Intro Lecture

I. Administrative Matters

Instructors: Eric Brewer and Joe Hellerstien

Eric Brewer

- PhD MIT, 1994
- Internet Systems, Mobile computing, Security, Parallel computing
- Founded Inktomi, Federal Search Foundation
- Contact: brewer@cs.berkeley.edu

Joe Hellerstien

- PhD, Wisconsin, 1995
- Databases, declarative networking
- Contact: hellerstein@cs.berkeley.edu

Course prerequisite: no **entrance exam** this year, but please review undergrad material

Traditional goals of the course:

- Introduce a variety of current OS research topics
- Teach you how to do OS research

New goals:

- Common foundation for OS and database research
- Motivation: OS and DB communities historically separate, despite much overlap in goals and ideas. We often use independent vocabulary and have largely separate "classic" papers that the other community typically hasn't read (especially OS people not reading DB papers, since a DB is "just an application")
- Part 1 of a year-long advanced systems course
- 262A satifies software breadth requirement by itself

Research = analysis & synthesis

- Analysis: understanding others' work - both what's good AND what's bad.
- Systems research isn't cut-and-dried: few "provably correct" answers.
- Synthesis: finding new ways to advance the state of the art.

Lecture 1

2 parts to course:

- literature survey
- term project

Will not cover basic material.

Analysis: literature survey: read, analyze, criticize papers.

- All research projects fail in some way.
- Successful projects: got some interesting results anyway.

In class: lecture format with discussion

Synthesis: do a small piece of real research.

- Suggested projects handed out in 3-4 weeks
- Teams of 2-3
- Poster session and "conference paper" at the end of the semester
- Usually best papers make it into a real conference (with extra work)

Class preparation:

- Reading papers is hard, esp. at first.
- **Read before class.**

Homework: *brief* summary/paper

- 1/2 page or less
- 2 most important things in paper
- 1 major flaw

Course topics: (much of systems design is about sharing)

- File systems
- Virtual memory
- Concurrency, scheduling, synchronization
- Communication
- Multiprocessors
- Distributed Systems
- Transactions, recovery, & fault-tolerance

Lecture 1

- Protection & security
- OS structure
- "Revealed truth" — overall principles

Grad class ⇒ material may be controversial (hopefully!)

Lecture format: cover 1-2 papers plus announcements

Grading: 50% project paper, 15% project demo, 25% midterm, 10% paper summaries

- can miss 3 summaries without penalty, no reason needed or wanted
- Summaries: < 1/2 page, at least one criticism, summary should be at most half

Reading list:

- No textbook
- Almost everything is online, otherwise I'll distribute copies in class.
- "Warm up" paper: "UNIX timesharing"
- Reading for next time: System R paper (in the Red Book)

II. The UNIX Time-Sharing System

"Warm-up" paper. Material should mostly be a review.

Classic system and paper: described almost entirely in 10 pages.

Key idea: elegant combination of a few concepts that fit together well.

System features:

- time-sharing system
- hierarchical file system
- device-independent I/O
- shell-based, tty user interface
- filter-based, record-less processing paradigm

Version 3 Unix:

- ran on PDP-11's
- < 50 KB
- 2 man-years to write
- written in C

Lecture 1

File System:

- ordinary files (uninterpreted)
- directories (protected ordinary files)
- special files (I/O)

Directories:

- root directory
- path names
- rooted tree
- current working directory
- back link to parent
- multiple links to ordinary files

Special files:

- uniform I/O model
- uniform naming and protection model

Removable file systems:

- tree-structured
- *mount*'ed on an ordinary file

Protection:

- user-world, RWX bits
- set-user-id bit
- super user is just special user id

Uniform I/O model:

- open, close, read, write, seek
- other system calls
- bytes, no records

File system implementation:

- table of i-nodes
- path name scanning

Lecture 1

- mount table
- buffered data
- write-behind

I-node table:

- short, unique name that points at file info.
- allows simple & efficient fsck
- can't handle accounting issues

Processes and images:

- text, data & stack segments
- process swapping
- pid = fork()
- pipes
- exec(file, arg1, ..., argn)
- pid = wait()
- exit(status)

The Shell:

- cmd arg1 ... argn
- stdio & I/O redirection
- filters & pipes
- multi-tasking from a single shell
- shell is just a program

Traps:

- hardware interrupts
- software signals
- trap to system routine

Advanced Topics in Computer Systems **Lecture 2**

Edited by Eric Brewer based on notes by Joe Hellerstein

System R & DBMS Overview

DBMS History

- late 60's: network (CODASYL) & hierarchical (IMS) DBMS. Charles Bachman: father of CODASYL predecessor IDS (at GE in early 1960's). Turing award #8 (1973, between Dijkstra and Knuth.)
  - IMS Example: *Suppliers* record type and *Parts* record type. One is parent, one is child. Problems include redundancy and requirement of having a parent (deletion anomalies.)
  - Low-level ``record-at-a-time'' data manipulation language (DML), i.e. physical data structures reflected in DML (no data independence).

