CS 6410: ADVANCED SYSTEMS

PROF. HAKIM WEATHERSPOON

Fall 2018

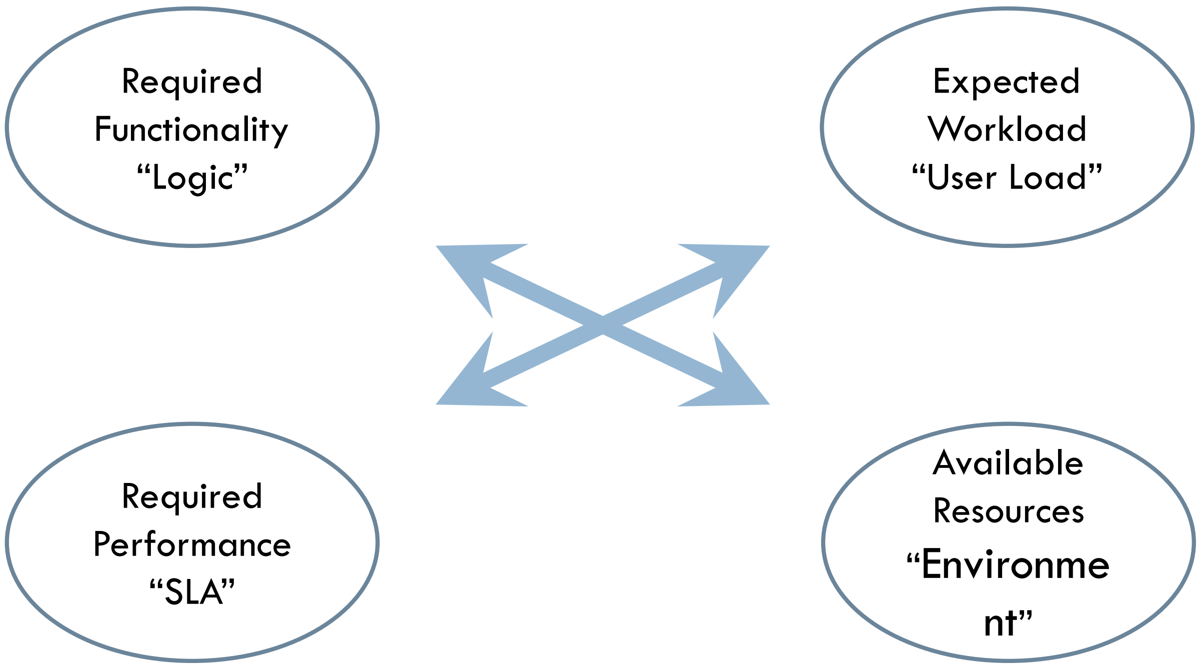
Principled Computer System Design

Systems Research

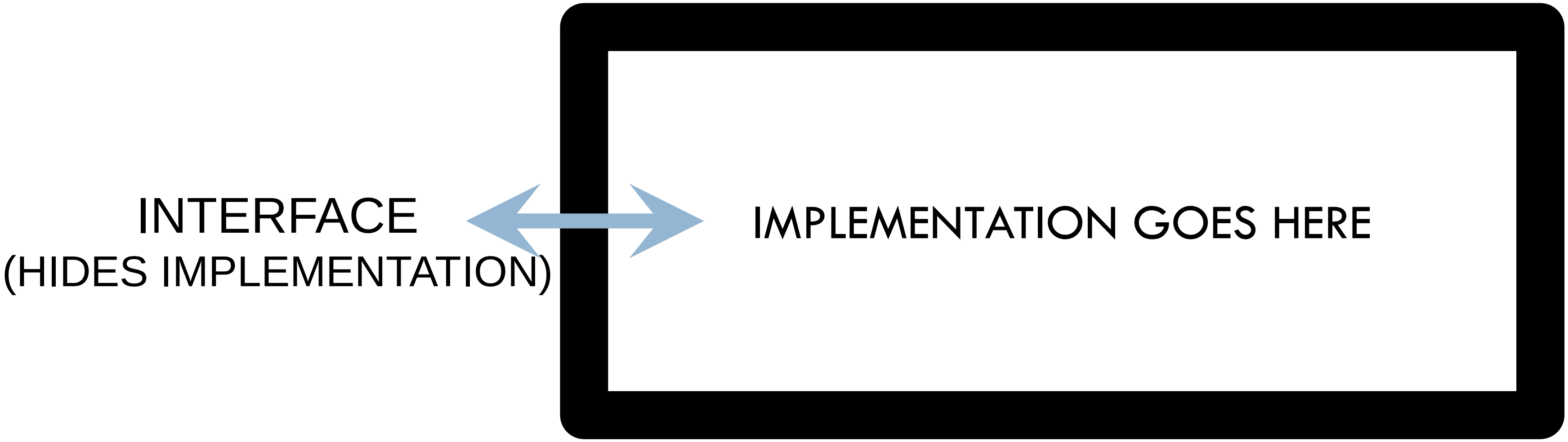
- The study of tradeoffs
  - Functionality vs performance
  - E.g. where to place error checking

