Week 1

Welcome to CSC469 / CSC2208: Advanced Operating Systems

Instructor:	Angela Demke Brown
TAs:	Abhishek Tiwari / Andy Hwang / Daniel Fryer
Email:	demke@cs.toronto.edu
Website:	http://www.teach.cs.toronto.edu/~csc469h/fall

(thanks to Bogdan Simion for some of the materials)

University of Toronto, Department of Computer Science

Plan for this week

- Overview of CSC 469 / CSC 2208
    - How it'll work
    - What I expect from you
    - What makes software systems tough and interesting
        - Reality
        - Complexity
    - Goals and Topics
- OS Structure

Overview (Fall 2018)

- Check web page for course notes, assignments, office hours, etc.
  - http://www.teach.cs.toronto.edu/~csc469h/fall
- Components
  - Regular lectures (by me) and discussion (by you)
  - Tutorials (concrete examples, assignment help, Q&A)
  - Three assignments
  - Midterm test
  - Final exam (CSC469) or project (CSC2208)
- Other stuff
  - Readings from the research literature will be assigned
  - No required text, but some recommended books

Making the grade in 469/2208

- Breakdown
  - 3 Assignments (15%, 15%, 15%)
    - Can work in pairs for all
    - Both partners must know all the work done in the assignment!
  - Midterm test (15%)
    - October 26th, 9-11AM (includes hour before tutorial time!)
  - Final exam (CSC469 only) or final project (CSC2208 only)
    - 40% final in exam period, cumulative (more focus on 2nd half)
- You need to be here (and in tutorial) to participate in discussions and get the most out of the class!

Assignments

Goal is to explore different operating systems concepts and the impact of design choices…

- Assignment 1 – Benchmarking (due Oct. 11)
- Assignment 2 – Concurrency (due Nov. 13)
- Assignment 3 – Fault Tolerance (due Dec. 5)

Start looking for partners now!

MarkUs instance will be available by end of week or early next week)

Assignments

- Due at 11:59 on the due date
- Grace tokens can be used, see info sheet
  - 3 grace tokens, each 24 hours
  - partners can use at most min(A,B) tokens
- Code **must work on the teaching labs/servers!**
- **Make sure you commit all your source files; we cannot find files you never submitted**
- Code style matters!
- **Test-as-you-go**
- The code you submit has to work, even if it doesn't implement everything
- **Code that does not compile gets zero marks!**

Assignments

- Write good, professional code
- Comment it properly, modularize it, etc.
- Debug it properly, find corner cases
- Solve problems as they come, find workarounds if needed
- Very important experience before getting a programming job
    - Please treat them as such!

More logistics

- Piazza link on the website
  - Useful for discussions, ready daily, ask questions there first
  - Come talk to me if you have any reservations about its terms of use
  - http://teaching.utoronto.ca/ed-tech/teaching-technology/piazza/

- Course info sheet (due dates, policies, etc.)
  - Linked from course webpage
  - Read carefully!

Prereqs and Refresher

- Prereq: CSC 369 or equivalent (ECE344, etc.)
  - 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
- Handout #1 & #2: helping you refresh
  - a bunch of questions from ACM Self Assessment Procedures
  - goal: to “swap in” your OS knowledge
    - use your favourite OS book
    - discuss the problems and topics with your peers
    - some of these questions will be discussed in tutorial
  - now is the time to refresh your memory!

Expectations

- What this course is or expects:
  - Cross-listed course, advanced topics!
  - Prior basic OS knowledge is implied (e.g., CSC369, ECE344)
  - Self-study is important, read papers!
  - You must be willing and ready to read more on your own!
- What this course isn't:
  - An intro course like CSC369
  - Something you can study a few days before the exam

Don’t Panic!

- Help is available in many forms
  - **Lectures/tutorials:** Ask questions!
  - **Office hours:** My time dedicated specifically to helping you
  - **Piazza:** Faster response!
  - Email: Longer turnaround time
  - TA Help Centre (may be helpful)
  - ***Anonymous feedback page …***

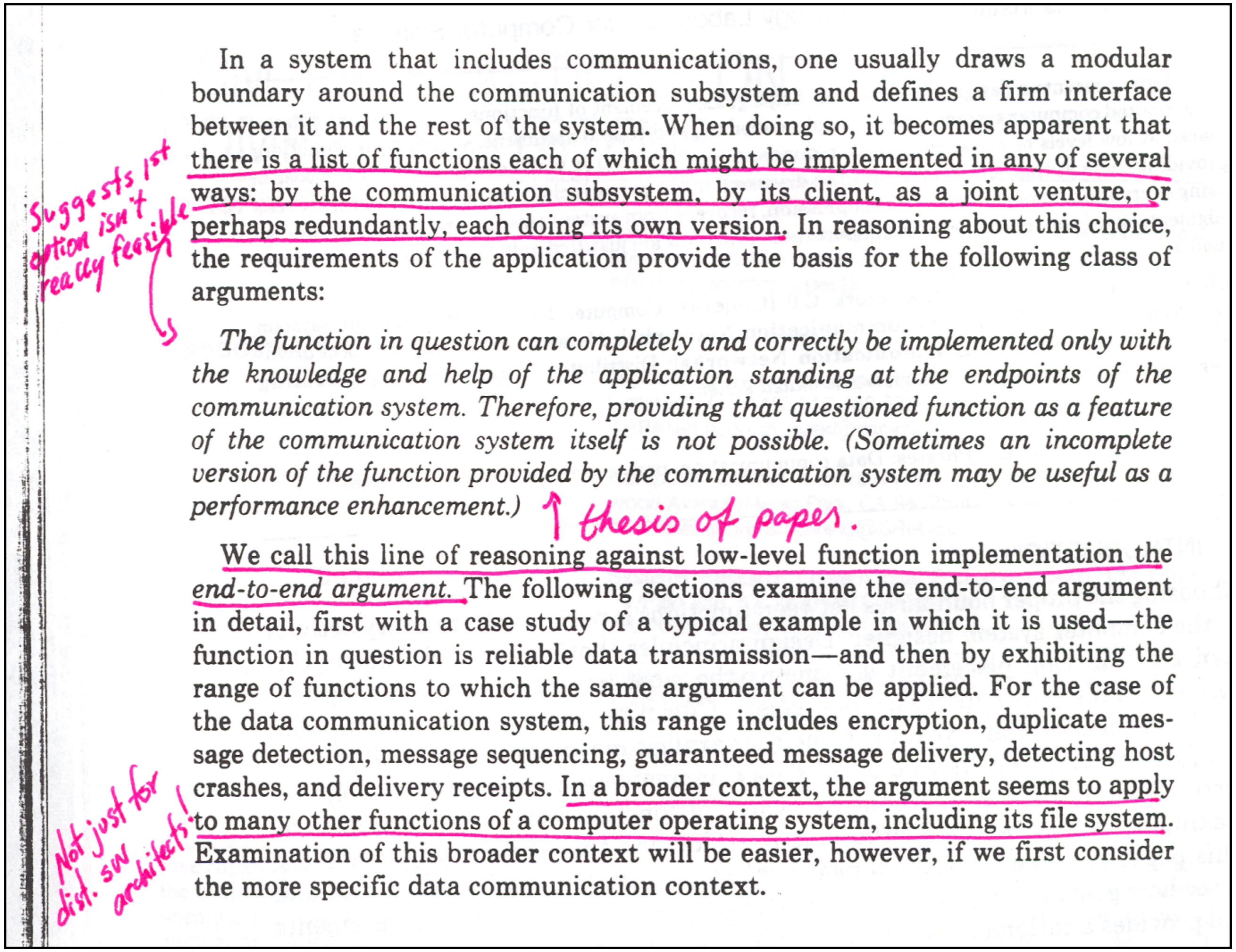
Don’t Copy!