- 1970: Codd's paper. The most influential paper in DB research. Set-at-a-time DML with the key idea of "data independence". Allows for schema and physical storage structures to change under the covers. Papadimitriou: "as clear a paradigm shift as we can hope to find in computer science"). Edgar F. Codd: Turing award #18 (1981, between Hoare and Cook).
  - *Data Independence*, both logical and physical.
  - What physical tricks could you play under the covers? Think about modern HW!
  - "Hellerstein's Inequality":
    - Need data independence when *d*app/*dt* << *d*environment/*dt*
    - Other scenarios where this holds?
    - This is an early, powerful instance of two themes: *levels of indirection* and *adaptivity*
- mid 70's: wholesale adoption of Codd's vision in 2 full-function (sort of) prototypes. Ancestors of essentially all today's commercial systems
  - Ingres : UCB 1974-77
    - a ``pickup team'', including Stonebraker & Wong. early and pioneering. Begat Ingres Corp (CA), CA-Universe, Britton-Lee, Sybase, MS SQL Server, Wang's PACE, Tandem Non-Stop SQL.
  - System R : IBM San Jose (now Almaden)
    - 15 PhDs. Begat IBM's SQL/DS & DB2, Oracle, HP's Allbase, Tandem Non-Stop SQL. System R arguably got more stuff ``right'', though there was lots of information passing between both groups
    - Jim Gray: Turing Award #22 (1998, between Englebart and Brooks)
    - Lots of Berkeley folks on the System R team, including Gray (1st CS PhD @ Berkeley), Bruce Lindsay, Irv Traiger, Paul McJones, Mike Blasgen, Mario Schkolnick, Bob Selinger , Bob Yost. See http://www.mcjones.org/System_R/SQL_Reunion_95/sqlr95-Prehisto.html#Index71.
  - Both were viable starting points, proved practicality of relational approach. Direct example of theory -> practice!
    - ACM Software Systems award #6 shared by both

        - Stated goal of both systems was to take Codd's theory and turn it into a workable system as fast as CODASYL but much easier to use and maintain
        - Interestingly, Stonebraker received ACM SIGMOD Innovations Award #1 (1991), Gray #2 (1992), whereas Gray got the Turing first.
- early 80's: commercialization of relational systems
    - Ellison's Oracle beats IBM to market by reading white papers.
    - IBM releases multiple RDBMSs, settles down to DB2. Gray (System R), Jerry Held (Ingres) and others join Tandem (Non-Stop SQL), Kapali Eswaran starts EsVal, which begets HP Allbase and Cullinet
    - Relational Technology Inc (Ingres Corp), Britton-Lee/Sybase, Wang PACE grow out of Ingres group
    - CA releases CA-Universe, a commercialization of Ingres
    - Informix started by Cal alum Roger Sippl (no pedigree to research).
    - Teradata started by some Cal Tech alums, based on proprietary networking technology (no pedigree to software research, though see parallel DBMS discussion later in semester!)
- mid 80's: SQL becomes "intergalactic standard''.
    - DB2 becomes IBM's flagship product.
    - IMS "sunseted''
- today: network & hierarchical are legacy systems (though commonly in use!)
    - IMS still widely used in banking, airline reservations, etc. A cash cow for IBM
    - Relational commoditized -- Microsoft, Oracle and IBM fighting over bulk of market. NCR Teradata, Sybase, HP Nonstop and a few others vying to survive on the fringes. OpenSource coming of age, including MySQL, PostgreSQL, Ingres (reborn). BerkeleyDB is an embedded transactional store that is widely used as well, but now owned by Oracle.
    - XML and object-oriented features have pervaded the relational products as both interfaces and data types, further complicating the "purity" of Codd's vision.

Database View of Applications

Big, complex record-keeping applications like SAP and PeopleSoft, which run *over* a DBMS. "Enterprise applications" to keep businesses humming. A smattering:

- ERP: Enterprise Resource Planning (SAP, Baan, PeopleSoft, Oracle, IBM, etc.)
- CRM: Customer Relationship Management (E.phiphany, Siebel, Oracle, IBM, etc.)
- SCM: Supply Chain Management (Trilogy, i2, Oracle, IBM, etc.)
- Human Resources, Direct Marketing, Call Center, Sales Force Automation, Help Desk, Catalog Management, etc.

Typically client-server (a Sybase "invention") with a form-based API. Focus on resource management secondary to focus on data management.

Traditionally, a main job of a DBMS is to make these kinds of apps easy to write

Relational System Architecture

See the article in Foundations and Trends in Databases for in-depth disussion of many of the issues below. Databases are BIG pieces of software. Typically somewhat hard to modularize. Lots of system design decisions at the macro and micro scale. We will focus mostly on micro decisions -- and hence ideas reusable outside DBMSs -- in subsequent lectures. Here we focus on macro design.