- Are there principles or rules of thumb that can help with large systems design?

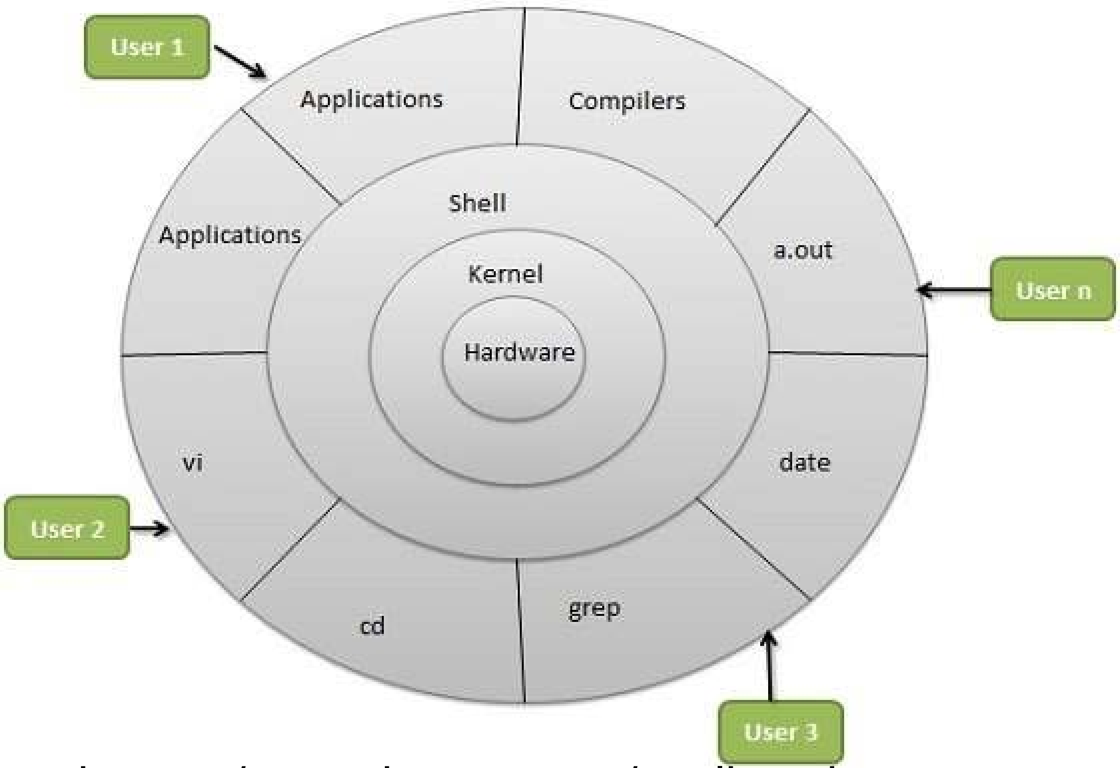
What is System Design: Science, Art, Puzzle?