- Academic Integrity: Plagiarism and cheating
  - Very serious academic offences
  - Clear distinction between collaboration and cheating
    - Of course you can help your friend track down a bug
    - It is **never ok to submit code that is not your own!**
    - Ask questions on Piazza, but **don’t add details about your solution (especially your code!)**
  - All potential cases will be investigated fully
  - **Don’t post your code in public places (Github, etc.) or look for solutions!**

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?
- Practice Active Reading
  - Underline key points, make notes in margin
  - Write down questions

Example: Active Reading



In a system that includes communications, one usually draws a modular boundary around the communication subsystem and defines a firm interface between it and the rest of the system. When doing so, it becomes apparent that there is a list of functions each of which might be implemented in any of several ways: by the communication subsystem, by its client, as a joint venture, or perhaps redundantly, each doing its own version. In reasoning about this choice, the requirements of the application provide the basis for the following class of arguments:

Suggests 1st option isn't really feasible

The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.) ↑ thesis of paper.

We call this line of reasoning against low-level function implementation the *end-to-end argument.* The following sections examine the end-to-end argument in detail, first with a case study of a typical example in which it is used—the function in question is reliable data transmission—and then by exhibiting the range of functions to which the same argument can be applied. For the case of the data communication system, this range includes encryption, duplicate message detection, message sequencing, guaranteed message delivery, detecting host crashes, and delivery receipts. In a broader context, the argument seems to apply to many other functions of a computer operating system, including its file system. Examination of this broader context will be easier, however, if we first consider the more specific data communication context.

Not just for dist. SW architects!

Introductory stuff

- Understanding systems in general
- Importance and impact of design decisions
- Problems in systems
- Tradeoffs
- Complexity

What is a “system”?

- Generic Definition:
    - Webster: “a complex unity formed of many often diverse parts subject to a common plan or serving a common purpose…”
    - a group of independent but inter-related elements comprising a unified whole
- Characteristics
    - Components, environment, boundary, emergence (the whole is more than the sum of the parts), processes, interfaces (I/O), structure
- Stability
    - Complex systems resist change
        - Social systems, environmental systems …

Why are 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

Real examples of disasters

- This and other examples taken from Jerome Saltzer's 1999 SOSP invited talk "Coping with Complexity"
  - Example 1: U.S. Tax system modernization to replace 27 aging systems
    - Started 1989, scrapped 1997, spent: $4B
    - Causes: complexity, all-or-nothing massive upgrade
    - IRS Assistant Commissioner Arthur Gross stated that the systems **"do not work in the real world."**
    - Started over ~1999 with new $5B contract
    - Development of centerpiece Customer Account Data Engine (CADE) software halted in 2009 due to unexpected complexities

Example 2: 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
  - Requirements reviewed, new contracts awarded
    - First use of early version of replacement system in 1999
- And the list goes on…

More examples of disasters

- 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, lowest bidder had no experience, no testing/overlap with old system, users not consulted during design
- Modern examples abound
  - AWS outages affect hundreds of services
  - Facebook network issue takes out WhatsApp and Instagram too

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
  - Must 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

- Emergent properties (aka surprises)
- Propagation of effects
    - “There are no small changes in a large system”
- Incommensurate Scaling
    - Not all parts of a system follow the same scaling rules
- Trade-offs

Emergent properties (aka Surprises)

- An unexpected consequence of a change
- Example: new (2006) TTC tokens were roughly 2.5X heavier than old ones
  - Consequence? Token shortage: delivery people can only carry half as many, delivering and filling token machines takes twice as long

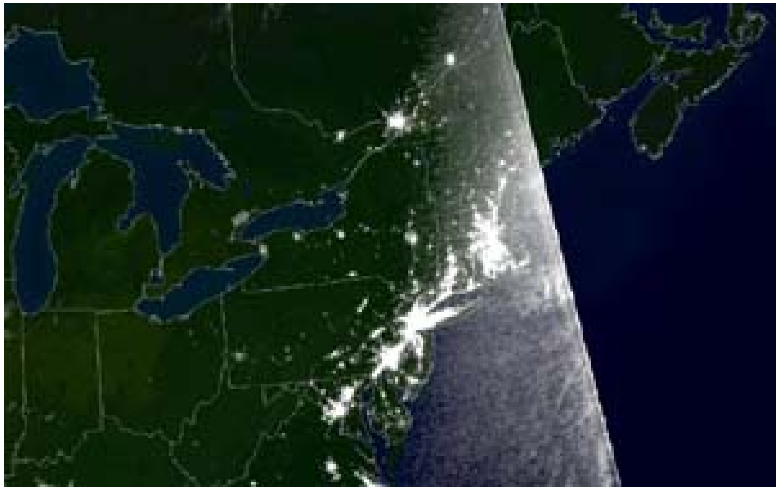
- Example: Millenium Bridge in London closed for 2 years after only 2 days of use
  - Small sway in bridge caused pedestrians to synchronize their steps, amplifying the sway

