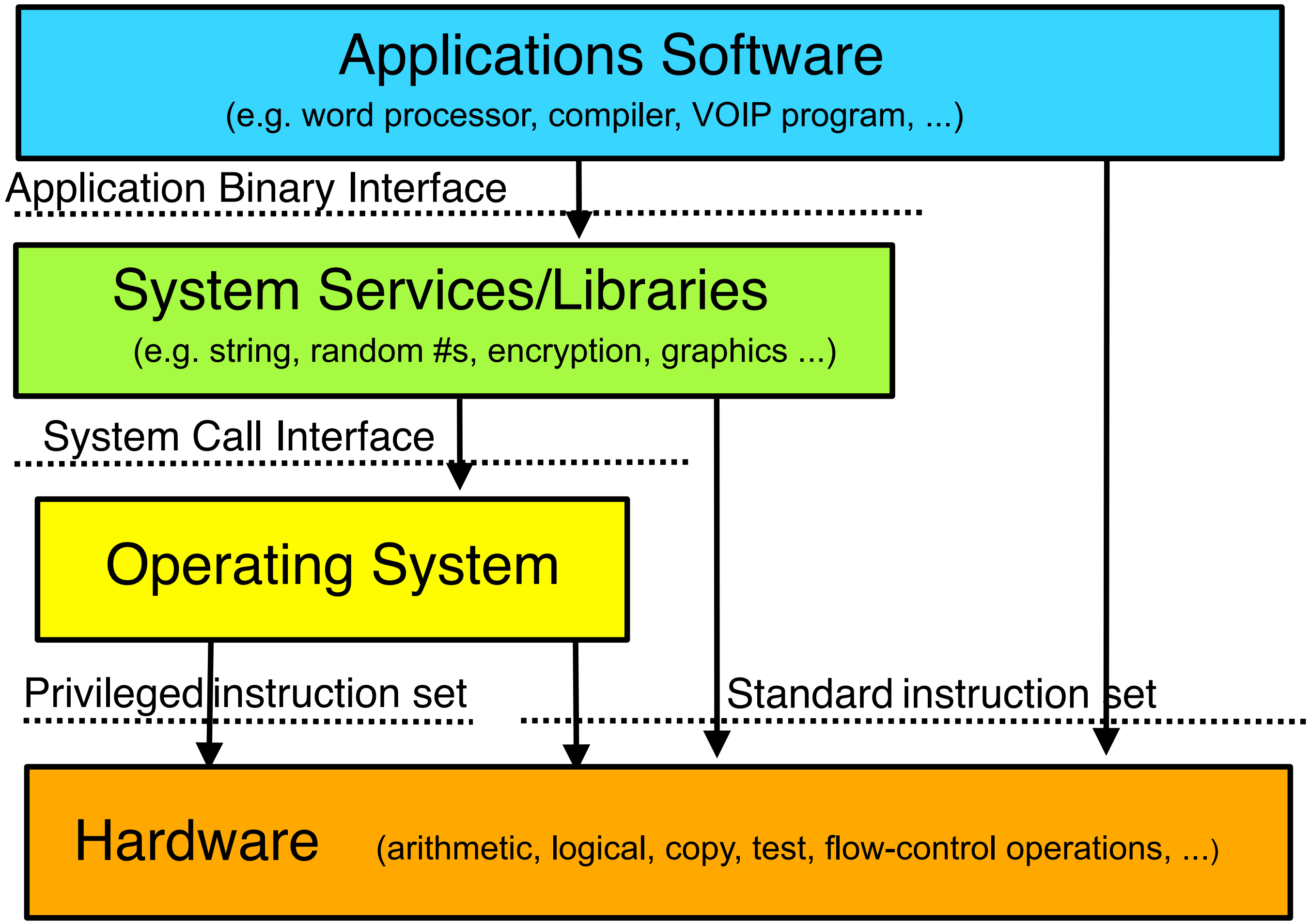
What Does an OS Do?

