

Lecture 0

Welcome to CSC 2227S

Topics in the Design and Implementation of Operating Systems

Plan for today

- Overview of CSC 2227S
 - How it'll work
 - What I expect from you

- What makes software systems tough and interesting
 - Reality
 - Complexity

- Goals and Topics

- What's next...

Overview of 2227S (Spring 2006)

- Check the web page for updates and news frequently
 - <http://www.cs.toronto.edu/~demke/2227S.06/>
- Components
 - Regular "lectures" by me (and possibly guest lecturers)
 - Critical study and discussion of systems papers
 - Summaries of papers
 - Term project
- Other stuff
 - No assigned books, but some on you might find useful (list on web page)
 - Prereqs
 - Grading plan

Making the grade in 2227

- Generally
 - put in the effort and your grade will take care of itself
- Breakdown
 - 60% project
 - 20% paper summaries
 - 20% class discussions
- Caveat/warning
 - this is an advanced graduate-level course, which means lots of effort on your part and less structure than undergraduate courses
 - if you dive into it, you'll learn lots and love it!

Prereqs

- Prereq: undergraduate OS
 - you should have a solid command of this material
 - if you don't, you will struggle
 - worse, you will not benefit nearly as much as you should
- Refresher questions – self-test and old homework
 - a bunch of questions from undergraduate OS
 - taken from CMU and Berkeley OS exams
 - the point is to swap in your OS knowledge
 - use your OS book(s) from undergrad
 - discuss the problems and topics with your peers
 - now is the time to refresh your memory!
 - This will not be handed in, but is a useful preparatory exercise

Paper reading and summaries

- 2-3 papers will be assigned for each week
 - You should read them carefully before the class
 - be prepared to recall and discuss their contents
 - You should type up a considered summary before class
 - You should hand in summary before class starts
 - don't be late or skip class to do this; participation counts too
- Summary contents: about 0.25-0.5 of a page
 - List the three most important things the paper says (to you)
 - Describe the paper's most glaring deficiency
 - Describe what the paper taught you about system building
 - DO NOT just repeat abstract or provide book report
- Grading
 - Complete/Incomplete
 - A very poor summary will be considered incomplete
 - Roughly 1% per paper (you can miss a few)

How to read a research paper

- Consider the source (don't dismiss, but do consider)
 - Who wrote it -- are they experts or unknowns?
 - Where was it published -- top journal or personal web page?
 - Other aspects: sponsor, review process, structure, tone, etc.
- Dig for the point
 - Read the abstract, intro, conclusion and related work (and bib)
 - Flip (semi-quickly) thru the paper, looking at headings, figures and data
 - Consider how much time you really want to devote to the guts
 - What is the hypothesis, how do they try to prove it, and do they succeed?
- Computer Systems papers
 - Often describe entire systems without a clear point or hypothesis
 - Unfortunately, they are sometimes worth the effort and sometimes not...
- Always think about more than what they are trying to tell you
 - How does the work relate to your research?
 - What did they do right? Wrong?
 - What other problems are created or can be solved by the work described?

2227 Projects

- Practical experience a must for understanding systems
 - Thus, you will be required to design, construct and evaluate an interesting software system
- What software system?
 - It's up to you
 - You are encouraged to propose your own project idea
 - various project topic ideas will be posted on web page to help
 - Projects that span traditional sub-areas of CS/CE are great
 - ... but it must relate to 2227
 - multiple of the 2227 topics should be involved
 - must be explicitly okay'd
- Working in groups of 2 is encouraged
- Talk to me early if special equipment is required for the project you want to do

2227 Project Documents

- **Project proposal - 5%**
 - 2 pages describing your project idea and plan
- **Project literature survey - 5%**
 - 2-3 pages (+ bib) describing related work and how it relates
- **Project design document - 5%**
 - 5 pages revising and detailing your project idea and plan
- **Project midpoint report - 5%**
 - 2 pages describing the status of your project
- **Poster session – 10%**
- **Project final report - 30%**
 - 10 pages describing the completed project, including the idea, the execution, the evaluation, and the related work

Where to find papers (quick tangent)

- You should not feel limited to reading the papers I give you
- Great source: web search engines and on-line paper listings
- Another great source: library!
 - Every serious researcher should spend time looking for related papers
- Some good computer systems conferences
 - SOSOP, OSDI, ASPLOS, Usenix, SIGMETRICS, SIGCOMM, ISCA, ...
- Some good computer systems journals
 - ACM Transactions on Computer Systems (TOCS)
 - IEEE Computer
 - Communications of the ACM
 - IBM Systems Journal

What makes software systems tough and interesting

- **Reality**
 - simplifying assumptions often don't hold up
 - people are rarely rational, arrivals are rarely Gaussian, environments are rarely clean, failures are rarely independent, etc...
 - poorly done systems can be incredibly expensive
 - billions of dollars and even life & death
- **Rapid changes in technology and applications**
 - technology advances change the rules
 - new applications change the requirements
- **Most generally: Complexity**
 - coping with complexity is what almost all of it boils down to

Let's try to define complexity, as a start

- **Webster: "the state of being complex"**
 - complex => "difficult to understand"
- **Relative term, not lending itself to quantification**
- **Symptoms of complexity**
 - large number of elements
 - large number of interconnections
 - irregularity (lots of exceptions, neither regular nor repetitive)
 - lack of a methodical description
 - like previous one, but highlights difficulty of understanding
 - minimum team size
 - combines all of above into "how many people to collectively get it"

Rapid pace of our field

- Technology is a major driver
 - Technology eliminates some problems and creates new ones (and enables new applications) over time
 - Incommensurate scaling makes things interesting
 - This means that one has to be on top of technology characteristics and trends
- New application requirements are another major driver
 - Changes the rules (assumptions), often forcing redesign
 - Example: video conferencing vs. best-effort networking
 - Example: mobile computing vs. file system caching
- Systems are complicated and consist of many parts
 - To do top-quality work, you must know about them all!
 - ... and their interactions too.

