15-712 Introduction

David Andersen

Today's Topics

- Waitlist status
- Course requirements
- Background you should have
- Topics that will be covered
- Things we're leaving out!
- Overview: what, why, how it fits together

Waitlist

- As of today: registered, on waitlist
- Course limit: 24 (project face-time limit!)
- Admittance priority: CSD Ph.D., ECE Ph.D., CSD 5th year masters
  - Others should check with dga
  - Attendance during first few classes counts. :)
- If you're not going to take class, please drop so we know the real # of students

Course Schedule

- Officially: MWF 10:30 - 12:50
- Real: average two days per week
  - More MWF at the start of the semester
  - More M towards the end (biasing for projects)
  - But will vary!
- Please pay attention to the online schedule

Course Requirements

- This is a paper-reading course
  - Attendance & Participation
  - Short summaries of papers (1/4 - 1/2 pg)
    - 3 most important things in paper
    - Describe biggest weakness
    - Describe concl. about how to build systems
- With two "midterm" exams
  - Questions about "important parts" of papers, e.g. "How do the consistency semantics of system X differ from system Y? Why?"
  - Questions about important concepts, e.g., "Briefly explain a scenario in which you would want to use optimistic concurrency"

Project

- 40% of course grade
- Groups of 3
- Goal: Real, original systems research
  - Ideal goal: publishable with some work after semester
- Milestones: Proposal, background research update, mid-semester status report, final presentation, final paper
- Several one-on-one status meetings during semester (times TBA)
- Get started early! Find groups early! Talk to us about ideas!
- See list of prior years projects for additional ideas
  - Our ideas list will be updated over next week or two
  - Proposals due early October

Background

- Assume familiarity with 15-410 / 15-441 material
  - OS organization, threads, paging, basics of concurrency, basic TCP/IP, packets, Ethernet, etc.
- Question: Do people want an OS/networks review session? What's the general background? What areas are people in now? (Why are you taking the class? :)
  - Have you taken 15-744 or other grad-level systems course?
  - (Only happens if we get a TA)

Topics

- (Not a complete set of everything important! Far more than one semester.)
- Concurrency, threads, parallelism
  - Becoming critical again with multi-core, massively parallel storage & processing systems, google/MSN/etc.
- Storage, local and distributed
- Briefly: OS architecture and extensibility
- Transactions and Databases
- Fault Tolerance
- Security

Not Enough Time for...

- Each of these could be/is a class on its own:
- Lots of networking detail (take 15-744. :)
- Traditional high-performance computing (HPC) ("supercomputer center stuff")
  - But we'll hit on the more systems-y side of things
- Compiler and languages meet systems
  - e.g., much on language-based safety, etc. (except SPIN)
- Formal methods meet systems
  - Verification starting to become practical (e.g., Engler)
- Much deep, formal theory (see Nancy Lynch book)
- Systems + architecture (*tons* of interesting stuff!)

Why are Systems Fun?

- Basic computer system:
  - Processing, memory, storage, network (picture)
  - Could build a basic, functioning prototype of those in a few undergrad classes
- Now, make it real:
  - High-performance, Scalable, Reliable, Useable, Extendable, Secure, Easy to manage, *Useful*
  - That's where you start having to use your melon
  - (Lampson: functionality, speed, and fault-tolerance)

Abstraction & Interface

- clients access an implementation using an interface
- Often the biggest part of good systems design.
- Conflicting goals: simple, complete, admit efficient impl.
  - ... and you have a nearly unbounded universe to draw from
- Things to think about: Orthogonal, "KISS", "DRY", small, consistent, atomic, stable, "don't hide power", "leave it to the client"
  - Common wrenches: Failure handling
- Strunk & White: "Omit needless words"; Tufte: "Omit needless lines!" -> "Omit needless features!"

Things to ask yourself

- Why this interface? Why do {you, the authors, someone} think the abstractions proposed in a paper are the right ones?
  - Are they easier to use?
    - More consistent? Making a better trade between system complexity and client complexity?
  - Do they admit more efficient implementations?
    - Do you care and need the efficiency?
  - Do they let you do more with less? e.g., are they more powerful?
- Example: fork(); exec("prog"); vs spawn("prog");
  - (consider: execve, execvp; need spawnve, spawnvp, too?)
  - what about embedded systems with no MMU for CoW?

Speed

- Do you need it? Are you sure?
  - Sometimes the answer is yes!
  - But sometimes it should be secondary to other concerns
    - Will you need it next year?
- Parallelism is coming back into vogue
  - It's done this a few times before (Connection Machine, SMP, CC-NUMA supercomputers, vector supercomputers...)
  - But parallelism is *fundamental to large systems*.
    - We live in a parallel world; many resources exist in multiples
- Always, Amdahl's Law; and don't optimize idle loop

Fault Tolerance

- *Where* do you do fault recovery?
  - End-to-end at the application
    - Absolutely necessary. (Saltzer - end-to-end arguments in system design)
  - Where else? Depends on performance vs. complexity trade-offs
    - Nice example: 802.11 wireless ethernet
- Implies:
  - Expose failures; simple, atomic operations that are easier to recover
  - *Understand* why a system should be correct (or not)

Concurrency

- Affects reliability (correctness) and performance
  - Must be able to harness parallelism for speed
    - Today's multi-core systems; tomorrow's 128-core systems
    - "DISC" style ("Data-intensive supercomputer") systems, e.g., massive cluster farms
    - Distributed systems like Akamai that put data closer to users
    - How? Threads and other concurrency models
  - And must be able to do so *correctly*
    - Theory: Lamport clocks, locking models, transactions, distributed algorithms (leader election, byzantine, etc.)
    - Verification (Eraser, etc.)
- Large, complex systems are fundamentally concurrent!