Millenium Bridge, London

(src: tripadvisor.com)

Propagation of Effects

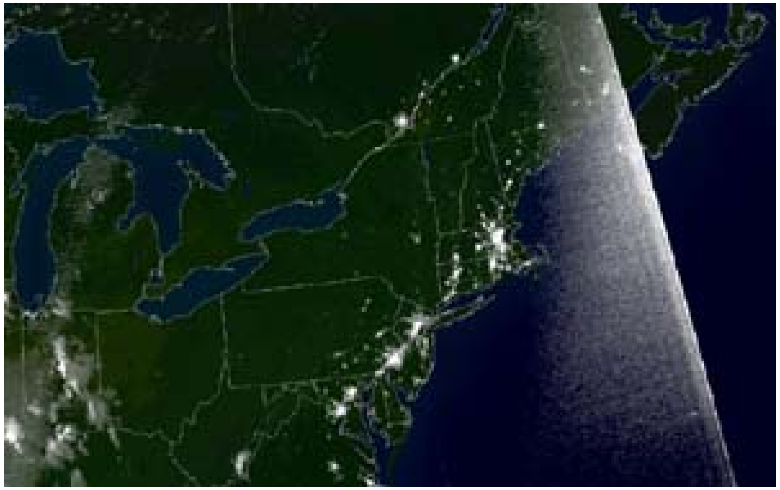
- A “small, localized change” often has far-reaching effects
  - Example: 13-inch tire to 15-inch tire to improve the ride
    - Consequences: wheel wells must be enlarged, spare tire space must be enlarged, back seat must be moved forward to accommodate spare, front seats must be thinner, etc.
  - +++Example: Northeast blackout, 2003
    - Bug (race condition) stalls alarm system, problems cascade from there



NOAA satellite images

←Night before

Night of →



- 2010 Facebook outage – automated “repair” caused more damage

☞ There are no “small changes” in a large system!

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
    - A structural redesign is needed instead

Galileo 1638: “To illustrate briefly, I have sketched a bone whose natural length has been increased three times and whose thickness has been multiplied until, for a correspondingly large animal, it would perform the same function which the small bone performs for its small animal.

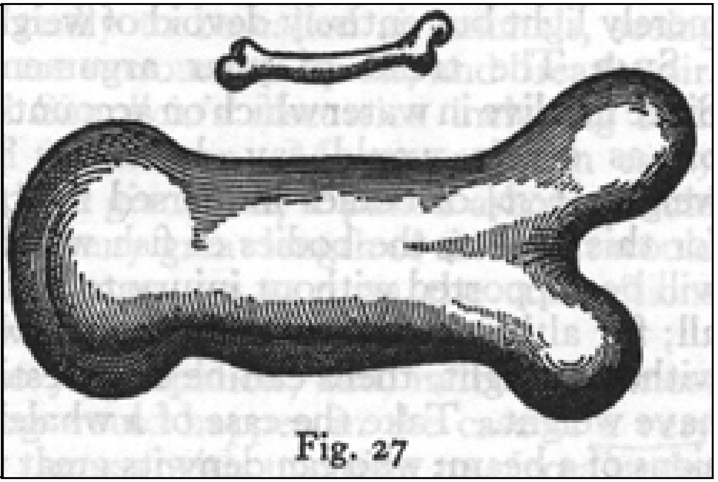


Fig. 27

Trade-Offs

- All the desirable features cannot be provided simultaneously
  - “Waterbed effect” – effort to reduce one problem makes another one worse
  - Example: Increase clock rate for performance
    - Also increases power consumption and risk of timing errors
    - Can reduce risk of timing errors with physically smaller circuit
    - But this means less area to dissipate heat from increased power consumption

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

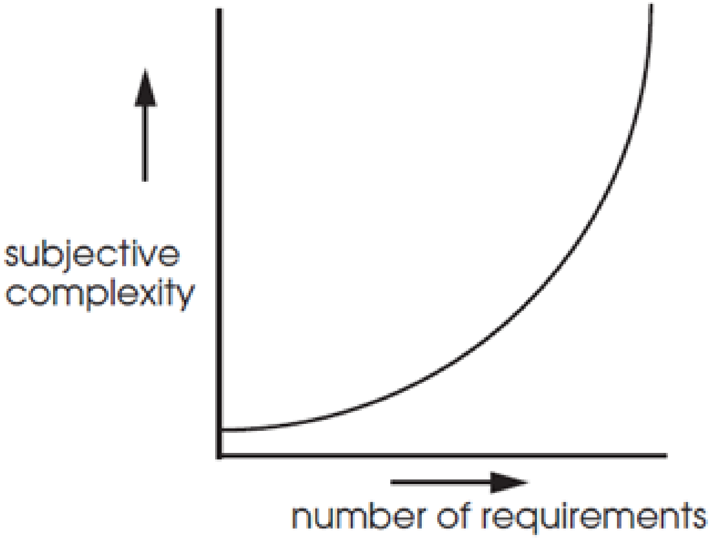
Let’s try to define complexity, as a start

- Webster: “the state of being complex”
    - complex == “difficult to understand”

- Relative term, not lending itself to quantification
- Symptoms of complexity
    - large number of components
    - 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”

Some sources of system complexity

- 1. Large number of objectives/requirements
  - Individually may be straightforward, interactions add complexity
  - Pressure for generality and exceptions/corner cases
  - New goals, requirements, or performance targets make it worse



Principle 1 (Escalating Complexity): "Adding a requirement increases complexity out of proportion"