Disk management choices:

- file per relation
- big file in file system
- raw device

Process Model:

- process per user
- server
- multi-server

Hardware Model:

- shared nothing
- shared memory
- shared disk

Basic modules:

- parser
- query rewrite
- optimizer
- query executor
- access methods
- buffer manager
- lock manager
- log/recovery manager

Notes on System R

See the System R reunion notes for fun background and gossip.

Some "systems chestnuts" seen in this paper:

- Expect to throw out the 1st version of the system
- Expose internals via standard external interfaces whenever possible (e.g. catalogs as tables, the /proc filesystem, etc.)
- Optimize the fast path
- Interpretation vs. compilation vs. intermediate "opcode" representations
- Component failure as a common case to consider
- Problems arising from interactions between replicated functionality (in this case, scheduling)

Some important points of discussion

- Flexibility of storage mechanisms: domains/inversions vs. heap-files/indexes. Use of TID-lists common in modern DBMS. Why be doctrinaire? What about Data Independence? One answer: you have to get transactions right for each "access method".
- System R was often CPU bound (though that's a coarse-grained assertion -- really means NOT disk-bound). This is common today in well-provisioned DBMSs as well. Why?
- DBMSs are not monolithic designs, really. The RSS stuff does intertwine locking and logging into disk access, indexing and buffer management. But RDS/RSS boundary is clean, and RDS is decomposable.
- Access control via views: a deep application of data independence?!
- Transactional contribution of System R (both conceptual and implementation) as important as relational model, and in fact should be decoupled from relational model.

The "Convoy Problem":

- A classic cross-level scheduling interaction. We will see this again!
- Poorly explained in the paper.

  I have always found this presentation confusing. A number of issues are going on. The first two have to do with interactions between OS and DB scheduling:
  a. the OS can preempt a database "process" even when that process is holding a high-traffic DB lock
  b. DB processes sitting in DB lock queues use up their OS scheduling quanta while waiting (this is poorly explained in the text). Once they use up all their quanta, they get removed from the "multiprogramming set" and go to "sleep" -- and an expensive OS dispatch is required to run them again.

  The last issue is that
  c. the DBMS uses a FCFS wait queue for the lock.

  For a high-traffic DB lock, DB processes will request it on average every T timesteps. If the OS preempts a DB process holding that high-traffic DB lock, the queue behind the lock grows to include almost all DB processes. Moreover, the queue is too long to be drained in T timesteps, so it's "stable" -- every DB process queues back up before the queue drains, and they burn up their quanta pointlessly waiting in line, after which they are sent to sleep. Hence each DB process is awake for only one grant of the lock and the subsequent T timesteps of useful work, after which they queue for the lock again, waste their quanta in the queue, and are put back to sleep. The result is that the useful work per OS waking period is about T timesteps, which is shorter than the overhead of scheduling -- hence the system is thrashing.
- Note that the solution attacks the only issue in the previous comment that can be handled without interating with the OS: (c) the FCFS DB lock queue. The explanation here is confusing, I think. The point is to always allow any one of the DB processes currently in the "multiprogramming set" to immediately get the lock without burning a quantum waiting on the lock -- hence no quanta are wasted on waiting, so each process spends almost all of its alloted quanta on "real work". Note that the proposed policy achieves this without needing to know which processes are in the OS' multiprogramming set.

System R and INGRES are the prototypes that all current systems are based on. Basic architecture is the same, and many of the ideas remain in today's systems:

- optimizer remains, largely unchanged
- RSS/RDS divide remains in many systems
- SQL, cursors, duplicates, NULLs, etc.
  - the pros and cons of duplicates. Alternatives?
  - pros and cons of NULLs. Alternatives?
  - grouping and aggregation
- updatable single-table views
- begin/end xact at user level
- savepoints and restore
- catalogs as relations
- flexible security (GRANT/REVOKE)
- integrity constraints
- triggers (!!)
- clustering
- compiled queries
- B-trees
- Nest-loop & sort-merge join, all joins 2-way
- dual logs to support log failure

Stuff they got wrong:

- shadow paging
- predicate locking
- SQL language
  - duplicate semantics
  - subqueries vs. joins
  - outer join
- rejected hashing

OS and DBMS: Philosophical Similarities & Differences

- UNIX paper: "The most important job of UNIX is to provide a file system".
  - UNIX and System R are both "information management" systems!
  - both also provide programming APIs for code
- Difference in focus: Bottom-Up (elegance of system) vs. Top-Down (elegance of semantics)
  - main goal of UNIX was to provide a small *elegant* set of mechanisms, and have programmers (i.e. C programmers) build on top of it. As an example, they are proud that "No large 'access method' routines are required to insulate the programmer from system calls". After all, OS viewed its role as *presenting hardware to computer programmers.*
  - main goal of System R and Ingres was to provide a complete system that insulated programmers (i.e. SQL + scripting) from the system, while guaranteeing clearly defined *semantics* of data and queries. After all, DBMS views its role as *managing data for application programmers.*