Something to do with “Abstraction”



Also, “Layering” (layered modules)

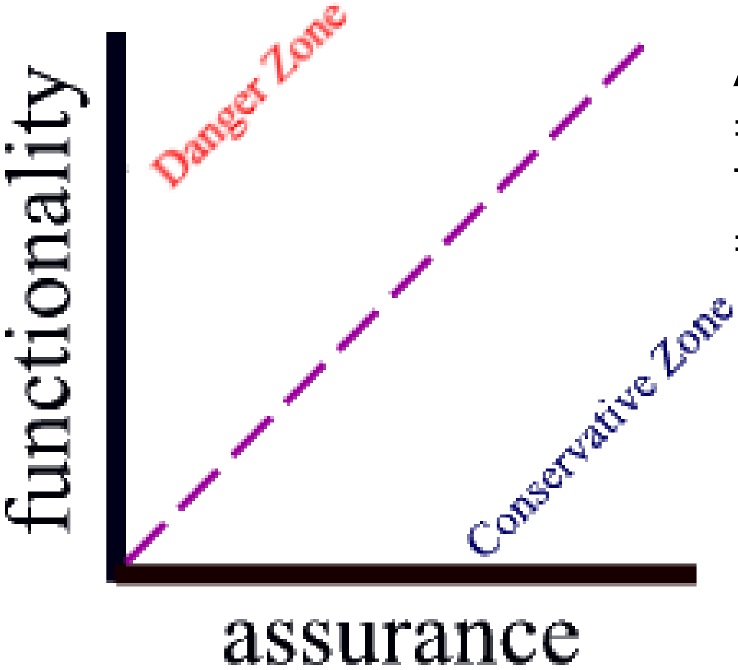


From: http://www.tutorialspoint.com/operating_system/os_linux.htm

Any problem in computer science can be solved with another level of indirection

- Attributed to David Wheeler (by Butler Lampson)

Functionality vs Assurance

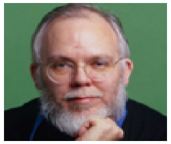


Assurance
== Required Performance (Speed, Fault Tolerance)
== Service Level Agreement (SLA)

End-to-End arguments in System Design –
Jerry H. Saltzer, David P. Reed, David D. Clark (MIT)

- Jerry H. Saltzer
  - A leader of Multics, key developer of the Internet, and a LAN (local area network) ring topology, project Athena