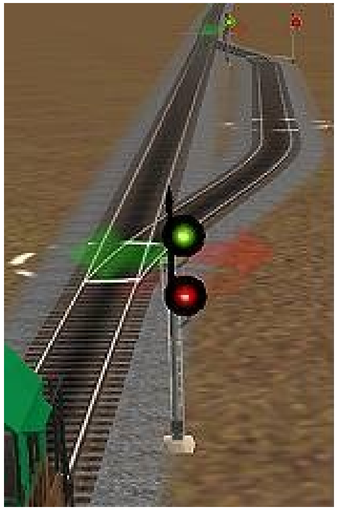
Some sources of system complexity

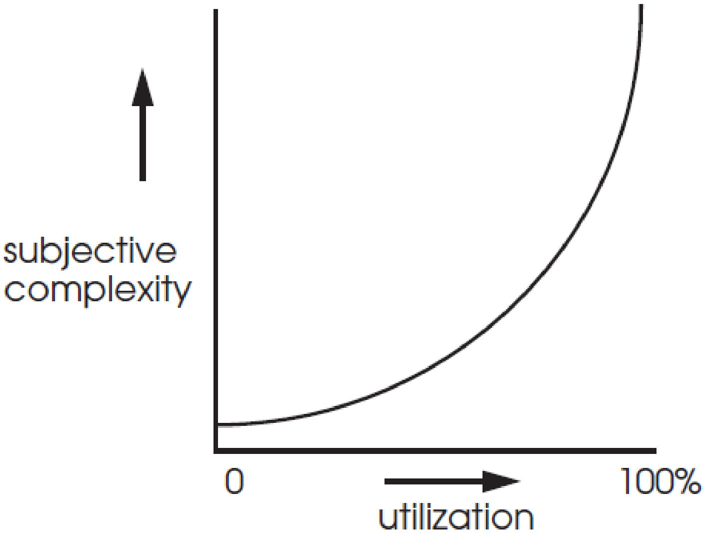
- 2. Generality
  - generally, generality increases complexity
  - frequently, it does so without real purpose
  - unnecessary generality - extreme example: separately steerable front wheels

Principle 2 (Avoiding Excessive Generality)
"If it's good for everything, it is good for nothing!"

Some sources of system complexity

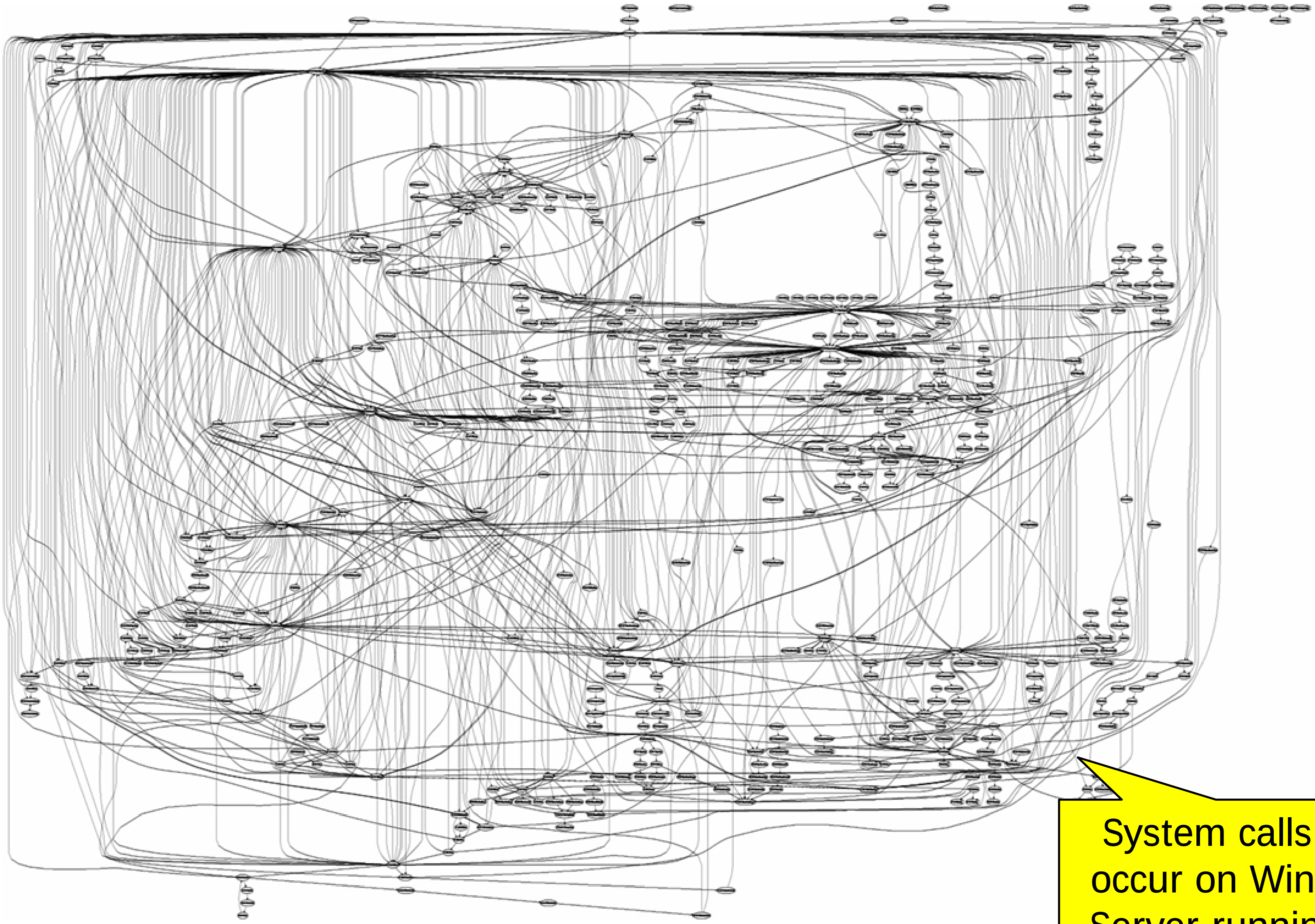
- 3. Need for high utilization of limited resources
  - example: single-track railroad line
  - The law of diminishing returns





Principle 3 (Diminishing Returns):
"The more one improves some measure of goodness, the more effort the next improvement will require".