    - Affects where the complexity goes!
        - to the system, or the end-programmer?
        - question: which is better? in what environments?
        - follow-on question: are internet systems more like enterprise apps (traditionally built on DBMSs) or scientific/end-user apps (traditionally built over OSes and files)? Why?
- Achilles' heel of RDBMSs: a closed box
    - Cannot leverage technology without going through the full SQL stack
    - One solution: make the system extensible, convince the world to download code into the DBMS
    - Another solution: componentize the system (hard? RSS is hard to bust up, due to transaction semantics)
- Achilles' heel of OSes: hard to decide on the "right" level of abstraction
    - As we'll read, many UNIX abstractions (e.g. virtual memory) hide *too* much detail, messing up semantics. On the other hand, too low a level can cause too much programmer burden, and messes up the elegance of the system
    - One solution: make the system extensible, convince the fancy apps to download code into the OS
    - Another solution: componentize the system (hard, due to protection issues)
- Traditionally separate communities, despite subsequently clear need to integrate
    - UNIX paper: "We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes."
    - System R: "has illustrated the feasibility of compiling a very high-level data sublanguage, SQL, into machine-level code".

So, a main goal of this class is to work from both of these directions, cull the lessons from each, and ask how to use these lessons today both within and OUTSIDE the context of these historically separate systems.

Lecture Notes

UNIX Fast File System
Log-Structured File System
Analysis and Evolution of Journaling File Systems

I. Background

i-node: structure for per-file metadata (unique per file)

- contains: ownership, permissions, timestamps, about 10 data-block pointers
- They form an array, indexed by "i-number". So each i-node has a unique i-number.
- Array is explicit for FFS, implicit for LFS (its i-node map is cache of i-nodes indexed by i-number)

indirect blocks:

- i-node only holds a small number of data block pointers
- for larger files, i-node points to an indirect block (holds 1024 entries for 4-byte entries in a 4K block), which in turn points to the data blocks.
- Can have multiple levels of indirect blocks for even larger files

II. A Fast File System for UNIX

Original UNIX file system was simple and elegant, but slow.

Could only achieve about 20 KB/sec/arm; ~2% of 1982 disk bandwidth

Problems:

- blocks too small
- consecutive blocks of files not close together (random placement for mature file system)
- i-nodes far from data (all i-nodes at the beginning of the disk, all data after that)
- i-nodes of directory not close together
- no read-ahead

Aspects of new file system:

- 4096 or 8192 byte block size (why not larger?)
- large blocks and small fragments
- disk divided into cylinder groups
- each contains superblock, i-nodes, bitmap of free blocks, usage summary info

FFS/LFS

- Note that i-nodes are now spread across the disk: keeps i-node near file, i-nodes of a directory together
- cylinder groups ~ 16 cylinders, or 7.5 MB
- cylinder headers spread around so not all on one platter

Two techniques for locality:

- don't let disk fill up in any one area
- paradox: to achieve locality, must spread unrelated things far apart
- note: new file system got 175KB/sec because free list contained sequential blocks (it did generate locality), but an old system has randomly ordered blocks and only got 30 KB/sec

Specific application of these techniques:

- goal: keep directory within a cylinder group, spread out different directories
- goal: allocate runs of blocks within a cylinder group, every once in a while switch to a new cylinder group (jump at 1MB).
- layout policy: global and local
- global policy allocates files & directories to cylinder groups. Picks "optimal" next block for block allocation.
- local allocation routines handle specific block requests. Select from a sequence of alternative if need to.

Results:

- 20-40% of disk bandwidth for large reads/writes.
- 10-20x original UNIX speeds.
- Size: 3800 lines of code vs. 2700 in old system.
- 10% of total disk space unusable (except at 50% perf. price)

Could have done more; later versions do.

Enhancements made to system interface: (really a second mini-paper)

- long file names (14 -> 255)
- advisory file locks (shared or exclusive); process id of holder stored with lock => can reclaim the lock if process is no longer around
- symbolic links (contrast to hard links)
- atomic rename capability (the only atomic read-modify-write operation, before this there was none)

FFS/LFS

- disk quotas
- Could probably have gotten copy-on-write to work to avoid copying data from user->kernel. (would need to copies only for parts that are not page aligned)
- Overallocation would save time; return unused allocation later. Advantages: 1) less overhead for allocation, 2) more likely to get sequential blocks

3 key features of paper:

- parameterize FS implementation for the hardware it's running on.
- measurement-driven design decisions
- locality "wins"

A major flaws:

- measurements derived from a single installation.
- ignored technology trends

A lesson for the future: don't ignore underlying hardware characteristics.

Contrasting research approaches: improve what you've got vs. design something new.

III. Log-Structured File System

Radically different file system design.

Technology motivations:

- CPUs outpacing disks: I/O becoming more-and-more of a bottleneck.
- Big memories: file caches work well, making most disk traffic writes.

Problems with current file systems:

- Lots of little writes.
- Synchronous: wait for disk in too many places. (This makes it hard to win much from RAIDs, too little concurrency.)
- 5 seeks to create a new file: (rough order) file i-node (create), file data, directory entry, file i-node (finalize), directory i-node (modification time).

