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
- 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, recently

TAs

- Tuan Le
  - tuanle@cs.ucla.edu
- Muhammad Mehdi
  - taqi@cs.ucla.edu
- Guanya Yang
  - guayang@g.ucla.edu
- Lab sessions:
  - Lab 1A, Fridays 8-10 AM, Boelter 9436
  - Lab 1B, Fridays 10 AM - 12 PM, Boelter 9436
  - Lab 1C, Fridays 10 AM - 12 PM, Boelter 5272
- 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

- http://www.lasr.cs.ucla.edu/classes/111_fall15
- What's there:
  - Schedules for reading, lectures, exams, projects
  - Copies of lecture slides (Powerpoint)
  - 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 (average 20 page) reading assignments
  - Mostly from the primary text
  - A few supplementary articles available on web
- Two weekly lectures
- Four (10-25 hour) team projects
  - Exploring and exploiting OS features
- One design project (10-25 hours)
  - Working off one of the team projects

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

- Saltzer and Kaashoek: *Principles of Computer Systems Design*
  - Background reading for most lectures
  - Available on line (for free) at
    http://www.sciencedirect.com/science/book/9780123749574
    - Probably only on-campus or through the UCLA VPN
- Supplemented with web-based materials

Course Grading

- Basis for grading:
  - 1 midterm exam 25%
  - Final exam 30%
  - Projects 45%
- 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

Midterm Examination

- When: Second lecture of the 5th week (in class section)
- 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: Friday, December11, 3-6 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:
  - 4 regular projects
  - 2 mini-projects
  - May be done solo or in teams (of two)
- Goals:
  - Develop ability to exploit OS features
  - Develop programming/problem solving ability
  - Practice software project skills
- Lab and lecture are fairly distinct
  - Instructor cannot help you with projects
  - TAs can't help with lectures, exams

Design Problems

- Each lab project contains suggestions for extensions
- Each student is assigned one design project from among the labs
  - Individual or two person team
- Requires more creativity than labs
  - Usually requires some coding
- Handled by the TAs

Late Assignments & 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

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 <u>write</u> your lab answers with a friend
  - Do not <u>copy</u> another student's work
  - Do not turn in solutions <u>from off the web</u>
  - If you do research on a problem, <u>cite your sources</u>
- 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
  - Work only with your team-mate
  - If you need additional help, ask the TA
- You must design and write <u>all</u> your own code
  - Other than cooperative work with your team-mate
  - 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
- 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 necessary
  - 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
- A brief history of operating systems

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

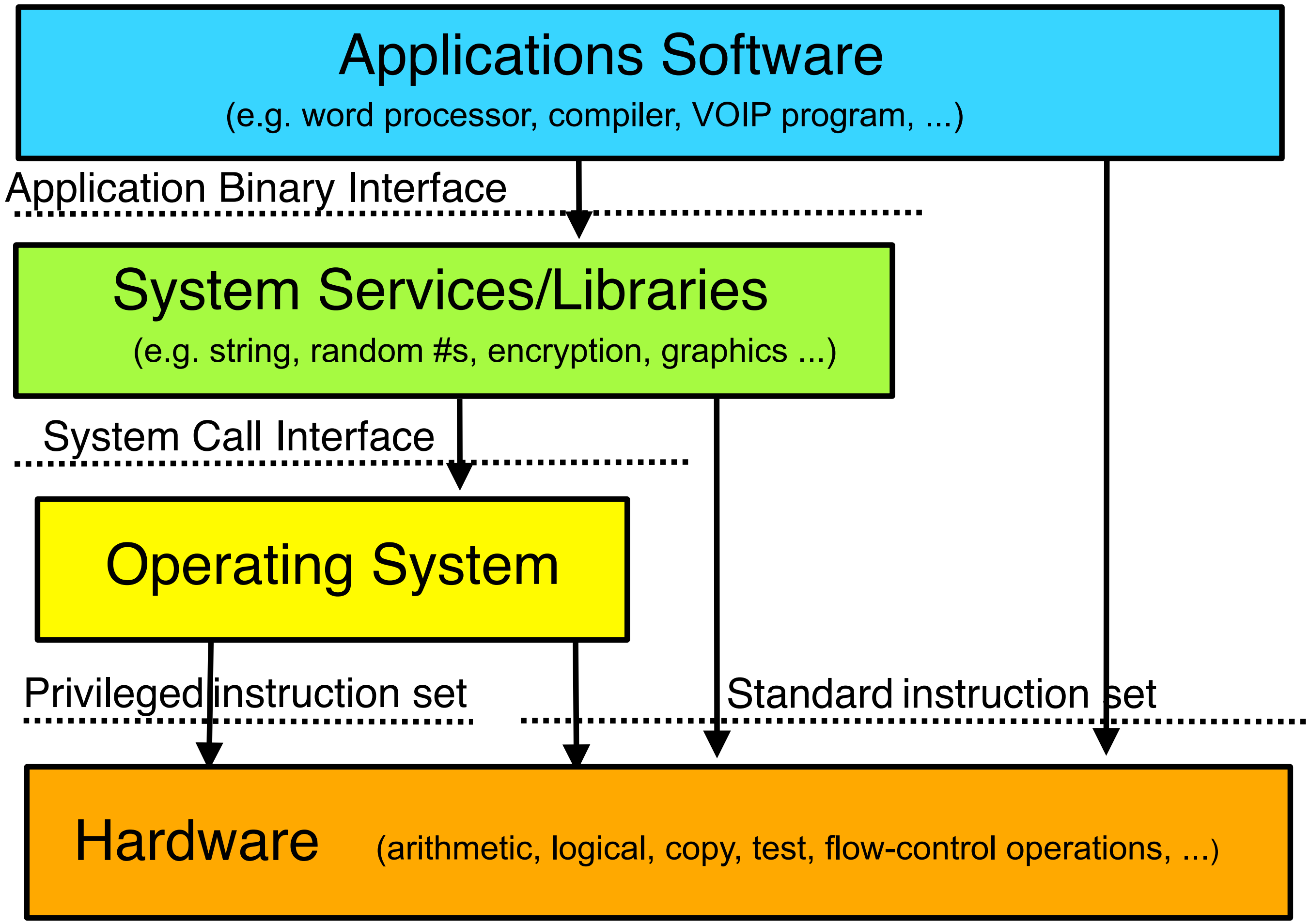
What Does an OS Do?