- It manages hardware for programs
  - Allocates hardware and manages its use
  - Enforces controlled sharing (and privacy)
  - Oversees execution and handles problems
- It abstracts the hardware
  - Makes it easier to use and improves SW portability
  - Optimizes performance
- It provides new abstractions for applications
  - Powerful features beyond the bare hardware

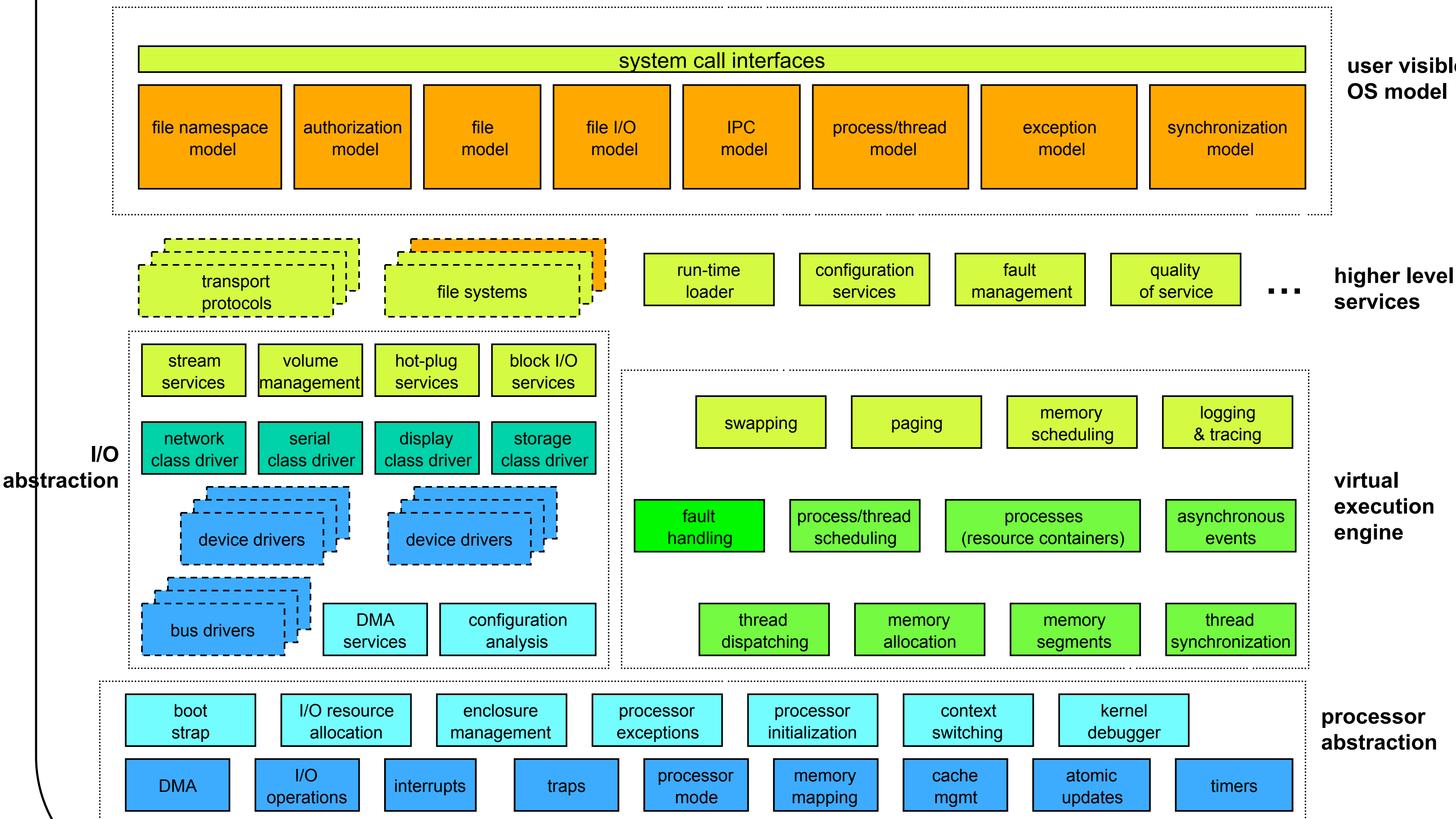
What Does An OS Look Like?

- A set of management & abstraction services
  - Invisible, they happen behind the scenes
- Applications see objects and their services
  - CPU supports data-types and operations
    - bytes, shorts, longs, floats, pointers, ...
    - add, subtract, copy, compare, indirection, ...
  - So does an operating system, but at a higher level
    - files, processes, threads, devices, ports, ...
    - create, destroy, read, write, signal, ...
- An OS extends a computer
  - Creating a much richer virtual computing platform
    - Supporting richer objects, more powerful operations

Where Does the OS Fit In?