- David P. Reed
  - Early development of TCP/IP, designer of UDP



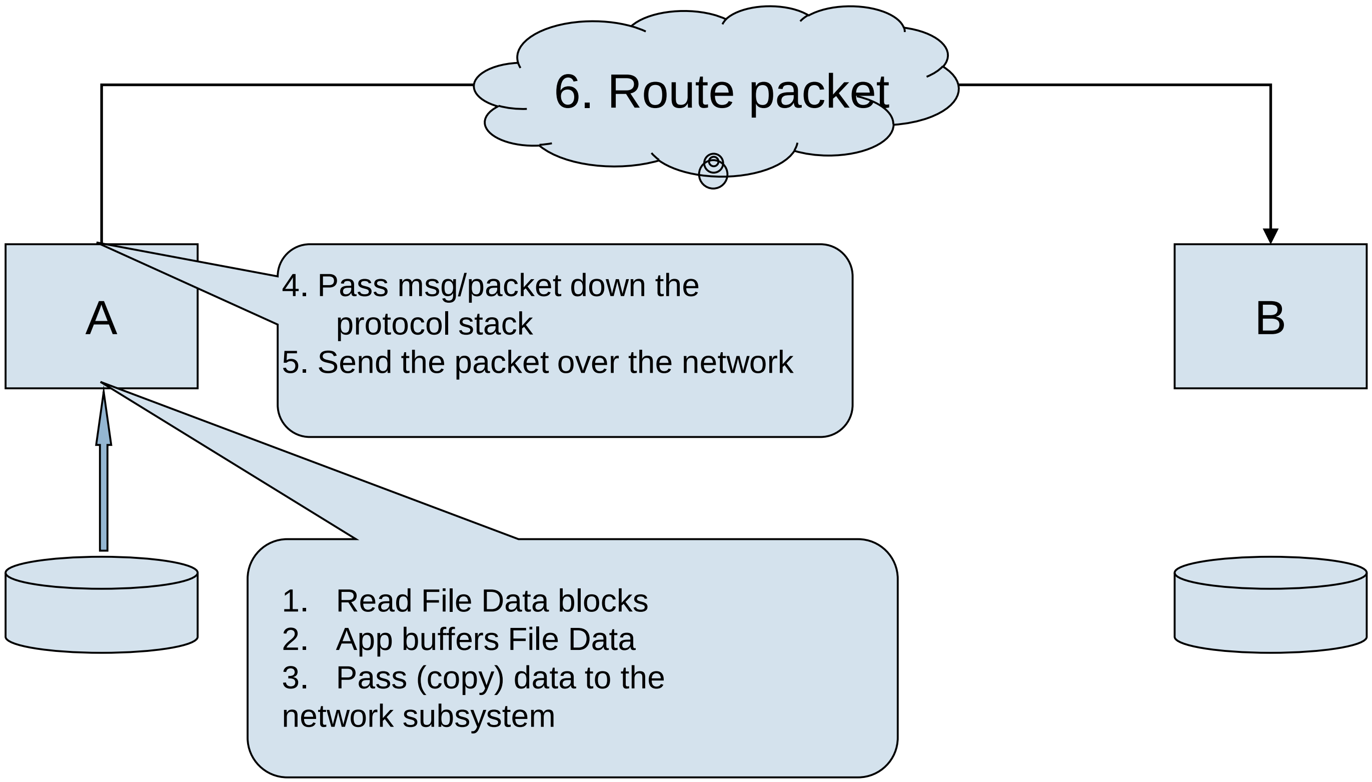
- David D. Clark
  - I/O of Multics, Protocol architect of Internet

    "We reject: kings, presidents and voting.
    We believe in: rough consensus and running code."

