Introduction
CS 111
Operating System Principles
Peter Reiher

Outline

- Administrative materials
- Why study operating systems?

Administrative Issues

- Instructor and TA
- 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

TA

- ???
- Lab sessions Fridays from 10-12 AM, in Geology 4660
- Office hours to be announced

Instructor/TA Division of Responsibilities

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

Web Site

- http://www.lasr.cs.ucla.edu/classes/cs111_summer2014
- 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 Introduction to Computer Science II
  - Objects, data structures, queues, stacks, tables, trees
- CS 33 Introduction to Computer Organization
  - Assembly language, registers, memory
  - Linkage conventions, stack frames, register saving
- CS 35 Software Construction Laboratory
  - Fundamental software tools used in handling complex systems

Course Format

- Two weekly (average 20 page) reading assignments
  - Mostly from the primary text
  - A few supplementary articles available on web
- Two weekly lectures
- Midterm and final exams
- 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
  - Summer schedule only increases the pace
- 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
- 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: end of the 4th week (in recitation 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: Last day of 8th week (recitation section)
- 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
- 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
  - TA 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 TA

Late Assignments & Make-ups

- Labs
  - Due dates set by TA
  - TA also sets policy on late assignments
- Exams
  - Only possible with prior consent of the instructor
  - Be careful of the exam dates!
  - If you miss it, you're out of luck

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, network protocols, computer architectures, ...
- Prepare you for advanced courses
  - Data bases and distributed computing
  - Security, fault-tolerance, high availability
  - 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
  - Must be able to change policies without changing mechanisms
- Parallelism and asynchrony are powerful and necessary
  - But dangerous when used carelessly

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

What Is An Operating System?

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

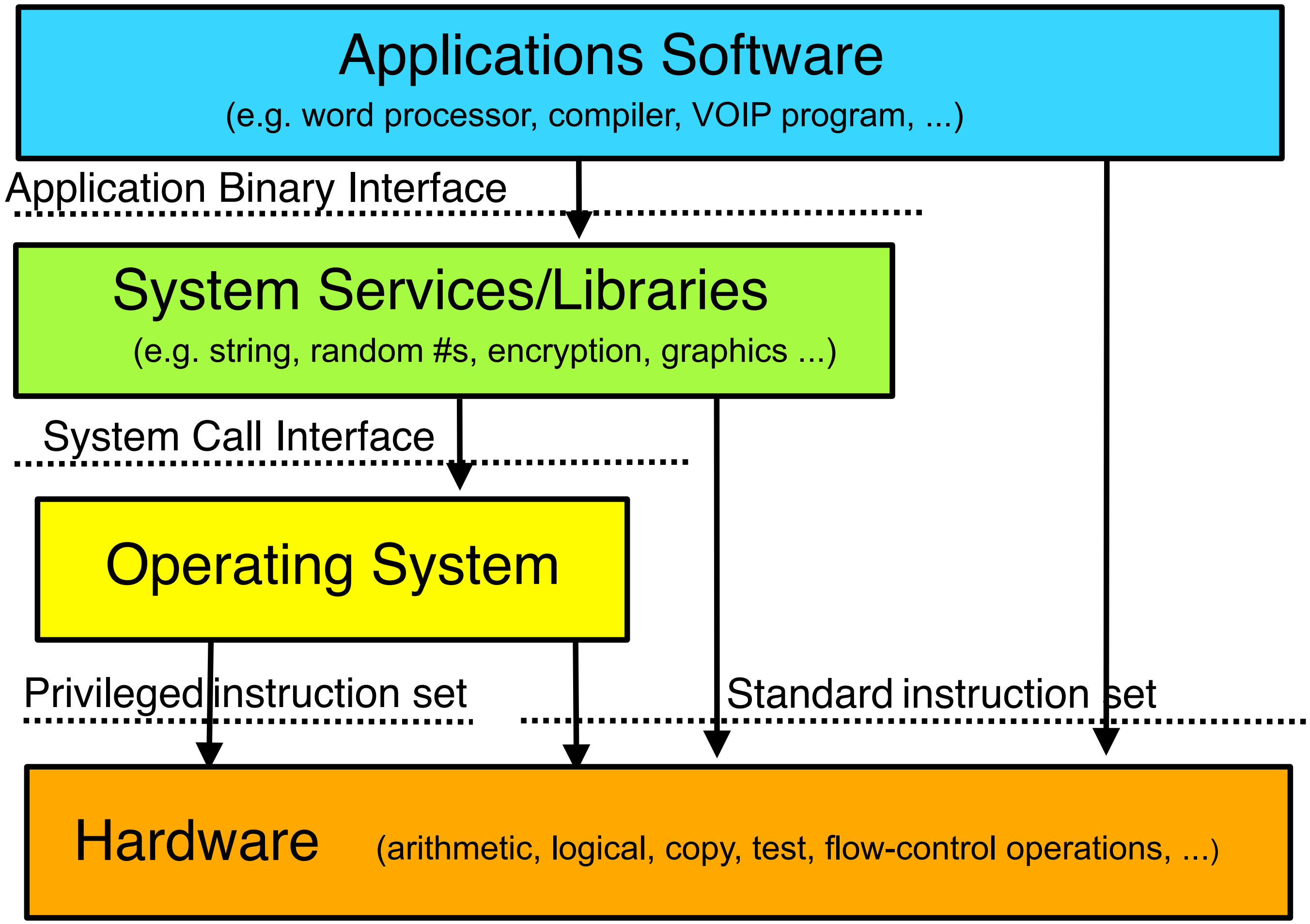
What Does an OS Do?

