CS111
Operating Systems Principles
Mark Kampe
(markk@cs.ucla.edu)

Introduction to Operating Systems
1A. Administrative introduction to course
1B. Why study Operating Systems?
1C. What is an Operating System?
1D. Operating Systems goals
1E. Principles to be covered in this course
1F. 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: markk@cs.ucla.edu
  — GoogleTalk: mark.kampe@gmail.com
  — office: BH 4532B, MW 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/spring16/cs111
• course syllabus
• reading, lecture and exam schedule
• copies of lecture slides
• supplementary reading and study materials
https://ccle.ucla.edu/course/view/16S-COMSCI111-1
• announcements
• (per lecture) on-line quizzes
• projects descriptions and submission
• discussion forum
Reading and Quizzes

• Reading
  – Remzi Arpaci-Dusseau OS in Three Easy Pieces
  – Saltzer – System Design (complexity and security)
  – 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

• Skill development and demonstration
  P0 – a warm-up to confirm your readiness
  P1 – processes, I/O and IPC (in 2 parts)
  P2 – synchronization (in 3 parts)
  P3 – file systems (in 2 parts)
  an embedded system project or research paper
  one part is due every Monday by midnight
  – start each project as soon as you finish previous
  – be ready to talk to TA about problems on Friday
  – finish the project over the weekend

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

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

Why do I build Operating Systems?

• 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:
  – they must be general, powerful, and elegant (to be
    understandable by a single person)
• 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
  — allocate hardware among the applications
  — enforce controlled sharing/privacy
  — oversee execution and handle errors
• abstract the bare hardware
  — make it easier to use
  — make the software more hardware 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
• It alone has 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

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, ...

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
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
  - manageability
  - well supported
- General
  - maintainable
  - extensible
  - binary distribution model

S/W Principles from this course

- Mechanism/Policy Separation
  - to meet a wide range of evolving needs
- Interfaces as contracts
  - implementations are not interfaces
- Appropriate abstraction and Information Hiding
  - to manage complexity and provide power
- Dynamic Equilibrium
  - robust adaptive resource allocation
- Fundamental role of data structures
  - find the right data structures, the code is easy

Life lessons from this course

- There Ain’t No Such Thing As A Free Lunch
  - everything has a cost, there are always trade-offs
- Keep it Simple, Stupid!
  - avoid overly complex/clever solutions
  - they 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
  - take responsibility for our actions/consequences
  - nothing is lost, everything is eventually recycled
  - it is all in the details

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

Assignments

- reading for the next lecture
  - Saltzer 1.4-1.5 (dealing with complexity)
  - Linux Programmers’ Guide: libraries and tools
- wikipedia articles:
  - linkage conventions
  - dynamic loading
  - APIs, ABIs

Quiz 2 is due before the lecture!
Get started on Project 0:
http://web.cs.ucla.edu/spring16/cs111/projects/Project0.html

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

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

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**

- 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