Storage

- Much of computing operates on data...
  - Have to store it somewhere! :)
- What makes it interesting?
  - Unreliable components (disks fail surprisingly often)
  - "Interesting" access characteristics (high seek latency, but fairly high bulk throughput)
  - Limited sizes/speeds of individual *physical* device => often necessary to use many devices together

Storage: Single Disk

- Coping with a single disk's behavior
  - Big challenge: synchronous (safe) writes: Must be able to know when data is persistent (e.g., databases, accounting, etc.)
    - Log-structured filesystem (allows more sequential writes)
    - Write all modifications to the end of large, sequential "log"
    - No other structures on disk.
    - Ideas from LFS found in many modern filesystems

Storage: Multi-disk

- Reliability and limited size:
  - RAID ("Redundant Array of In{expensive,dependent} Disks")
  - Techniques for crafting larger logical device from many smaller devices
    - Increase speed, reliability, or both
- Getting even bigger:
  - Cluster filesystems.
    - Common goal: scale by adding another "unit" (computer +disk). Term: "self-*" (ganger). Self-managing, self-healing, self-tuning, etc.
    - "Total Cost of Ownership" a big focus
    - Crazy high performance another

Storage System Example

- NetApp FAS6280
  - Multi-rack (up to 6) systems
  - 1,440 disk drives, 192GB of memory
  - 8 onboard 10gigE ports, can go up to 40, ...
  - 2,880TB of storage (this year)
  - Dual-parity RAID
  - Copy-on-write snapshots to ease backups
  - Lots of stuff to make admin life easy (e.g., it phones home and asks for replacement hard drives without bugging you)
  - $$$$$, consumes kilowatts of power

Storage: Distributed

- Ease of use:
  - One copy of data, available anywhere, anytime, fast
  - Examples: AFS
- We're not going to touch _too_ much on the wide-area versions this semester
  - High-performance, read-write, redundant, wide-area filesystems are still a research challenge -- and fundamentally hard.

Transactions & DBs

- Major issue: Concurrency
  - DBs are most often disk-seek limited
  - Can greatly increase throughput by overlapping computation & IO
- Transactions: Usability & Correctness
  - (And help with concurrency. :)
  - Allow atomic sets of operations - all succeed or fail together
- Recovery and Logging
  - Keeping the DB in good state (despite being left in weird state b/c of on-going operations during a crash)

Trans. & DBs, 2

- Worth noting: The major differences between "entry-level" DBs and huge ones are the factors we discussed earlier:
  - Better query optimization (speed)
  - Better scalability
    - More ability to handle concurrency
    - Clever implementations
  - Rollback and replication (correctness & safety)
  - etc.

Fault Tolerance

- Bringing in the theory: Guarantees of surviving particular types of faults (correctness)
- Byzantine fault tolerance:
  - *Arbitrary* failures, including malicious or those designed to break system/protocol
  - How can you design a system robust to byzantine failures of a certain number of components?
  - Typically involve voting/agreement protocols
- Fault containment: Preventing the effects of faults from spreading
  - Earlier focus: Hive (cell to cell multiprocessor)
  - Modern systems: Nooks (device drivers to kernel)
  - Future: Highly multi-core chips (back to the future...)

Security

- The need for security should be clear. If not, put an unpatched Windows box on CMU's network for 5 minutes...
- From the theory
  - Access control; authentication mechanisms
  - Protocols, cryptographic and otherwise
- To isolation mechanisms and secure foundations
  - Modern habit: Throw things in tiny virtual machines
  - Trusted Computing Platform: hardware crypto attestation

Philosophy

- Even Lamport notes that simplicity, efficiency, consistency, completeness *conflict*.
- "The Perfect is the Enemy of the Good" -- Voltaire, via Jay Lepreau
  - (But make sure it's still *good*. :)
  - A working system is better than a non-working one
  - A good system on time beats a perfect one that's 5 years late and 3x over budget
- How to balance practicality and ideals?
- ("A Witty Saying Proves Nothing." -- Voltaire again)

Worse is Better (kinda)

- Simplicity: Still good. Both implementation & interface.
  - Simple interface: Easy to build to
  - Simple implementation: Easy to build (think about both customers)
- Correctness: Correct in all observable aspects
  - In impl: sometimes it's better to leave a known bug in than fix it and introduce an unknown bug. Bizarre, eh? Big companies make this judgement every day. And they're often right. (Of course, their systems are probably too complex to start with. This happens *often* in windows...)
- Consistency: Keep things mostly consistent, particularly in implementation. Interface? Meh.
- Completeness: *Cover as many important situations as is practical.* Kick completeness if you need to for simplicity!

- 80/20 rule: The last 20% takes 80% of the effort. So don't bother at the start.
- Consider:
  - UNIX
  - C
  - Perl, Ruby, Python
- IETF motto: "Rough Consensus and Working Code"
  - Worked pretty well for the Internet
- Iterative/spiral design, XP, etc: All attest to practicality of having something working quickly

What's that all mean?

- Many systems have a few "neat" things
  - e.g., databases have cool concurrency models
    - But aren't known for innovation in security protocols. :)
  - OSes might have neat schedulers
    - But a ton of very ordinary hash tables and linked lists
- Pick the innovations that *matter* to your system and domain
  - And keep everything else as simple as possible until proven otherwise
- This class: Examining powerful techniques in particular areas
  - Note how and when to use them, not just what they are
  - Most important: Examine spectrum of research and learn both systems principles, useful techniques, and *how to do systems research* and develop & evaluate new systems.