- 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 s/w 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?



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 <u>complete access</u> 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 sectors and tracks of the disk
  - Create more convenient behavior
    - E.g., make it look like you have the network interface entirely for your own use

Generalizing Abstractions

- 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: disk space, hotel rooms

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 <u>under-the-covers</u> multiplexing
  - Cell-phone channel (time and frequency multiplexed)
  - Shared network interface (time multiplexed)

A Brief History of the Evolution of Operating Systems

- Early computers
- Batch processing
- Time sharing
- Work stations, PCs
- Embedded systems
- Client/server computing

Early Computers (1940s-1950s)

- Usage
  - Scheduled for use by one user at a time
- Input
  - Paper cards, paper tape, magnetic tape, dip switches
- Output
  - Paper cards, paper tape, print-outs, magnetic tape, lights
- Software
  - Compilers, assemblers, math packages
  - No “resident” operating system
  - Typically one program resident at a time
- Debugging
  - In binary, via lights and switches

Batch Computing (1960s)

- Typified by the IBM System/360 (mid 1960s)
  - Programs submitted and picked up later
  - Input and output spooling to tape and disk
- Goals: efficient CPU use, maximize throughput
  - Computer was an expensive resource to be shared
  - I/O able to proceed with minimal CPU
  - Overlapped execution and I/O maximize CPU usage
  - Limited multi-tasking ability to minimize idle time
- Software
  - Batch monitor … to move from one job to the next
  - I/O supervisor … to manage background I/O
- Debugging (in hex or octal via paper core dumps)
  - Long analysis cycle between test runs

Time Sharing (1970s)

- Typified by IBM/CMS, Multics, UNIX
  - Multi-user, interaction through terminals
  - All programs and data stored on disk
- Goals: sharing for interactive users
  - Interactive apps demand short response time
  - Enhanced security required to ensure privacy
- OS and system services expanded greatly
  - Terminal I/O, synchronization, inter-process communication, networking, protection, etc.

How Do Batch and Multitasking Differ?

1. No interaction between tasks in a batch system
   - Each thinks it has the whole computer to itself
   - Parallel tasks in a timesharing system can interact
2. A timesharing system wants to provide good interactive response time to every task
   - Which probably means preemptive scheduling
   - Batch systems run each job to completion
     - Queueing theory tells us this can greatly increase average response time
     - But gives us great utilization of the CPU

Workstations and PCs (1980s)

- PCs returned to single user paradigm
  - Initially minimal I/O and system services
  - File systems & interactivity from timesharing systems
- Advent of personal productivity applications
  - High end applications gave rise to workstations
- Advent of local area networking
  - File transfer and e-mail led to group collaboration
  - The evolution of work groups and work-group servers
- PCs and workstations "grew together"
- OS worked for one user, but ran multiple processes for him

Embedded Systems (1990s)

- General purpose systems vs. appliances
  - Running software vs. performing a service
- Many appliances based on computers
  - Video games, CD players, TV signal decoders
  - Telephone switches, avionics, medical imaging
- Appliances require increasingly powerful OSs
  - Multi-tasking, networking, plug-n-play devices
- General purpose OS becoming more appliance-like
  - Ultra-high availability, more automation
  - Easier to use, less management intensive

Client/Server Computing (1990s)

- Computing specifically designed to provide services across the network
  - To multiple distinct users, but using the same service
  - Centralized file and print servers for work groups
  - Centralized mail, database servers for organizations
  - World Wide Web for everybody
  - Clients got thinner, servers became necessary
- Wide-Area Networking
  - No longer just on a LAN
  - e-mail, HTML/HTTP and the World Wide Web
  - Electronic business services

Distributed and Cloud Computing (2000s)

- Distributed Computing Platforms
  - Single servers couldn’t handle required loads
  - So services offered by/among groups of systems
    - Sometimes load balancing, sometimes functionally divided
  - System services must enable distributed applications
- More recently, move to general remote distributed pools of computers
  - Cloud computing
  - Providing arbitrary distributed computing for many users

Ubiquitous and Mobile Computing

- Modern devices put great computing power in everyone's hands
  - E.g., a typical tablet or smart phone
- Networking available in most places
  - But at varying qualities
  - Perhaps other local sensing and computation, too
- Most activities require some remote access
  - The "powerful" computer may not be able to do much on its own
  - Often primarily an interface device

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

Another Important OS Trend

- Convergence
  - There are a handful of widely used OSs
  - New ones come along very rarely
- OSs in the same family (e.g., Windows or Linux) are used for vastly different purposes
  - Making things challenging for the OS designer
- Most OSs 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

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