Introduction to Operating Systems

1A. Administrative introduction to course
1B. Why is OS a required course?
1C. What is an Operating System?
1D. Operating Systems Principles
1E. A (very) Brief History of Operating Systems

Instructor

- Background (non-academic)
  - professional engineer w/over 40 years in OS
    - commercial Unix/Linux, SMP and distributed
    - development, leadership, staff and executive roles
  - I am here because I love teaching and I love OS
- Getting in touch with me (in order)
  - email: mark.kampe@gmail.com
  - GoogleTalk: mark.kampe@gmail.com
  - office: BH 4531M, TR 1-1:50, 4-4:50

This Course

- This is a revised curriculum with new goals:
  - understanding and exploiting OS services
  - foundation concepts and principles
  - common problems that have been solved in OS
  - evolving directions in system architecture
- This is not a course in how to build an OS
  - you will not read or write any kernel-mode code
  - you will not study or build any parts of a toy OS

Learning Objectives

- We started with a list of learning objectives
  - over 300 concepts, issues, approaches and skills
- All activities in this course are based on them
  - the reading has been chosen introduce them
  - the lectures are designed to reinforce them
  - the projects have been chosen to exercise them
  - the exams will test your mastery of them
- Study this list to understand the course goals
- Use this list to guide your pre-exam review

Course Web Site(s)

http://web.cs.ucla.edu/classes/spring17/cs111
- course syllabus
- reading, lecture and exam schedule
- copies of lecture slides
- supplementary reading and study materials

https://ccle.ucla.edu/course/view/17S-COMSCI111-1
- announcements
- (per lecture) on-line quizzes
- projects descriptions and submission
- discussion forum (and lecture topic requests)

Reading and Quizzes

- Reading
  - Remzi Arpaci-Dusseau OS in Three Easy Pieces
  - numerous monographs to fill in gaps
  - average 40pp/day, but there is one 84 page day
- Quizzes
  - 4-8 short questions on the assigned reading
  - online (CCLE), due before start of each lecture
  - purpose: to ensure that you do the reading

Lectures

- Lectures will not
  - re-teach material well-covered by the reading
- Lectures will be used to
  - clarify and elaborate on the reading
  - explore implications and applications
  - discuss material not covered by the reading
  - discuss questions raised by students
- All lecture slides will be posted on-line
  - to aid you in your note-taking and review

Projects

- Format:
  - individual programming projects w/questions
  - written in C to be run on Linux systems
  - one will require you to buy an Intel Edison kit
- Goals:
  - Develop ability to exploit OS features
  - Reinforce principles from reading and lectures
  - Develop programming/problem solving ability
  - Practice software project skills

Projects

- Subjects
  - P0 – a warm-up to confirm your readiness
  - P1 – processes, I/O and IPC (2 parts)
  - P2 – synchronization (2 parts)
  - P3 – file systems (2 parts)
  - P4 – Embedded Systems/Internet of Things (3 parts)
- broken into ~weekly deliverables
  - start each project as soon as you finish previous
  - be ready to discuss problems on Friday
  - finish the project over the weekend

Instructor/TA Responsibilities

- Instructor: lectures, readings, and tests
  - ask me about issues related to these
  - TA's do not follow the reading and lectures
- TA's: projects
  - they will do all assistance and grading
  - all questions on projects should go to them

Course Grading

- Basis for grading:

– quizzes	10%
– projects	45% (P0 5%, all others 10%)
– midterm	15%
– final exam part-1	15%
– final exam part-2	15%

- I do not grade on a curve
  - I do look at score distribution to set break points

Late Assignments & Make-ups

- Quizzes
  - no late quizzes accepted, no make-ups
  - but I usually drop the lowest score
- Labs
  - each student gets FIVE slip days (usable on any project)
  - after that score drops by 10% per late day
- Exams
  - alternate times or make-ups may be schedulable (with advanced notice)

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

Academic Honesty

- Acceptable:
  – study and discuss problems/approaches w/friends
  – independent research on problems/approaches
- Unacceptable:
  – submitting work you did not independently create (or failing to cite your sources)
  – sharing code or answers with class-mates
  – using reference materials in closed-book exams
- Detailed rules are in the course syllabus

Academic Honesty – Projects

- Do your own projects
  – If you need additional help, ask the instructor
- You must design and write all 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

Why is OS a required course?