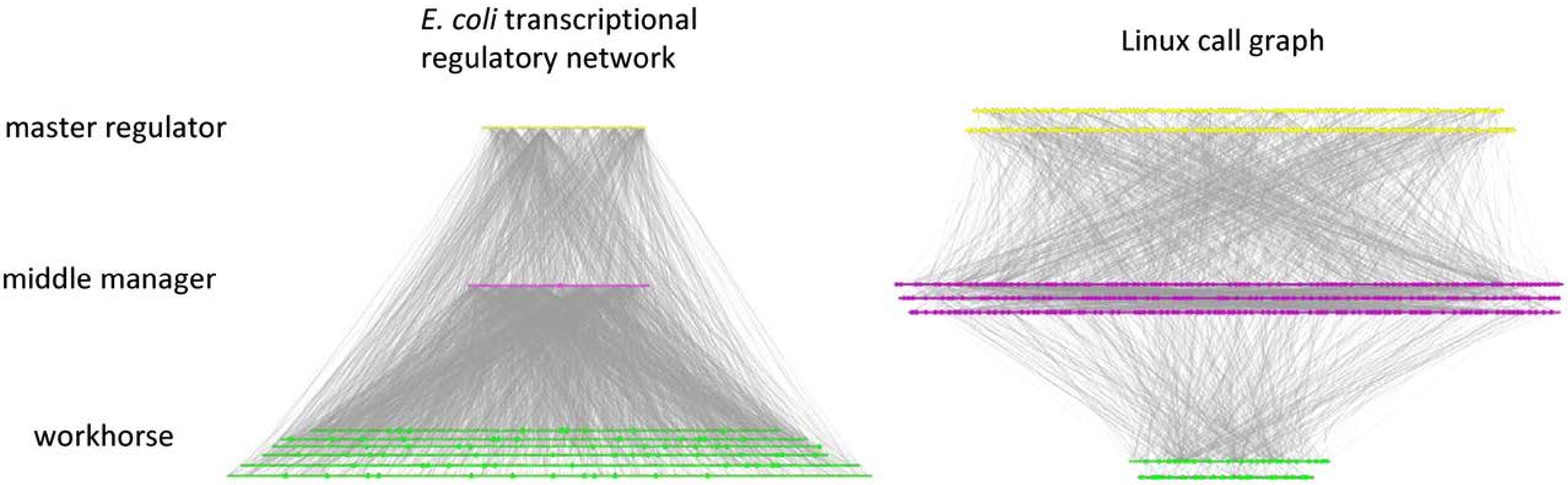
Windows system call graph



System calls that occur on Windows Server running IIS

Linux Call graph

The hierarchical layout of the E. coli transcriptional regulatory network and the Linux call graph.



As the genome of an organism grows larger, it can reuse its tools more often and thus require fewer and fewer new tools for novel metabolic tasks ... Thus, **it may be that further analysis will demonstrate the increasing resemblance of more complex eukaryotic regulatory networks to the structure of the Linux call graph.**

Yan K et al. PNAS 2010;107:9186-9191
©2010 by National Academy of Sciences

Next...

- Coping with complexity ...

Coping with complexity

- Modularity
  - Divide into collection of interacting subsystems (modules)
  - Easy to replace various components
- Abstraction
  - Separation of specification (interface) from implementation (internals)
  - Treat module only by its external specs (no knowledge of what's inside)
- Layering
  - Reduce module interconnections
  - Layer A module only communicates with next higher and lower layer Examples?
- Hierarchy
  - Assemble small group of modules into a self-contained subsystem, then a group of subsystems, etc. => form a hierarchy; also reduces module interconnections.

Still, not enough ..

- Modularity, abstraction, layering and hierarchy help, but not always enough to keep complexity under control
- The designer must understand the system being designed
- In the realm of computer systems
  - Hard to choose the right modularity from many plausible alternatives
  - Hard to choose the right abstraction
  - Hard to choose the right layering
  - Hard to choose the right hierarchy
- Only real guidance comes from experience with previous systems
  - Iteration (you won't get it right the first time – keep iterating)
  - Simplicity (KISS principle)

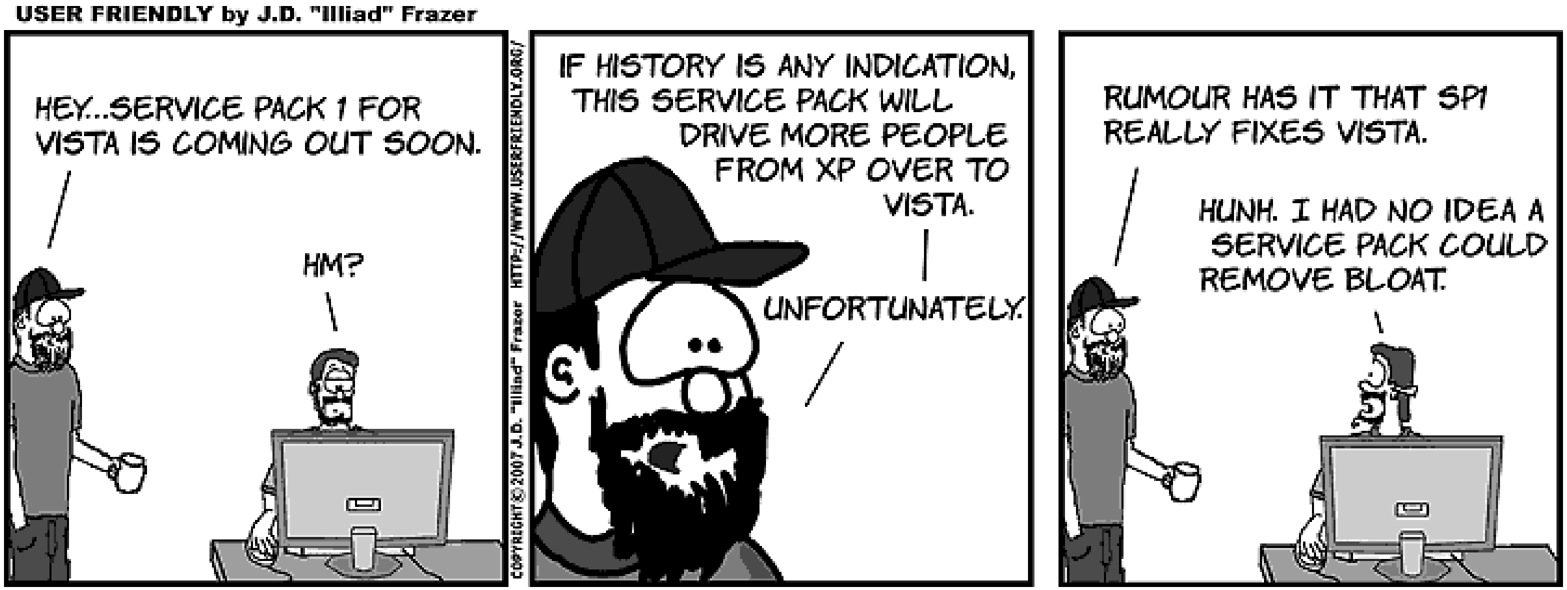
Major goals of 469

- 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 issues in multicore systems
  - Synchronization, Scheduling, Memory management …
- 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
- Approach: case studies and existing experience (papers!)