Kernel Structure (artists conception)



Instruction Set Architectures (ISAs)

- The set of instructions supported by a computer
  - What bit patterns correspond to what operations
- There are many different ISAs (all incompatible)
  - Different word/bus widths (8, 16, 32, 64 bit)
  - Different features (low power, DSPs, floating point)
  - Different design philosophies (RISC vs. CISC)
  - Competitive reasons (68000, x86, PowerPC)
- They usually come in families
  - Newer models add features (e.g., Pentium vs. 386)
  - But remain upwards-compatible with older models
    - A program written for an ISA will run on any compliant CPU

Privileged vs. General Instructions

- Most modern ISAs divide instruction set into privileged vs. general
- Any code running on the machine can execute general instructions
- Processor must be put into a special mode to execute privileged instructions
  - Usually only in that mode when the OS is running
  - Privileged instructions do things that are "dangerous"

Platforms

- ISA doesn't completely define a computer
  - Functionality beyond user mode instructions
    - Interrupt controllers, DMA controllers
    - Memory management unit, I/O busses
    - BIOS, configuration, diagnostic features
    - Multi-processor & interconnect support
  - I/O devices
    - Display, disk, network, serial device controllers
- These variations are called "platforms"
  - The platform on which the OS must run

Portability to Multiple ISAs

- A successful OS will run on many ISAs
  - Some customers cannot choose their ISA
  - If you don't support it, you can't sell to them
- Minimal assumptions about specific HW
  - General frameworks are HW independent
    - File systems, protocols, processes, etc.
  - HW assumptions isolated to specific modules
    - Context switching, I/O, memory management
  - Careful use of types
    - Word length, sign extension, byte order, alignment

Distributing an OS

- Developers want their OS to run on as many machines as possible
- There are many different ISAs
  - And other platform differences
- Even more types of peripherals
- And vast numbers of different applications and configurations
- How to get wide, effective distribution?

Binary Distribution Model

- Binary is the opposite of source
  - A source distribution must be compiled
  - A binary distribution is ready to run
- OSes usually distributed in binary
- One binary distribution per ISA
  - No need for special per-OEM OS versions
- Binary model for platform support
  - Device drivers can be added, after-market
    - Can be written and distributed by 3rd parties
    - Same driver works with many versions of OS

Why Not a Source Distribution?

- What is wrong with distributing an OS in source form?
  1. *On what are you going to compile it?*
  2. *Are your customers competent to build the OS? Do they want to build the OS?*
  3. *Do you really want to give all of your customers the sources to your main product?*
- Open source OSes are available
  – But most users still download the binary versions

Binary Configuration Model

- Good to eliminate manual/static configuration
  - Enable one distribution to serve all users
  - Improve both ease of use and performance
- Automatic hardware discovery
  - Self-identifying busses
    - PCI, USB, PCMCIA, EISA, etc.
  - Automatically find and load required drivers
- Automatic resource allocation
  - Eliminate fixed sized resource pools
  - Dynamically (re)allocate resources on demand

Flexibility

- Different customers have different needs
- We cannot anticipate all possible needs
- We must design for flexibility/extension
  - Mechanism/policy separation
    - Allow customers to override default policies
    - Changing policies without having to change the OS
  - Dynamically loadable features
    - Allow new features to be added, after market
    - File systems, protocols, load module formats, etc.
  - Feature independence and orthogonality

Interface Stability

- People want new releases of an OS
  - New features, bug fixes, enhancements
- People also fear new releases of an OS
  - OS changes can break old applications
- How can we prevent such problems?
  - Define well specified Application Interfaces
    - Application programming interfaces (APIs)
    - Application binary interfaces (ABIs)
  - Applications only use committed interfaces
  - OS vendors preserve upwards-compatibility

What's Special About the OS?

- It is always in control of the hardware
  - Automatically loaded when the machine boots
  - First software to have access to hardware
  - Continues running while apps come & go
- It alone has complete access to hardware
  - Privileged instruction set, all of memory & I/O
- It mediates applications' access to hardware
  - Block, permit, or modify application requests
- It is trusted
  - To store and manage critical data
  - To always act in good faith
- If the OS crashes, it takes everything else with it
  - So it better not crash . . .