Basic idea of LFS:

- Log all data and meta-data with efficient, large, sequential writes.
- Treat the log as the truth (but keep an index on its contents).
- Rely on a large memory to provide fast access through caching.

FFS/LFS

- Data layout on disk has “temporal locality” (good for writing), rather than “logical locality” (good for reading). Why is this a better? Because caching helps reads but not writes!

Two potential problems:

- Log retrieval on cache misses.
- Wrap-around: what happens when end of disk is reached?
  - No longer any big, empty runs available.
  - How to prevent fragmentation?

Log retrieval:

- Keep same basic file structure as UNIX (inode, indirect blocks, data).
- Retrieval is just a question of finding a file’s inode.
- UNIX inodes kept in one or a few big arrays, LFS inodes must float to avoid update-in-place.
- Solution: an *inode map* that tells where each inode is. (Also keeps other stuff: version number, last access time, free/allocated.)
- Inode map gets written to log like everything else.
- Map of inode map gets written in special checkpoint location on disk; used in crash recovery.

Disk wrap-around:

- Compact live information to open up large runs of free space. Problem: long-lived information gets copied over-and-over.
- Thread log through free spaces. Problem: disk will get fragmented, so that I/O becomes inefficient again.
- Solution: *segmented log*.
  - Divide disk into large, fixed-size segments.
  - Do compaction within a segment; thread between segments.
  - When writing, use only clean segments (i.e. no live data).
  - Occasionally *clean* segments: read in several, write out live data in compacted form, leaving some fragments free.
  - Try to collect long-lived information into segments that never need to be cleaned.
  - Note there is not free list or bit map (as in FFS), only a list of clean segments.

Which segments to clean?

- Keep estimate of free space in each segment to help find segments with lowest

utilization.

- Always start by looking for segment with utilization=0, since those are trivial to clean...
- If utilization of segments being cleaned is U:
  - write cost = (total bytes read & written)/(new data written) = 2/(1-U). (unless U is 0).
  - write cost increases as U increases: U = .9 => cost = 20!
  - need a cost of less than 4 to 10; => U of less than .75 to .45.

How to clean a segment?

- Segment summary block contains map of the segment. Must list every i-node and file block. For file blocks you need {i-number, block #}
- To clean an i-node: just check to see if it is the current version (from i-node map). If not, skip it; if so, write to head of log and update i-node map.
- To clean a file block, must figure out it if is still live. First check the UID, which only tells you if this file is current (UID only changes when is deleted or has length zero). Note that UID does not change every time the file is modified (since you would have to update the UIDs of all of its blocks). Next you have to walk through the i-node and any indirect blocks to get to the data block pointer for this block number. If it points to this block, then move the block to the head of the log.

Simulation of LFS cleaning:

- Initial model: uniform random distribution of references; greedy algorithm for segment-to-clean selection.
- Why does the simulation do better than the formula? Because of variance in segment utilizations.
- Added locality (i.e. 90% of references go to 10% of data) and things got worse!
- First solution: write out cleaned data ordered by age to obtain hot and cold segments.
  - What prog. language feature does this remind you of? Generational GC.
  - Only helped a little.
- Problem: even cold segments eventually have to reach the cleaning point, but they drift down slowly. tying up lots of free space. *Do you believe that's true?*
- Solution: it's worth paying more to clean cold segments because you get to keep the free space longer.
- Better way to think about this: don't clean segments that have a high d-free/dt (first derivative of utilization). If you ignore them, they clean themselves! LFS uses age as an approximation of d-free/dt, because the latter is hard to track directly.
- New selection function: MAX(T*(1-U)/(1+U)).
  - Resulted in the desired bi-modal utilization function.
  - LFS stays below write cost of 4 up to a disk utilization of 80%.

FFS/LFS

Checkpoints:

- Just an optimization to roll forward. Reduces recovery time.
- Checkpoint contains: pointers to i-node map and segment usage table, current segment, timestamp, checksum (?)
- Before writing a checkpoint make sure to flush i-node map and segment usage table.
- Uses “version vector” approach: write checkpoints to alternating locations with timestamps and checksums. On recovery, use the latest (valid) one.

Crash recovery:

- Unix must read entire disk to reconstruct meta data.
- LFS reads checkpoint and rolls forward through log from checkpoint state.
- Result: recovery time measured in seconds instead of minutes to hours.
- Directory operation log == log *intent* to acheive atomicity, then redo during recovery, (undo for new files with no data, since you can’t redo it)

Directory operation log:

- Example of “intent + action”: write the intent as a “directory operation log”, then write the actual operations (create, link, unlink, rename)
- This makes them atomic
- On recovery, if you see the operation log entry, then you can REDO the operation to complete it. (For new file create with no data, you UNDO it instead.)
- => “logical” REDO logging

An interesting point: LFS’ efficiency isn’t derived from knowing the details of disk geometry; implies it can survive changing disk technologies (such variable number of sectors/track) better.