Major 469 Topics

- Operating system structure and internals
- Performance evaluation and benchmarking
- Communication models
- Concurrency, distributed event ordering, multi-party consensus
- Multiprocessor operating system issues
- Advanced virtual memory and storage systems
- Fault tolerance and distributed systems
- Security
- Impact on future of computer systems

And now, on to the content…


USER FRIENDLY by J.D. "Illiad" Frazer
HEY...SERVICE PACK 1 FOR VISTA IS COMING OUT SOON.
HM?
COPYRIGHT © 2007 J.D. "Illiad" Frazer HTTP://WWW.USERFRIENDLY.ORG/
IF HISTORY IS ANY INDICATION, THIS SERVICE PACK WILL DRIVE MORE PEOPLE FROM XP OVER TO VISTA.
UNFORTUNATELY.
RUMOUR HAS IT THAT SP1 REALLY FIXES VISTA.
HUNH. I HAD NO IDEA A SERVICE PACK COULD REMOVE BLOAT.

- Design options for operating system structure
  - 2 papers on design advice, 2 papers on OS structure:
    - Original UNIX paper (Ritchie & Thompson, 1974)
    - Original Mach paper (Acetta et al, 1986)

Overview

- Motivation: Why talk about structure?
    - Complexity!
- Kernel structures
    - Layered systems
    - Monolithic kernels
    - Open systems
    - Microkernels
    - Kernel Extensions (Next week)
    - Virtual Machines (Next week)

Complexity and the OS

- What do operating systems do?
  - Provide abstractions of system resources
    - processes, files, sockets, memory, 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

Techniques for coping with complexity (reminder)

- 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
- Layering and Hierarchy
  - builds on modularity by grouping module sets (into layers / hierarchy)
  - most surviving complex systems use hierarchy: army, company, etc.

- Now, let’s look at some general advice

The End-to-End Argument

- From J. H. Saltzer, D. P. Reed & D. D. Clark
- Briefly:

Don’t implement some function in low-level software layers if higher-level software must help get that function right.

Example: Careful file transfer

- Node A wants to transfer a file to Node B
- File is stored on A's local file system
- Transfer is successful if file is stored safely on B's local file system
- The communication network moves packets from Node A to Node B
- User-level file transfer application runs at A and B
- What could go wrong?

Possible problems

- Faults in storage system at A (file corrupted when read off disk)
- Software (file system, file transfer program, or communication layers) might make a mistake in either buffering or copying data, either at A or B
- CPU or RAM experiences transient error (bit flip) while doing copying or buffering, at either A or B
- Network drops or duplicates packets
- Host may crash part way through => incomplete transfer
- What can be done about it?

Possible Solutions

- Make each step as reliable as possible
  - File / storage system keeps checksums
  - Software is verified
  - Network layer handles lost/duplicate packets
  - Application on B confirms correct receipt and storage of file
- Alternative: end-to-end check and retry
  - Store with each file a checksum with sufficient redundancy to detect err.
  - B recalculates the checksum and makes sure it matches A's

Possible Solutions

- Common proposal: communication system provides internally a guarantee of reliable data transmission
  - Packet checksums, sequence number checking, retry schemes
  - The communication system is basically trying to be useful by adding some correctness guarantees. Is it useful though?
  - There are still problems that can happen at the application level, no matter how reliable the network is => still need application-level checks!
- Endpoint checks are required *no matter how reliable the intermediate steps are!*
  - So repeating the checks at the internal points just adds overhead!

Considerations

- Should the intermediate levels provide no reliability then?
  - Depends, choice depends on expected error frequency
  - E.g., what if network errors are too common?
  - E.g., what if network reliability is too beefed up?
  - Carefully consider the tradeoff!
- Use performance to justify function placement!
  - Performing a function at a lower level may be more efficient
  - However, it could also indirectly cost more.. Why?
    - Lower-level layer is common to many applications that might not need this function => slowdown others!

Performance trade-off is quite complex!

Awareness of end-to-end arguments helps reduce bad design!

Advice on System Design

- Butler Lampson’s “Hints for Computer System Design”, SOSP, 1983
- Key principle is separating interface (how clients interact with the system) from implementation
- Interface: set of assumptions
  - Simple, complete, and admits sufficiently small and fast implementation
- Functionality
- Speed
- Fault tolerance

Butler Lampson - Background

- Founding member of Xerox PARC (1970), DEC (1980s), MSR (current)
- ACM Turing Award (1992)
- Laser printer design
- PC
- Two-phase commit protocols
- Bravo, the first WYSIWYG text formatting program
- Ethernet, the first high-speed local area network (LAN)

Functionality Hints

- 1. Keep it simple
    - Do one thing at a time, and do it well
        - Get it right!
        - Make it fast, rather than general or powerful
        - Don’t hide power
        - Leave it to the client
            - End-to-end argument

More Functionality Hints

- 2. Maintain stability/continuity
  - Keep basic interfaces stable
  - Keep a place to stand
- 3. Making implementations work
  - Plan to throw one away, you will anyhow!
  - Keep secrets of the implementation
  - Divide and conquer
  - Use a good idea again instead of generalizing it
- 4. Handling all the cases
  - Handle normal and worst case separately

Speed Hints

- Split resources in a fixed way if in doubt
- Use static analysis if you can
- Use dynamic translation when its reasonable
- Cache results
- Use hints

Speed Hints

- Split resources in a fixed way if in doubt
- Use static analysis if you can
- Use dynamic translation when its reasonable
- Cache results
- Use hints
- Use brute force
- Compute in the background
- Batch if possible
- Safety first
- Shed load to control demand