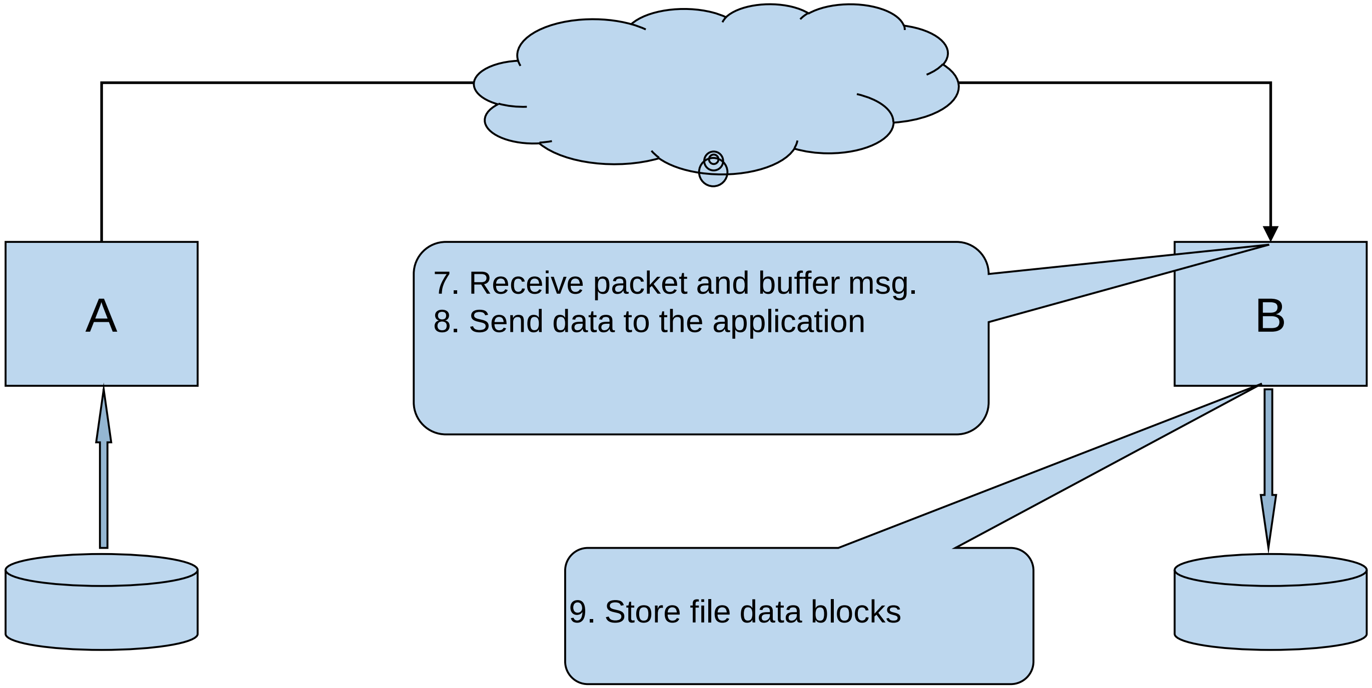
End-to-End argument

- Helps guide function placement among modules of a *distributed* system
- Argument
  - implement the functionality in the lower layer only if
    - a large number of higher layers / applications use this functionality and implementing it at the lower layer improves the *performance* of many of them, AND
    - does not hurt the remaining applications

Example : File Transfer (A to B)



Example : File Transfer (A to B)



Possible failures

- Reading and writing to disk
- Transient errors in the memory chip while buffering and copying
- network might drop packets, modify bits, deliver duplicates
- OS buffer overflow at the sender or the receiver
- Either of the hosts may crash

Solution: make the network reliable?

- Packet checksums, sequence numbers, retry, duplicate elimination
  - Example: TCP
- Solves only the network problem
- What about the other problems listed?
- Not *sufficient* and not *necessary*

Solution: end-to-end retransmission?

- Introduce file checksums and verify once transfer completes – *end-to-end check*.
  - On failure – retransmit file
  - Works!   (modulo rotting bits on disk)

Is network-level reliability useful?

- Per-link retransmission leads to faster recovery from dropped packets than end-to-end
- Seems particularly useful in wireless networks or very high latency networks
- But this may not benefit all applications
  - Huge unnecessary overhead for, say, Real-Time speech transmission

TCP/IP

- Transmission Control Protocol (TCP)
  - It is a transport protocol providing error detection, retransmission, congestion control, and flow control
  - TCP is almost-end-to-almost-end
    - kernel-to-kernel, socket-to-socket, but not app-to-app
- Internet Protocol (IP)
  - IP is a simple ("dumb"), stateless protocol that moves datagrams across the network
  - The network itself (the routers) needs only to support the simple, lightweight IP; the endpoints run the heavier TCP on top of it when needed.

Other end-to-end examples

- End-to-end authentication
  - TLS, SSL
- Duplicate msg suppression

Is argument complete?

- E.g. congestion control
  - TCP leaves it to the ends
    - Should the network trust the ends?
      - RED
    - In a wireless setting
      - packet loss != congestion
- performance problems may appear in end-end systems under heavy load
- Performance enhancing Proxies

“Hints for Computer System Design” --- Butler Lampson, 1983

- Based on author’s experience in systems desig
- Founding member of Xerox PARC (1970)
- Currently Technical Fellow at MSR and adjunct prof. at MIT
- Winner of ACM Turing Award (1994). IEEE Von Neumann Medal (2001)
- Was involved in the design of many famous systems, including databases and networks

Some Projects & Collaborators

- Charles Simonyi - Bravo: WYSIWYG editor (MS Office)
- Bob Sproull - Alto operating system, Dover: laser printer, Interpress: page description language (VP Sun/Oracle)
- Mel Pirtle - 940 project, Berkeley Computer Corp.
- Peter Deutsch - 940 operating system, QSPL: system programming language (founder of Ghostscript)
- Chuck Geschke, Jim Mitchell, Ed Satterthwaite - Mesa: system programming language

Some Projects & Collaborators (cont.)