What Functionality Is In the OS?

- As much as necessary, as little as possible
  - OS code is very expensive to develop and maintain
- Functionality must be in the OS if it ...
  - Requires the use of privileged instructions
  - Requires the manipulation of OS data structures
  - Must maintain security, trust, or resource integrity
- Functions should be in libraries if they ...
  - Are a service commonly needed by applications
  - Do not actually have to be implemented inside OS
- But there is also the performance excuse
  - Some things may be faster if done in the OS

Where To Offer a Service?

- Hardware, OS, library or application?
- Increasing requirements for stability as you move through these options
- Hardware services rarely change
- OS services can change, but it's a big deal
- Libraries are a bit more dynamic
- Applications can change services much more readily

Another Reason For This Choice

- Who uses it?
- Things literally everyone uses belong lower in the hierarchy
  - Particularly if the same service needs to work the same for everyone
- Things used by fewer/more specialized parties belong higher
  - Particularly if each party requires a substantially different version of the service

The OS and Speed

- One reason operating systems get big is based on speed
- It's faster to offer a service in the OS than outside it
- Thus, there's a push to move services with strong performance requirements down to the OS

Why Is the OS Faster?

- Than something at the application level, above it?
  - If it involves processes communicating, working at app level requires scheduling and swapping them
  - The OS has direct access to many pieces of state and system services
    - If an operation requires such things, application has to pay the cost to enter and leave OS, anyway
  - The OS can make direct use of privileged instructions

Is An OS Implementation Always Faster?

- Not always
- Running standard instructions no faster from the OS than from applications
- Entering the OS involves some fairly elaborate state saving and mode changing
- If you don't need special OS services, may be cheaper to manipulate at the app level
  - Maybe by an order of magnitude

The OS and Abstraction

- One major function of an OS is to offer abstract versions of resources
  - As opposed to actual physical resources
- Essentially, the OS implements the abstract resources using the physical resources
  - E.g., processes (an abstraction) are implemented using the CPU and RAM (physical resources)
  - And files (an abstraction) are implemented using disks (a physical resource)

Why Abstract Resources?

- The abstractions are typically simpler and better suited for programmers and users
  - Easier to use than the original resources
    - E.g., don't need to worry about keeping track of disk interrupts
  - Compartmentalize/encapsulate complexity
    - E.g., need not be concerned about what other executing code is doing and how to stay out of its way
  - Eliminate behavior that is irrelevant to user
    - E.g., hide the slow erase cycle of flash memory
  - Create more convenient behavior
    - E.g., make it look like you have the network interface entirely for your own use

Generalizing Abstractions

- Lots of variations in machines' HW and SW
- So make many different types appear to be same
  - So applications can deal with single common class
- Usually involves a common unifying model
  - E.g., portable document format (pdf) for printers
  - Or SCSI standard for disks, CDs and tapes
- Usually involves a <u>federation framework</u>
  - Per sub-type implementations of standard functions
- For example:
  - Printer drivers make different printers look the same
  - Browser plug-ins to handle multi-media data

Why Do We Want This Generality?

- For example, why do we want all printers to look the same?
  - So we could write applications against a single model, and have it "just work" with all printers
- What's the alternative?
  - Program our application to know about all possible printers
  - Including those that were invented after we had written our application!

Does a General Model Limit Us?

- Does it stick us with the "least common denominator" of a hardware type?
  - Like limiting us to the least-featureful of all printers?
- Not necessarily
  - The model can include "optional features"
    - If present, implemented in a standard way
    - If not present, test for them and do "something" if they're not there
- Many devices will have features not in the common model
  - There are arguments for and against the value of such features

Common Types of OS Resources

- Serially reusable resources
- Partitionable resources
- Sharable resources

Serially Reusable Resources

- Used by multiple clients, but only one at a time
  - Time multiplexing
- Require access control to ensure exclusive use
- Require graceful transitions from one user to the next
- Examples: printers, bathroom stalls

What Is A Graceful Transition?