- 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 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 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
  - 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
  - The OS can make direct use of privileged instructions
- Thus, there's a push to move services with strong performance requirements down to the OS

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

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
  - A switch that totally hides the fact that the resource used to belong to someone else
- Examples: printers, bathroom stalls

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, dormitory 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)

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
- 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

A Resulting OS Challenge

- We are basing the OS we use today on an architecture designed 30-40 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

Important OS Properties

- For real operating systems built and used by real people
- Differs depending on who you are talking about
  - Users
  - Service providers
  - Application developers
  - OS developers

For the End Users,

- Reliability
- Performance
- Upwards compatibility in releases
- Support for differing hardware
  - Currently available platforms
  - What's available in the future
- Availability of key applications
- Security

Reliability

- Your OS really should never crash
  - Since it takes everything else down with it
- But also need dependability in a different sense
  - The OS must be depended on to behave as it's specified
  - Nobody wants surprises from their operating system
  - Since the OS controls everything, unexpected behavior could be arbitrarily bad

Performance

- A loose goal
- The OS must perform well in critical situations
- But optimizing the performance of all OS operations not always critical
- Nothing can take too long
- But if something is "fast enough," adding complexity to make it faster not worthwhile

Upward Compatibility

- People want new releases of an OS
  - New features, bug fixes, enhancements
  - Security patches to protect from malware
- People also fear new releases of an OS
  - OS changes can break old applications
- What makes the compatibility issue manageable?
  - Stable interfaces

Stable Interfaces

- Designers should start with well specified Application Interfaces
  - Must keep them stable from release to release
- Application developers should only use committed interfaces
  - Don’t use undocumented features or erroneous side effects

APIs

- Application Program Interfaces
  - A source level interface, specifying:
    - Include files, data types, constants
    - Macros, routines and their parameters
- A basis for software portability
  - Recompile program for the desired architecture
  - Linkage edit with OS-specific libraries
  - Resulting binary runs on that architecture and OS
- An API compliant program will compile & run on any compliant system

