Operating Systems Principles

Mark Kampe - kampe@g.hmc.edu

- 1A - Administrative introduction to course
- 1B - Why should we study Operating Systems
- 1D - Principles to be covered in this course
- 1E - a (very) brief history of Operating Systems

Instructor

- Background (non-academic)
  - professional engineer w/over 45 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
  - office: McGregor 312

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
- But if you want to build systems infrastructure
  - you will find this an excellent foundation

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)

https://www.cs.hmc.edu/courses/2023/fall/cs134/

- course syllabus
- reading, lecture and exam schedule
- copies of lecture slides
- announcements, discussions: Canvas
- per-lecture quizzes: Canvas Quizzes
- project submissions: Canvas Assignments
- project feedback: Canvas Assignments

Reading and Quizzes

- Reading
  - Remzi Arpaci-Dusseau OS in Three Easy Pieces
  - numerous monographs to fill in gaps
  - average 25pp/day, but there is one 65 page day
- Quizzes
  - 4-5 short questions on the assigned reading
  - online, due before start of each lecture
  - purpose: to ensure that you do the reading, and show me what concepts are not yet clear

Lectures

- Lectures will try not to
  - 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 & topics 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
  - most written in C to be run on Linux systems
  - one will require you to obtain an SoC and sensors
- Goals:
  - Develop ability to exploit OS features
  - Reinforce principles from reading and lectures
  - Develop programming/problem solving ability
  - Develop ability to study and master new APIs
  - 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

Course Grading

- Basis for grading:
  - quizzes 10%
  - projects 40% (typically ~5% each)
  - presentations 10%
  - exams 40%
    - midterm 15%
    - final exam part-1 15%
    - final exam part-2 10%
- 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 will drop a few of the lowest scores
- 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)

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
  - reference materials in closed-book quiz/exam
- Detailed rules are in the course syllabus

Academic Honesty – Projects

- Do your own projects
  - If you need additional help, ask the instructor
  - project 3 can be done in 2-person teams
- 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 key subject?

- Most CS discussions involve OS concepts
- Many hard problems have been solved in OS
  - synchronization, dynamic resource management, security, scalability, distributed consistency, consensus, managing large & complex s/w, …
  - 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
  - design/implement abstracted services/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 I built 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

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 interface definitions
- 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 all consequences of our actions
  - 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
- 2000s large scale distributed systems
  - the network IS the computer
- 2010s cloud computing
  - don't think about computers (boxes), think about services

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 (threads, comm, NFS, KVS, …)
  - applications they enable (web services, big data, …)
  - problems they eliminate (node/link faliures, hetero, …)

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

Reading and Assignments

Reading:

- Kampe: Software Interrace Standards
- remember to take on-line quiz

Projects:

- start Project 0 (to confirm your C/Linux skills)

Supplementary Slides

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

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

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 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

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