- Most CS discussions involve OS concepts
- Many hard problems have been solved in OS
  – synchronization, security, scalability, distributed computing, dynamic resource management, …
  – the same solutions apply in other areas
- Few will ever build an OS, but most of us will:
  – set-up, configure, and manage computer systems
  – write programs that exploit OS features
  – work w/complex distributed/parallel software
  – build abstracted services and resources
  – troubleshoot problems in complex systems

Relation to Other Courses

- Build on concepts and skills from other courses
  – data structures, algorithms, computer architecture
  – programming languages, assembly language programming
- Provide you with valuable foundation concepts
  – processes, threads, virtual address space, files
  – capabilities, synchronization, leases, deadlock, granularity
- Prepare you to work with more advanced subjects
  – data bases, file systems, and distributed computing
  – security, fault-tolerance, high availability
  – computer system modelling, queuing theory, ...

Why do I build Operating Systems?

- They (and their problems) are extremely complex
- They are held to high pragmatic standards:
  – performance, correctness, robustness, scalability, availability, maintainability, extensibility
  – they demand meticulous attention to detail
- They must also meet high aesthetic standards
  – general, powerful, and elegant (these characteristics make the complexity manageable)
- The requirements are ever changing
  – exploit the capabilities of ever-evolving hardware
  – enable new classes of systems and applications
- *Worthy adversaries* attract interesting people

What does Operating System do?

- manages the hardware
  - fairly allocate hardware among the applications
  - ensure privacy and enable controlled sharing
  - oversee program execution and handle errors
- abstract the bare hardware
  - make it easier to use
  - make the applications platform-independent
- new abstractions to enable applications
  - powerful features beyond the bare hardware

What makes the OS special?

- It is always in control of the hardware
  - first software loaded when the machine boots
  - continues running while apps come and go
- Parts of it have complete access to hardware
  - privileged instructions, all memory and devices
  - mediates application access to the hardware
- It is trusted
  - to store, manage, and protect critical data
  - to perform all requested operations in good faith

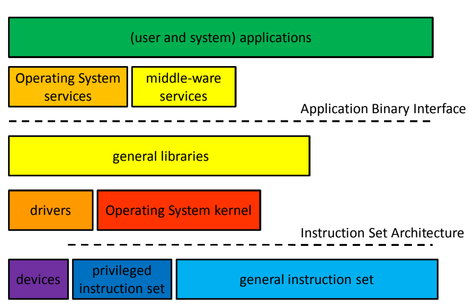
Privileged Instructions

- most CPU instructions can be used by anyone
  - e.g. arithmetic, logical, data movement, flow control
- some instructions are privileged
  - e.g. operations associated with I/O, interrupts, virtual address spaces, and processor mode.
  - these could compromise data privacy or integrity
  - they can only be executed when in privileged modes
  - otherwise they are illegal operations (cause exception)
- the operating system runs in privileged modes
  - giving it full control of the computer

What does an OS look like?

- applications see objects and operations
  - CPU supports data types and operations
    - bytes, shorts, longs, floats, pointers …
    - add, multiply, copy, compare, indirection, branch …
  - OS supports richer objects, higher operations
    - files, processes, threads, segments, ports, …
    - create, destroy, read, write, signal, …
- much of what OS does is behind-the-scenes
  - plug & play, power management, fault-handling, domain services, upgrade management, …

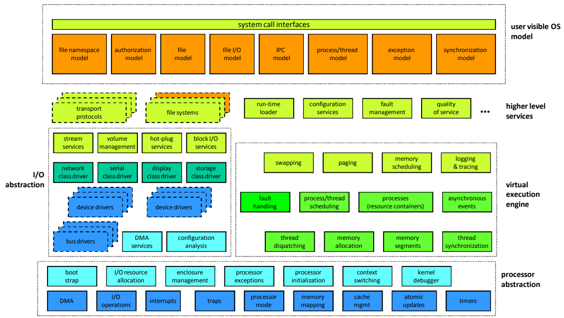
Software Layering



What functionality is in the OS

- as much as necessary, as little as possible
  - OS code is very expensive to develop and maintain
  - it is important to distinguish OS from kernel
- functionality must be in the OS if it …
  - requires the use of privileged instructions
  - requires the manipulation of OS data structures
  - required for security, trust, or resource integrity
- other simple functions can be in libraries
- complex functionality provided by services

Internal Structure (artists conception)