ABIs

- Application Binary Interfaces
  - A binary interface, specifying
    - Dynamically loadable libraries (DLLs)
    - Data formats, calling sequences, linkage conventions
  - The binding of an API to a hardware architecture
- A basis for binary compatibility
  - One binary serves all customers for that hardware
    - E.g. all x86 Linux/BSD/MacOS/Solaris/…
    - May even run on Windows platforms
- An ABI compliant program will run (unmodified) on any compliant system

For the Service Providers,

- Reliability
- Performance
- Upwards compatibility in releases
- Platform support (wide range of platforms)
- Manageability
- Total cost of ownership
- Support (updates and bug fixes)
- Flexibility (in configurations and applications)
- Security

For the Application Developers,

- Reliability
- Performance
- Upwards compatibility in releases
- Standards conformance
- Functionality (current and roadmap)
- Middleware and tools
- Documentation
- Support (how to ...)

For the OS Developers,

- Reliability
- Performance
- Maintainability
- Low cost of development
  - Original and ongoing

Maintainability

- Operating systems have very long lives
  – Solaris, the “new kid on the block,” came out in 1993
- Basic requirements will change many times
- Support costs will dwarf initial development
- This makes maintainability critical
- Aspects of maintainability:
  – Understandability
  – Modularity/modifiability
  – Testability

Critical OS Abstractions

- One of the main roles of an operating system is to provide abstract services
  - Services that are easier for programs and users to work with
- What are the important abstractions an OS provides?

Abstractions of Memory

- Many resources used by programs and people relate to data storage
  - Variables
  - Chunks of allocated memory
  - Files
  - Database records
  - Messages to be sent and received
- These all have some similar properties

The Basic Memory Operations

- Regardless of level or type, memory abstractions support a couple of operations
  - WRITE(name, value)
    - Put a value into a memory location specified by name
  - value <- READ(name)
    - Get a value out of a memory location specified by name
- Seems pretty simple
- But going from a nice abstraction to a physical implementation can be complex

An Example Memory Abstraction

- A typical file
- We can read or write the file
- We can read or write arbitrary amounts of data
- If we write the file, we expect our next read to reflect the results of the write
  - Coherence
- If there are several reads/writes to the file, we expect each to occur in some order
  - With respect to the others

Abstractions of Interpreters

- An interpreter is something that performs commands
- Basically, the element of a computer (abstract or physical) that gets things done
- At the physical level, we have a processor
- That level is not easy to use
- The OS provides us with higher level interpreter abstractions

Basic Interpreter Components

- An instruction reference
  - Tells the interpreter which instruction to do next
- A repertoire
  - The set of things the interpreter can do
- An environment reference
  - Describes the current state on which the next instruction should be performed
- Interrupts
  - Situations in which the instruction reference pointer is overriden

An Example Interpreter Abstraction

- A CPU
- It has a program counter register indicating where the next instruction can be found
  - An instruction reference
- It supports a set of instructions
  - Its repertoire
- It has contents in registers and RAM
  - Its environment

Abstractions of Communications Links

- A communication link allows one interpreter to talk to another
  - On the same or different machines
- At the physical level, wires and cables
- At more abstract levels, networks and interprocess communication mechanisms
- Some similarities to memory abstractions
  - But also differences

Basic Communication Link Operations

- SEND(link_name, outgoing_message_buffer)
  - Send some information contained in the buffer on the named link
- RECEIVE(link_name, incoming_message_buffer)
  - Read some information off the named link and put it into the buffer
- Like WRITE and READ, in some respects

An Example Communications Link Abstraction

- A Unix-style socket
- SEND interface:
  - send(int sockfd, const void *buf, size_t len, int flags)
  - The sockfd is the link name
  - The buf is the outgoing message buffer
- RECEIVE interface:
  - recv(int sockfd, void *buf, size_t len, int flags)
  - Same parameters as for send

Some Other Abstractions

- Actors
  - Users or other "active" entities
- Virtual machines
  - Collections of other abstractions
- Protection environments
  - Security related, usually
- Names
- Not a complete list
- Not everyone would agree on what's distinct