Key features of paper:

- CPUs outpacing disk speeds; implies that I/O is becoming more-and-more of a bottleneck.
- Write FS information to a log and treat the log as the truth; rely on in-memory caching to obtain speed.
- Hard problem: finding/creating long runs of disk space to (sequentially) write log records to. Solution: clean live data from segments, picking segments to clean based on a cost/benefit function.

Some flaws:

- Assumes that files get written in their entirety; else would get intra-file fragmentation in LFS.
- If small files “get bigger” then how would LFS compare to UNIX?

FFS/LFS

A Lesson: Rethink your basic assumptions about what's primary and what's secondary in a design. In this case, they made the log become the truth instead of just a recovery aid.

IV. Analysis and Evolution of Journaling File Systems

Journaling file systems:

- Write-ahead logging: commit data by writing it to log, synchronously and sequentially
- Unlike LFS, then later moved data to its normal (FFS-like) location; this write is called *checkpointing* and like segment cleaning, it makes room in the (circular) journal
- Better for random writes, slightly worse for big sequential writes
- All reads go the the fixed location blocks, not the journal, which is only read for crash recovery and checkpointing
- Much better than FFS (fsck) for crash recovery (covered below) becuase it is much faster
- Ext3 filesystem is the main one in Linux; RieserFS was becoming popular

Three modes:

- **writeback** mode: journal only metadata, write back data and metadata independently
  metadata may thus have dangling references after a crash (if metadata written before the data with a crash in between)
- **ordered** mode: journal only metadata, but always write data blocks before their referring metadata is journaled. This mode generally makes the most sense and is used by Windows NTFS and IBM's JFS.
- **data journaling** mode: write both data and metadata to the journal
  Huge increase in journal traffic; plus have to write most blocks twice, once to the journal and once for checkpointing (why not all?)

Crash recovery:

- Load superblock to find the tail/head of the log
- Scan log to detect whole commited transactions (they have a commit record)
- Replay log entries to bring in-memory data structures up to date
  This is called "redo logging" and entries must be "idempotent"
- Playback is oldest to newest; tail of the log is the place where checkpointing stopped
- How to find the head of the log?

Some fine points:

- Can group transactions together: fewer syncs and fewer writes, since hot metadata may changes several times within one transaction
- Need to write a commit record, so that you can tell that all of the compound transaction made it to disk
- ext3 logs whole metadata blocks (physical logging); JFS and NTFS log logical records instead, which means less journal traffic
- head of line blocking: compound transactions can link together concurrent streams (e.g.

from different apps) and hinder asynchronous apps performance (Figure 6). This is like having no left turn lane and waiting on the car in front of you to turn left, when you just want to go straight.

- Distinguish between ordering of writes and durability/persistence: careful ordering means that after a crash the file system can be recovered to a consistent ***past*** state. But that state could be far in the past in the case of JFS. 30 seconds behind is more typical for ext3. If you really want something to be durable you must flush the log synchronously.

Semantic Block-level Analysis (SBA):

- Nice idea: interpose special disk driver between the file system and the real disk driver
- Pros: simple, captures ALL disk traffic, can use with a black-box filesystem (no source code needed and can even use via VMWare for another OS), can be more insightful than just a performance benchmark
- Cons: must have some understanding of the disk layout, which differs for each filesystem, requires a great deal of inference; really only useful for writes
- To use well, drive filesystem with smart applications that test certain features of the filesystem (to make the inference easier)

Semantic trace playback (STP):

- Uses two kinds of interpositionL 1) SBA driver that produces a trace, and 2) user-level library that fits between the app and the real filesystem
- User-level library traces dirty blocks and app calls to fsync
- Playback: given the two traces, STP generates a timed set of commands to the raw disk device. This sequence can be timed to understand performance implications.
- Claim: faster to modify the trace than to modify the filesystem and simpler and less error-prone than building a simulator
- Limited to simple FS changes
- Best example usage: showing that dynamically switching between ordered mode and data journaling mode actually gets the best overall performance. (Use data journaling for random writes.)

AutoRAID

Goals: automate the efficient replication of data in a RAID

- RAIDs are hard to setup and optimize
- Mix fast mirroring (2 copies) with slower, more space-efficient parity disks
- Automate the migration between these two levels

Levels of RAID (those in **bold** are actually used):

- RAID 0: striping with no parity (just bandwidth)
- **RAID 1: Mirroring** (simple, fast, but requires 2x storage)
  - Reads faster, writes slower (why?)
- RAID 2: bit interleaving with error-correcting codes (ECC)
- **Dedicated parity disk (RAID level 3), byte-level striping**
  - dedicated parity disk is a write bottleneck, since every write also writes parity
- RAID 4:dedicated parity disk, block-level striping
- **RAID 5: Rotating parity disk, block-level striping**
  - most popular; rotating disk spreads out parity load
- RAID 6: RAID 5 with two parity blocks (tolerates two failures)

RAID small-write problem:

- to overwrite part of a block required 2 reads and 2 writes!
- read data, read parity, write data, write parity

Each kind of replication has a narrow range of workloads for which it is best...