Fault Tolerance Hints

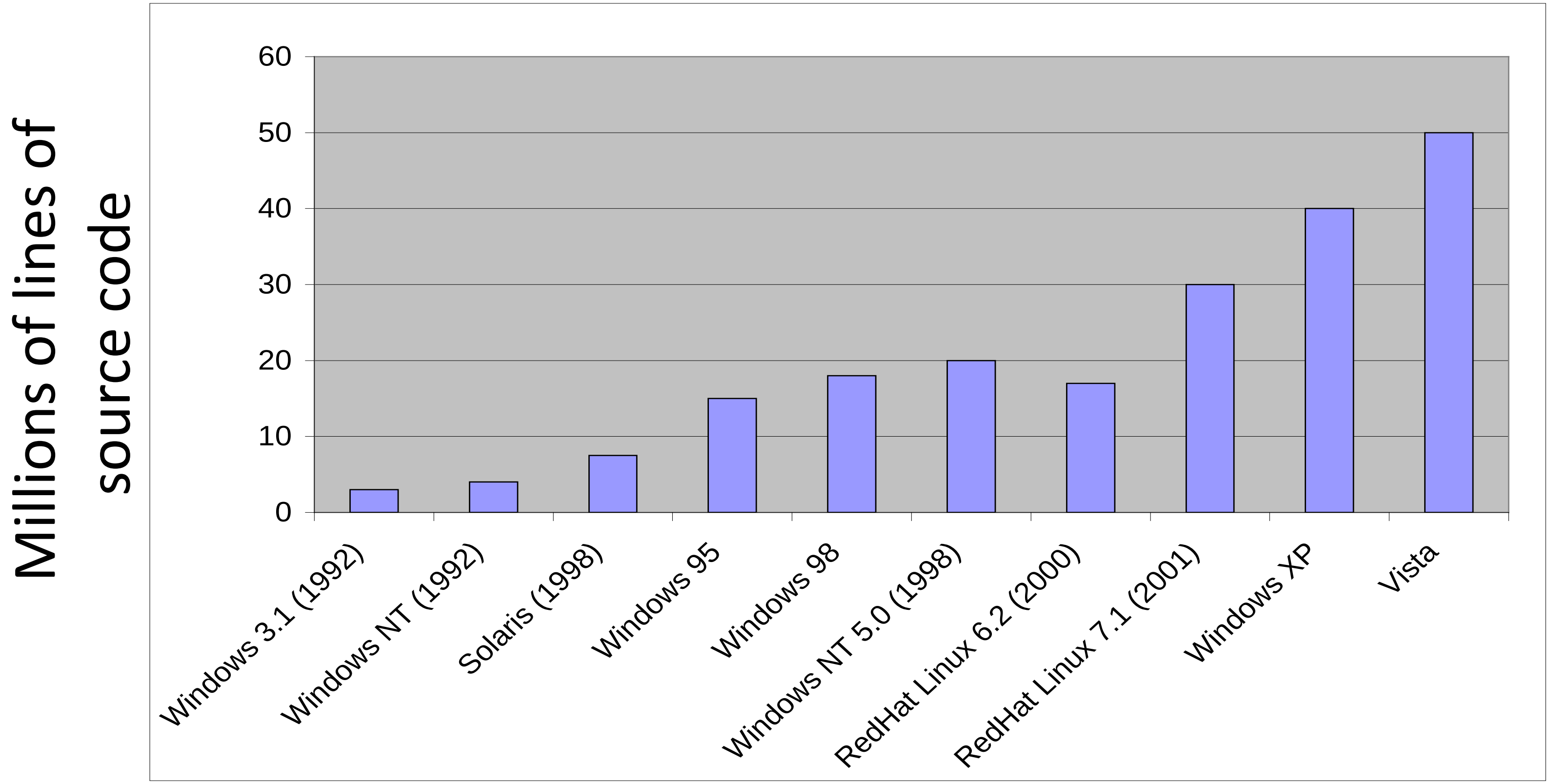
- End to end (Saltzer)
- Log updates
- Make actions atomic or restartable

Some add’l practical techniques

- Control novelty
    - sources of excessive novelty
        - second-system effect, better technology, marketing pressure…
    - 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