- Roy Levin - Wildflower: Star workstation prototype, Vesta: software configuration
- Andrew Birrell, Roger Needham, Mike Schroeder - Global name service and authentication
- Eric Schmidt - System models: software configuration (CEO/Chairman of Google/Executive Chairman of Alphabet)
- Rod Burstall - Pebble: polymorphic typed language

System Design Hints organized along two axes: Why and Where

- Why:
  - Functionality: does it work?
  - Speed: is it fast enough?
  - Fault-tolerance: does it keep working?
- Where:
  - Completeness
  - Interface
  - Implementation

Hints for Computer System Design - Butler Lampson

Why?	*Functionality* Does it work?	*Speed* Is it fast enough?	*Fault-tolerance* Does it keep working?
Where?			
Completeness	Separate normal and worst case	Shed load	
		End-to-end	End-to-end
		Safety first	
Interface	Do one thing well:	Make it fast	End-to-end
	Don't generalize	Split resources	Log updates
	Get it right	Static analysis	Make actions atomic
	Don't hide power	Dynamic translation	
	Use procedure arguments		
	Leave it to the client		
	Keep basic interfaces stable		
	Keep a place to stand		
Implementation	Plan to throw one away	Cache answers	Make actions atomic
	Keep secrets	Use hints	Use hints
	Use a good idea again	Use brute force	
	Divide and conquer	Compute in background	
		Batch processing	

Figure 1: Summary of the slogans

FUNCTIONALITY

- Interface
  - Between user and implementation of an abstraction
  - Contract, consisting of a set of assumptions about participants
    - Assume-Guarantees specification
  - Same interface may have multiple implementations
- Requirements:
  - Simple but complete
  - Admit efficient implementation
- Examples: Posix File System Interface, Network Sockets, SQL, …

- Lampson: "Interface is a small programming language"
  - Do we agree with this?

Keep it Simple Stupid (KISS Principle)

- Attributed to aircraft engineer Kelly Johnson (1910—1990)
- Based on observation: systems work best if they are kept simple
- Related:
  - *Make everything as simple as possible, but not simpler* (Einstein)
  - *It seems that perfection is reached not when there is nothing left to add, but when there is nothing left to take away* (Antoine de Saint Exupéry)
  - *If in doubt, leave it out* (Anon.)
  - *Complexity is the Enemy: Exterminate Features* (Charles Thacker)
  - *The unavoidable price of reliability is simplicity* (Tony Hoare)

Do one thing at a time, and do it well
Don't generalize
Get it right!

- A complex interface is hard to implement correctly, efficiently
- Don't penalize all for wishes by just a few
- Basic (fast) operations rather than generic/powerful (slow) ones
- Good interface admits implementation that is
  - Correct
  - Efficient
  - Predictable Performance
- Simple does not imply good
  - A simple but badly designed interface makes it hard to build applications that perform well and/or predictably

Make it Fast
Leave it to the Client
Don’t Hide Power
Keep Secrets

- Design basic interfaces that admit implementations that are fast
  - Consider monolithic O.S. vs. microkernels
- Clients can implement the rest
- Abstraction should hide only undesirable properties
  - What are examples of undesirable?
    - Non-portable
- Don’t tell clients about implementation details they can exploit
  - Leads to non-portability, applications breaking when modules are updated, etc.
  - Bad example: TCP

Use procedure arguments

- High-level functions passed as arguments
  - Requires some kind of interpreter within the abstraction
  - Hard to secure
    - Requires safe language or sandboxing

Keep basic interfaces stable
Keep a place to stand

- Ideally do not change interfaces
  - Extensions are ok
- If you have to change the interface, provide a backward compatibility option
  - Good example: Microsoft Windows

Plan to throw one away
Use a good idea again

- Prototyping is often a good strategy in system design
- You end up building a series of prototypes
- The same good idea may be usable in multiple contexts
- Example: Unix developed this way, leading to Linux, Mac OS X, and several others

Divide and Conquer

- Several forms:
  - Recursion
  - Stepwise Refinement
  - Modularization
- Lampson only talks about recursion
- Stepwise refinement is a useful technique to contain complexity of systems
- Modules contain complexity
  - Principle of "Separation of Concerns" (Edsger Dijkstra)