- Mistake ⇒ 1) poor performance, 2) changing layout is expensive and error prone
- Also difficult to add storage: new disk ⇒ change layout and rearrange data...

(another problem: spare disks are wasted)

Key idea: mirror active data (hot), RAID 5 for cold data

- Assumes only part of data in active use at one time
- Working set changes slowly (to allow migration)

Where to deploy:

- sys-admin: make a human move around the files.... BAD. painful and error prone
- File system: best choice, but hard to implement/deploy; can't work with existing systems

AutoRAID

- Smart array controller: (magic disk) block-level device interface. Easy to deploy because there is a well-defined abstraction; enables easy use of NVRAM (why?)

Features:

- Block Map: level of indirection so that blocks can be moved around among the disks
  - implies you only need one "zero block" (all zeroes), a variation of copy on write
  - in fact could generalize this to have one real block for each unique block
- Mirroring of active blocks
- RAID 5 for inactive blocks or large sequential writes (why?)
- Start out fully mirrored, then move to 10% mirrored as disks fill
- Promote/demote in 64K chunks (8-16 blocks)
- Hot swap disks, etc. (A hot swap is just a controlled failure.)
- Add storage easily (goes into the mirror pool)
  - useful to allow different size disks (why?)
- No need for an active hot spare (per se); just keep enough working space around
- Log-structured RAID 5 writes. (Why is this the right thing? Nice big streams, no need to read old parity for partial writes)

Sizes:

- PEX = 1MB
- PEG = set of PEXs
- segment = 128K
- relocation block = RB = 64K (size for holeplugging)

Issues:

- When to demote? When there is too much mirrored storage (>10%)
- Demotion leaves a hole (64KB). What happens to it? Moved to free list and reused
- Demoted RBs are written to the RAID5 log, one write for data, a second for parity
- Why log RAID5 better than update in place? Update of data requires reading all the old data to recalculate parity. Log ignores old data (which becomes garbage) and writes only new data/parity stripes.
- When to promote? When a RAID5 block is written... Just write it to mirrored and the old version becomes garbage.
- How big should an RB be? Bigger $\Rightarrow$ finer-grain migration, smaller $\Rightarrow$ less mapping information, bigger $\Rightarrow$ fewer seeks
- How do you find where an RB is? Convert addresses to (LUN, offset) and then lookup RB in a table from this pair. Map size = Number of RBs and must be proportional to size of total storage.
- Controller uses cache for reads
- Controller uses NVRAM for fast commit, then moves data to disks. What if NVRAM is full? Block until NVRAM blocks flushed to disk, then write to NVRAM.

AutoRAID

- Disks writes normally go to two disks (since newly written data is “hot”). Must wait for both to complete (why?). Does the host have to wait for both? No, just for NVRAM.
- What happens in the background? 1) compaction, 2) migration, 3) balancing.
- Compaction: clean the RAID5 and plug holes in the mirrored disks. Do mirrored disks get cleaned? Yes, when a PEG is needed for RAID5; i.e., pick a disks with lots of holes and move its used RBs to other disks. Resulting empty PEG is now usable by RAID5.
- What if there aren’t enough holes? Write the excess RBs to RAID5, then reclaim the PEG.
- Migration: which RBs to demote? Least-recently-**written (not LRU)**
- Balancing: make sure data evenly spread across the disks. (Most important when you add a new disk)

Bad cases? One is thrashing when the working set is bigger than the mirrored storage

Performance:

- consistently better than regular RAID, comparable to plain disks (but worse)
- They couldn’t get traditional RAIDs to work well...

Other things:

- “shortest seek” -- pick the disk (of 2) whose head is closet to the block
- When idle, plugs holes in RAID5 rather than append to log (easier because all RBs are the same size!) Why not all the time? Requires reading the rest of the stripe and recalculating parity
- Very important that the behavior is dynamic: makes it robust across workloads, across technology changes, and across the addition of new disks. Greatly simplifies management of the disk system

Key features of paper:

- RAIDs difficult to use well -- two levels and automatic data movement simplifies it
- Mix mirroring and RAID5 automatically
- Hide magic behind simple SCSI interface

Advanced Topics in Operating Systems, CS262a

Prof. Eric A. Brewer (with help from Rusty Sears)

Segment-oriented Recovery

ARIES works great but is 20+ years old and has some problems:

- viewed as very complex
- no available implementations with source code
- part of a monolithic DBMS: can use reuse transactions for other systems?
- LSN in the page breaks up large objects (Fig 1), prevents efficient I/O
- DBMS seems hard to scale for cloud computing (except by partitioning)

Original goals (Stasis):

- build an open-source transaction system with full ARIES-style steal/no-force transactions
- try to make the DBMS more "layered" in the OS style
- try to support a wider array of transactional systems: version control, file systems, bioinformatics or science databases, graph problems, search engines, persistent objects, ...
- competitive performance

These goals were mostly met, although the open-source version needs more users and we have only tried some of the unusual applications.