Complexity Management

- Layered/Hierarchical Structure
  – can be understood progressively, piece-at-a-time
- Modularity and Functional Encapsulation
  – hiding complexity and simplifying interfaces
- Generalizing and Unifying Abstractions
  – high level organizing concepts
  – reusable solution paradigms
- Indirection, Federation and Deferred Binding
  – TBD plug-ins for TBD problems
- Appropriate Abstraction
  – functionality well-suited for intended uses

S/W Principles from this course

- Mechanism/Policy Separation
  – to meet a wide range of evolving needs
- Interfaces as contracts
  – implementations are not interfaces
- Dynamic Equilibrium
  – robust adaptive resource allocation
- Fundamental role of data structures
  – find the right data structures, the code is easy
- Iterative Solutions/Progressive Refinement
  – incremental improvements to working approaches

Life lessons from Operating Systems

- There Ain't No Such Thing As A Free Lunch
  – everything has a cost, there are always trade-offs to make
- Keep it Simple, Stupid!
  – avoid overly complex/clever solutions
- The Devil is in the Details
  – precious few things are as simple as they initially seem
- Correctness and Expedience are often at odds
  – correct solutions are often complex and/or expensive
- Be very clear about what your goals are
  – make the right trade-offs, focus on the right problems
  – don't over-constrain your problems
- Responsible and sustainable living
  – take responsibility for our actions/consequences
  – nothing is lost, everything is eventually recycled

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

General OS Trends

- They have grown larger and more sophisticated
- Role has changed from shepherding the h/w to:
  – shielding applications from the hardware
  – providing powerful platform for applications
  – coordinating computation and data movement
- Best understood through services they provide
  – capabilities they add
  – applications they enable
  – problems they eliminate

OS Convergence

- In the 1960s, there were many OS
  - one for every different computer and use
  - they were (relatively) small, simple, and cheap
  - software portability wasn't even a concept
- The world is now a very different place
  - OS are extremely large, complex and expensive
  - software portability is critically important
  - the number of surviving OS is small and shrinking
  - they must serve a much wider range of platforms

Operating Systems Goals

- Application Platform
  - powerful
  - standards compliant
  - advanced/evolving
  - stable interfaces
  - tool availability
  - well supported
  - wide adoption
  - domain versatility
- Service Platform
  - high performance
  - robust and reliable
  - highly available
  - multi/omni-platform
  - managablility
  - well supported
- General
  - maintainable
  - extensible
  - binary distribution model

Assignments

- Project 0
  - look at the project description
  - get started on implementation
  - encounter problems before your lab session
- Reading for next Lecture
  - OS Principles
  - Interface Stability
  - Software Interfaces

Supplementary Slides

Maintainability

- operating systems have very long lives
  - basic requirements will change many times
  - support costs will dwarf initial development
  - this makes maintainability critical
    - understandability
    - modularity/modifiability
    - testability

Maintainable: understandability

- code must be learnable by mortals
  - it will not be maintained by the original developers
  - new people must be able to come up to speed
- code must be well organized
  - nobody can understand 1M lines of random code
  - it must have understandable, hierarchical structure
- documentation
  - high level structure, and organizing principles
  - functionality, design, and rationale for modules
  - how to solve common problems

Maintainable: modularity

- modules must be understandable in isolation
  - modules should perform coherent functions
  - well specified interfaces for each module
  - implementation details hidden within module
  - inter-module dependencies are few/simple/clean
- modules must be independently changeable
  - lots of side effects mean lots of bugs
  - changes to one module should not affect others
- Keep It Simple Stupid
  - costs of complexity usually outweigh the rewards

Maintainable: testability

- thorough testing is key to reliability
  - all modules must be thoroughly testable
  - most modules should be testable in isolation
- testability must be designed in from the start
  - observability of internal state
  - triggerability of all operations and situations
  - isolability of functionality
- testing must be automated
  - functionality, regression, performance,
  - stress testing, error handling handling

Portability to multiple ISAs

- 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 h/w
  - general frameworks are h/w independent
    - file systems, protocols, processes, etc.
  - h/w assumptions isolated to specific modules
    - context switching, I/O, memory management
  - careful use of types
    - word length, sign extension, byte order, alignment

Binary Distribution Model

- binary is the opposite of source
  - a source distribution must be compiled
  - a binary distribution is ready to run
- 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

Binary Configuration Model

- 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 w/o 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
  - apps only use committed interfaces
  - OS vendors preserve upwards-compatibility