Software follows hardware

- Developers use capabilities of new hardware

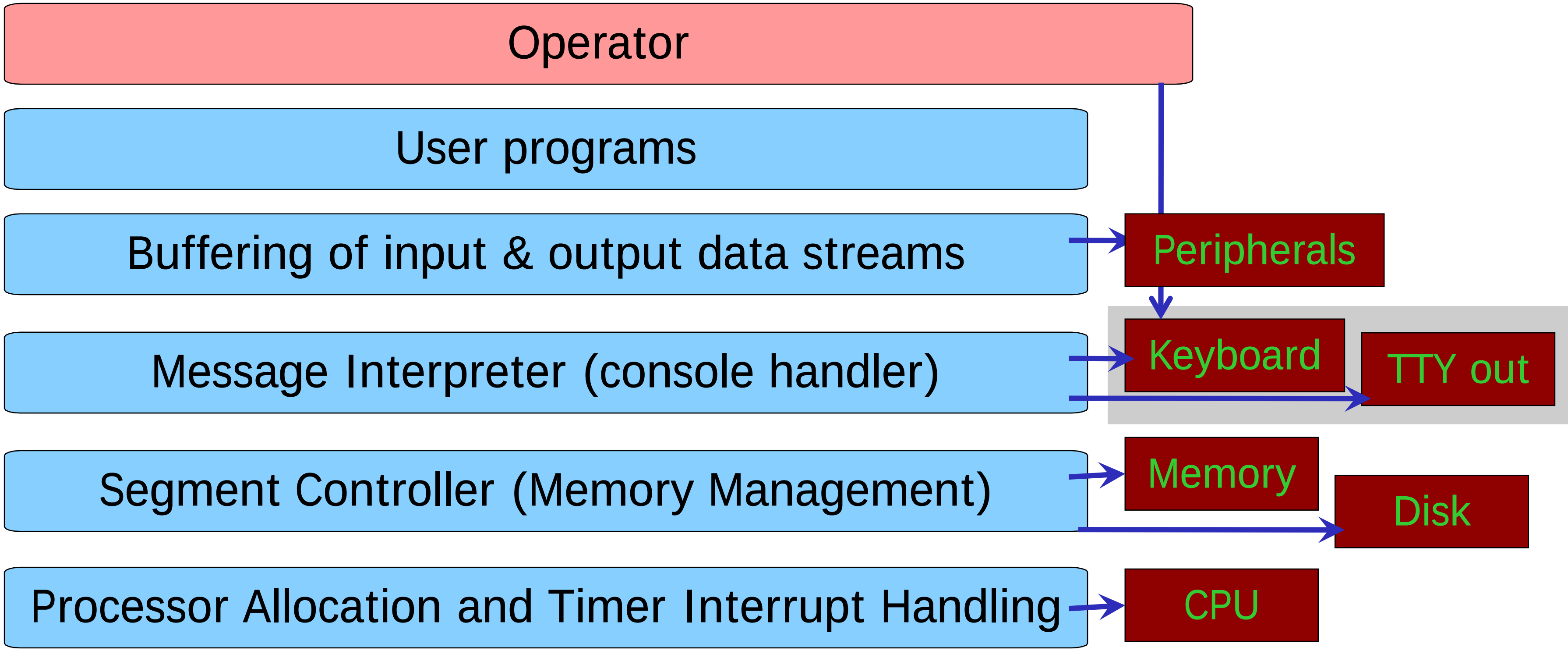


Overview

- Motivation: Why talk about structure?
  - Complexity!
- Kernel structures
  - Layered systems
  - Open systems
  - Monolithic kernels (Probably next week)
  - Microkernels (Also next week)
  - Kernel Extensions (Next week)
  - Virtual Machines (Next week)

Early Layered System: THE

- Djikstra, 1st SOSP, 1967

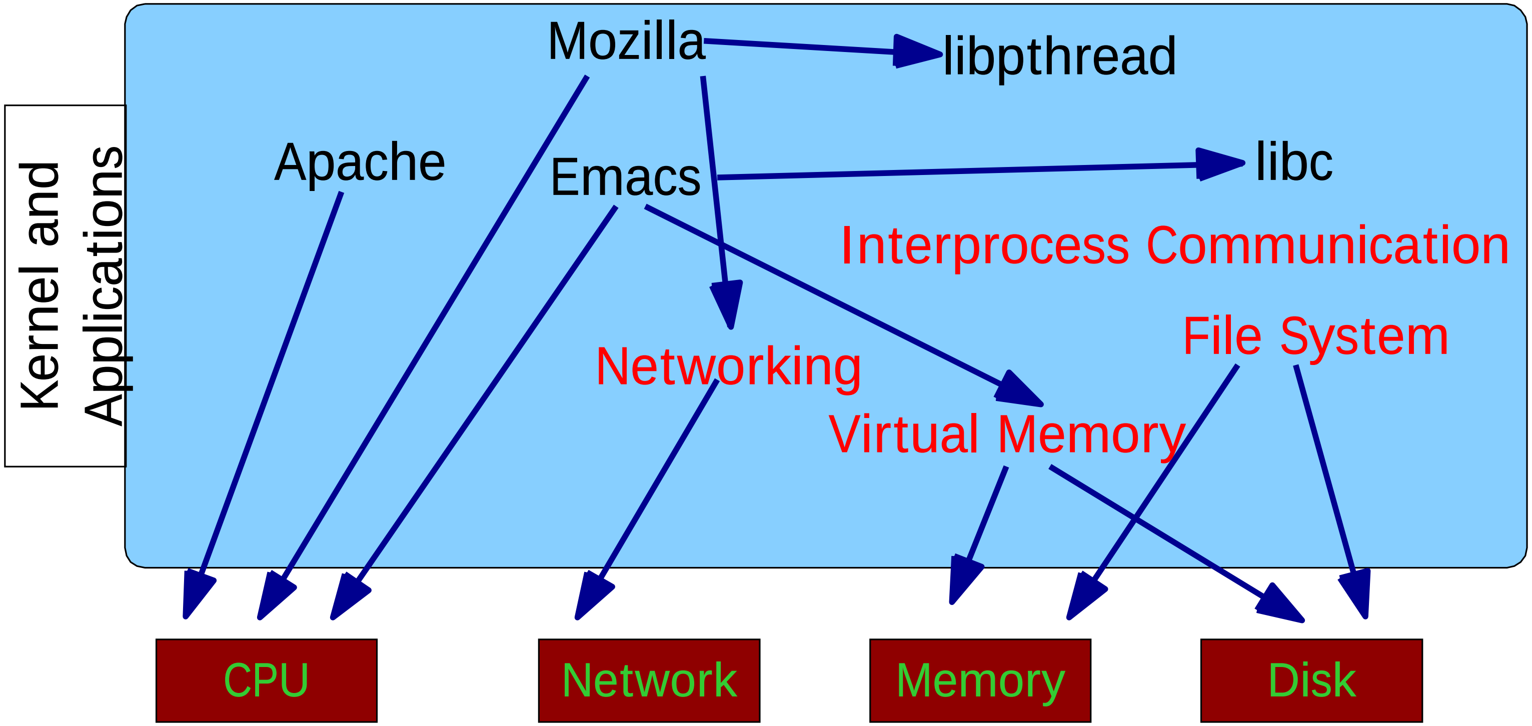


Operator
User programs
Buffering of input & output data streams
Message Interpreter (console handler)
Segment Controller (Memory Management)
Processor Allocation and Timer Interrupt Handling
Peripherals
Keyboard
TTY out
Memory
Disk
CPU

Properties of Layered Systems

- Each layer has well-defined function and interface to layer above/below
  - Provides easier-to-use abstraction for higher layers
- Other examples: MULTICS (rings)
- Advantages?
  - Each layer can be designed, implemented and tested independently
  - Processes at any level can only invoke services of level below → no circular wait → no deadlock
- Disadvantages?
  - Hard to partition functions into this strict hierarchy (why is console below other peripherals?)

Open Systems



Properties of Open Systems

- Applications, libraries, kernel all in the same address space
- Crazy?
  - Idea first described by Lampson & Sproull, 7th SOSP, 1979 “An open operating system for a single-user machine”
  - MS-DOS; Mac OS 9 and earlier; Windows ME, 98, 95, 3.1
  - Palm OS and some embedded systems
- Used to be *very* common
- Advantages?
  - *Very* good performance, very extensible, works well for single-user
- Disadvantages?
  - No protection between kernel and/or apps, not very stable, composing extensions can lead to unpredictable behavior