Handle normal and worst case separately

- Use a highly optimized code path for normal case
- Just try to implement handling the worst case correctly
- Sometimes optimizing normal case hurts worst case performance!
  - And sometimes good worst case performance is more important than optimal normal case performance
- Example: normal case in TCP/IP highly optimized

SPEED

- Lampson talks mostly about making systems *fast*
- Other, perhaps more subtle considerations include
  - Predictable performance
  - Meeting service-level objectives
  - Cheap to run in terms of resources

Split resources
Safety first

- Partitioning may result in better performance than sharing
  - but not always..
    - for example: a shared cache would result in better overall utilization typically than a partitioned cache
    - but a partitioned cache may give more predictable performance to any particular user
  - most low-level resources these days tend to be shared…
- Prioritize safety over optimality

Static analysis
Dynamic translation

- No, this is not a PL course
- If you know something about the workload, exploit it!
  - For example, workload might exhibit locality, periodicity, etc.
  - Related to "normal case" handling
- Prefetching allows I/O and compute to overlap
- Examples: paging and scheduling algorithms

Cache answers
Use hints

- Caching answers to expensive computations trades storage for other resources (CPU, network, etc.)
  - What does “expensive” mean in this context?
- “Hints” are typically caches of potentially wrong information
  - Example: DNS uses this extensively to provide scalability
  - Should be easy to check if hint works, and correct for it if not

When in doubt, use brute force

- Related idea: don't optimize blindly
  1. build the system "stupidly"
  2. identify bottlenecks through profiling
  3. eliminate bottlenecks
  4. go back to Step 2 if necessary
- If the system is modular, such "adjustments" are typically easy to make
  - If not, difficult refactoring might be necessary
  - Related: building series of prototypes

Compute in background
Use batch processing
Shed load

- "Compute in background" essentially means to do I/O and compute in parallel
  - examples: paging, GC, …
  - in this day and age, we do everything in parallel…
- Batching multiple small jobs into a larger one can significantly improve throughput
  - although often at the expense of latency
  - example: TCP
- Avoid overload by admission control
  - example: TCP

Fault Tolerance

- We expect 24x7x365.25 reliability these days
- In spite of what Lampson says, it's pretty hard…

Log updates
Make actions atomic or restartable

- Cheap: many storage devices optimal or optimized for append-only
- Useful: after a crash, state can be restored by replaying log
  - helps if updates are "idempotent" or restartable
  - example: ARIES "WAL" (Write-Ahead Log)
- Atomic (trans-)actions simplify reliable system design
  - group of low-level operations that either complete as a unit or have no effect
- Isolation and Durability are also very useful properties!

Concrete conclusions?

- Lessons Learned
  - Pose your problem in a clean way
  - Next decompose into large-scale components
  - Think about the common case that will determine performance: the critical path or the bottleneck points
  - Look for elegant ways to simultaneously offer structural clarity and yet still offer fantastic performance
- This can guide you towards very high-impact success

Next Time

- Read and write review:
  - *The UNIX time-sharing system*, Dennis M. Ritchie and Ken Thompson. Communications of the ACM Volume 17, Issue 7, July 1974, pages 365 – 375

    http://dl.acm.org/citation.cfm?id=361011.361061

  - *The structure of the "THE"-multiprogramming system*, E.W. Dijkstra. Communications of the ACM Volume 11, Issue 5, May 1968, pages 341—346

    http://dl.acm.org/citation.cfm?id=363143

    - Need to be on campus, or use VPN to access some papers. Or, change ".acm.org/" to ".acm.org.proxy.library.cornell.edu/" in the URL

- Check website for updated schedule

Before Next time

- Rank-order papers to present
- Read first papers below and write review
  - The UNIX time-sharing system, Dennis M. Ritchie and Ken Thompson. Communications of the ACM Volume 17, Issue 7 (July 1974), pages 365--375.
  - The structure of the "THE"-multiprogramming system, E.W. Dijkstra. Communications of the ACM Volume 11, Issue 5 (May 1968), pages 341--346.
- Miniproject0
  - Using Amazon's EC2/S3 infrastructure
- Check website for updated schedule