Problems in complex systems

- Propagation of effects
 - a “small, localized change” often has far-reaching effects
 - example: 13-inch tire to 15-inch tire to improve tire lifetime
 - consequences: wheel wells must be enlarged, spare tire space must be enlarged, back seat must be moved forward to accommodate spare
 - “there are no small changes in a large system”
- Surprises
 - an unexpected consequence of a change
 - example: 15-inch tires may weigh 21 pounds
 - consequence: exceed limit of current land shipping mechanism
- Incommensurate scaling
 - as a system scales up or down in size or speed, not all parts of it follow the same scaling rules
 - example: a mouse the size of an elephant would collapse

How does this translate to software systems?

- The software systems we’re concerned with suffer from all of these problems
- We’d like to have a constructive theory to apply
 - e.g., like linear control systems, thermodynamic systems, ...
- Unfortunately, we don’t
 - note that this would be a worthy career contribution
- Where does that leave us?
 - Case studies
 - Lessons from study of other complex systems
 - major difference is the unprecedented rate of change

Real examples of disasters

Taken from “Coping with Complexity”, Jerome Saltzer
SOSP 17, December 1999

- Tax system modernization
 - U.S. Internal Revenue Service
 - tried to replace 27 aging systems
 - Started 1989, scrapped 1997, spent: \$4B
 - Causes: complexity
 - all-or-nothing massive upgrade
- Advanced automation system
 - U.S. Federal Aviation Administration
 - tried to replace 1972 Air Route Traffic Control System
 - Started 1982, scrapped 1994, spent: \$6B
 - Causes: complexity
 - changing specifications, grandiose expectations, congressional meddling

More examples of disasters (from J. Saltzer)

- **Ambulance dispatching**
 - London ambulance service
 - automate dispatching and routing
 - Started 1991, scrapped 1992, cost: 20 lives lost (and \$2.5M)
 - shut down after 2 days of operation
 - Causes: complexity and poor management
 - unrealistic schedule (5 months), overambitious objectives, unidentifiable project manager, low bidder had no experience, no testing/overlap with old system, users not consulted during design
- **Automated DMV**
 - California department of motor vehicles
 - Started 1987, scrapped 1994, spent: \$44M
 - Causes: complexity and blame shedding
 - underestimated cost by 3X, slower than 1965 system, governor fired whistleblower, DMV blames Tandem, Tandem blames DMV

Some sources of system complexity

- **Large number of objectives**
 - example: new goals, requirements, or performance targets
- **Need for high utilization of limited resources**
 - example: single-track railroad line
- **Generality**
 - example: separately steerable front wheels
 - generally, generality increases complexity
 - frequently, it does so without real purpose

Techniques for coping with complexity

- **Modularity**
 - divide-and-conquer can often reduce growth (as function of size) from square to linear
- **Abstraction**
 - separation of interface from internals
 - or specification from implementation
- **Hierarchy**
 - builds on modularity by recursively grouping module sets
 - most surviving complex systems use it: army, company, etc
- **Level Definition**
 - a particular combination of modularity and abstraction
 - used extensively in computer systems
 - gates => microprocessor => higher-level language => algorithm

Some additional practical techniques

- **Control novelty**
 - sources of excessive novelty
 - second-system effect, better technology, marketing pressure, ego, ...
 - some novelty is necessary; the hard part is figuring out when to say NO
- **Install feedback**
 - design for iteration, iterate the design
 - something simple working first; one new problem at a time
- **Find bad ideas fast**
 - understand the design loop and examine the initial requirements
 - try ideas out, but don't hesitate to scrap them
- **Conceptual integrity**
 - one mind controls design
 - good aesthetics yield better systems
 - also makes them much easier to debug and maintain

What was the point of that?

- Well, what do operating systems do?
 - Provide abstractions of system resources
 - processes, files, sockets, malloc, etc...
 - Isolate application writers from details and each other
- Also, tend to be highly complex
 - many objectives
 - performance, reliability, ease of use, security, maintainability, ...
 - desire for high utilization of resources
 - generality: support all applications well
- Also, the problem keeps changing
 - technology advances and new applications
- ... and combining multiple systems (the soul of distributed systems) complicates everything further

Major goals of 2227

- Understand how OS design changes in response to
 - key technological advances (e.g., CPU vs. disk)
 - new application requirements (e.g., mobility, QoS)
 - advanced system objectives (e.g., fault tolerance, security)
- Understand key aspects of distributed systems
 - getting systems to talk to each other
 - marshalling separate resources to achieve common goals
 - figuring out where to perform particular functions
- Understand how to use available tools to build better systems
 - How do we declare one system “better” than another?
- Approach: case studies and existing experience

Major 2227 Topics

- System evaluation and benchmarking
- File systems, distributed file systems, decentralized storage
- Communication models and mobile code
- System structure and virtual machines
- Security dilemmas and solutions
- Concurrency, distributed event ordering, multi-party consensus
- Language tools for building better systems
- Fault tolerance and reliability
- Naming and resource discovery
- Mobile computing, and energy management
- Distributed operating systems
- Future environs

- Plan is to start with something concrete (see first line)
 - then, build up various other topics from a solid base

What's next ...

- Performance evaluation and benchmarking

- The papers
 - Lmbench: an extensible microbenchmark suite
 - Scale and Performance in a Distributed File System
 - Will be posted on the course web site by tomorrow