- A switch that totally hides the fact that the resource used to belong to someone else
  - Don't allow the second user to access the resource until the first user is finished with it
    - No incomplete operations that finish after the transition
  - Ensure that each subsequent user finds the resource in "like new" condition
    - No traces of data or state left over from the first user

Partitionable Resources

- Divided into disjoint pieces for multiple clients
  - Spatial multiplexing
- Needs access control to ensure:
  - Containment: *you cannot access resources outside of your partition*
  - Privacy: *nobody else can access resources in your partition*
- Examples: RAM, hotel rooms

Do We Still Need Graceful Transitions?

- Yes
- Most partitionable resources aren't permanently allocated
  - The page frame of RAM you're using now will belong to another process later
- As long as it's "yours," no transition required
- But sooner or later it's likely to become someone else's

Shareable Resources

- Usable by multiple concurrent clients
  - Clients do not have to “wait” for access to resource
  - Clients don’t “own” a particular subset of resource
- May involve (effectively) limitless resources
  - Air in a room, shared by occupants
  - Copy of the operating system, shared by processes
- May involve under-the-covers multiplexing
  - Cell-phone channel (time and frequency multiplexed)
  - Shared network interface (time multiplexed)

Do We Still Need Graceful Transitions?

- Typically not
- The shareable resource usually doesn't change state
- Or isn't "reused"
- We never have to clean up what doesn't get dirty
  - Like an execute-only copy of the OS
- Or things that disappear after use
  - Like the previous second's bandwidth on a cellular telephone channel
- Shareable resources are great!
  - When you can have them . . .

A Brief History of Operating Systems

- 1950s … OS? We don't need no stinking OS!
- 1960s batch processing
  - Job sequencing, memory allocation, I/O services
- 1970s time sharing
  - Multi-user, interactive service, file systems
- 1980s work stations and personal computers
  - Graphical user interfaces, productivity tools
- 1990s work groups and the world wide web
  - Shared data, standard protocols, domain services
- 2000 large scale distributed systems
  - The network IS the computer

A Certain Irony

- Today's smart phone is immensely more powerful than 1960s mainframes
- But we used the mainframes for the biggest computing tasks we had
- While we use our powerful smart phones to move information around and display stuff
- Which has implications for their operating systems . . .

General OS Trends

- They have grown larger and more sophisticated
- Their role has fundamentally changed
  - From shepherding the use of the hardware
  - To shielding the applications from the hardware
  - To providing powerful application computing platform
  - To becoming a sophisticated "traffic cop"
- They still sit between applications and hardware
- Best understood through services they provide
  - Capabilities they add
  - Applications they enable
  - Problems they eliminate

Why?

- Ultimately because it's what users want
- The OS must provide core services to applications
- Applications have become more complex
  - More complex internal behavior
  - More complex interfaces
  - More interactions with other software
- The OS needs to help with all that complexity

OS Convergence

- There are a handful of widely used OSes
  - And a few special purpose ones (E.g., real time and embedded system OSes)
- New ones come along very rarely
- OSes in the same family (e.g., Windows or Linux) are used for vastly different purposes
  - Making things challenging for the OS designer
- Most OSes are based on pretty old models
  - Linux comes from Unix (1970s vintage)
  - Windows from the early 1980s

Operating Systems for Mobile Devices

- What's down at the bottom for our smart phones and other devices?
- For Apple devices, ultimately XNU
  - Based on Mach (an 80s system), with some features from other 80s systems (like BSD Unix)
- For Android, ultimately Linux
- For Microsoft, ultimately Windows CE
  - Which has its origins in the 1990s
- None of these is all that new, either

Why Have OSes Converged?

- They're expensive to build and maintain
  - So it's a hard business to get into and stay in
- They only succeed if users choose them over other OS options
  - Which can't happen unless you support all the apps the users want (and preferably better)
  - Which requires other parties to do a lot of work
- You need to have some clear advantage over present acceptable alternatives

A Resulting OS Challenge

- We are basing the OS we use today on an architecture designed 20-40 years ago
- We can make some changes in the architecture
- But not too many
  - Due to compatibility
  - And fundamental characteristics of the architecture
- Requires OS designers and builders to shoehorn what's needed today into what made sense yesterday

Conclusion

- Understanding operating systems is critical to understanding how computers work
- Operating systems interact directly with the hardware
- Operating systems provide services via abstractions
- Operating systems are constrained by many non-technical factors