Original version was basically straight ARIES; development led to many insights and the new version based on segments.

Segment-oriented recovery (SOR)

A *segment* is just a range of bytes

- may span multiple pages
- may have many per page
- analogous to segments and pages in computer architecture (but arch uses segments for protection boundaries, we use them for recovery boundaries; in both, pages are fixed size and the unit of movement to disk)

Key idea: change two of the core ARIES design points:

- Recover segments not pages
- No LSNs on pages
  - ARIES: LSN per page enable *exactly once* redo semantics
  - SOR: estimate the LSN conservatively (older) => *at least once* semantics [but must now limit the actions we can take in redo]

The Big Four Positives of SOR:

1) Enable DMA and/or zero-copy I/O

- No LSNs per page mean that large objects are contiguous on disk
- No "segmentation and reassembly" to move objects to/from the disk
- No need to involve the CPU in object I/O (very expensive); use DMA instead

2) Fix persistent objects

- Problem: consider page with multiple objects, each of which has an in memory representation (e.g. a C++ or Java object)
  - Suppose we update object A and generate a log entry with LSN=87
  - Next we update object B (on the same page), generate its log entry with LSN=94, and write it back to the disk page, updating the page LSN
  - This pattern breaks recovery: the new page LSN (94) implies that the page reflects redo log entry 87, but it does not.
  - ARIES "solution": disk pages (in memory) must be updated on every log write; this is "write through caching" -- all updates are written through to the buffer manager page
- SOR solution:
  - there is no page LSN and thus no error
  - Buffer manager is written on cache eviction -- "write back caching". This implies much less copying/ marshalling for hot objects.
  - This is like "no force" between the app cache and the buffer manager (!)

3) Enable high/low priority transactions (log reordering on the fly)

- With pages, we assign the LSN atomically with the page write [draw fig 2]
  - not possible to reorder log entries at all
  - in theory, two independent transactions could have their log entries reordered based on priority or because they are going to different log disks (or log servers)
- SOR: do not need to know LSN at the time of the page update (just need to make sure it is assigned before you commit, so that it is ordered before later transactions; this is trivial to ensure -- just use normal locking and follow WAL protocol)
  - High priority updates: move log entry to the front of the log or to high priority "fast" log disk
  - Low priority updates go to end of log as usual

4) Decouple the components; enable cloud computing versions

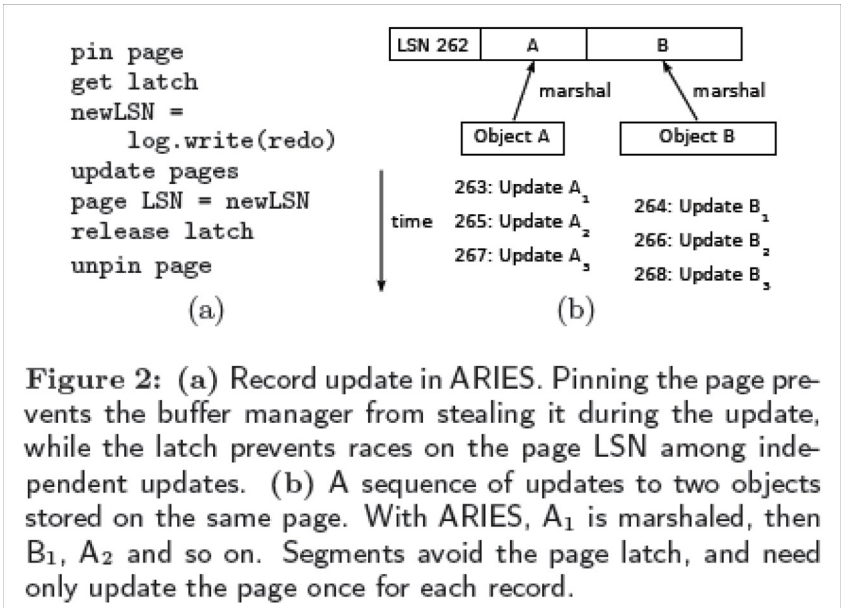

Figure 2: (a) Record update in ARIES. Pinning the page prevents the buffer manager from stealing it during the update, while the latch prevents races on the page LSN among independent updates. (b) A sequence of updates to two objects stored on the same page. With ARIES, A_1 is marshaled, then B_1, A_2 and so on. Segments avoid the page latch, and need only update the page once for each record.

- background: "pin" pages to keep them from being stolen (short term), "latch" pages to avoid race conditions within a page
- subtle problem with pages: must latch the page across the call to log manager (in order to get an LSN atomically)
- SOR has no page LSN and in fact no shared state at all for pages => no latch needed
- SOR decouple three different things:
  - App <-> Buffer manager: this is the write-back caching described above: only need to interact on eviction, not on each update
  - Buffer manager <-> log manager: no holding a latch across the log manager call; log manager call can now by asynchronous and batched together
  - Segments can be moved via zero-copy I/O directly, with no meta data (e.g. page LSN) and no CPU involvement. Simplifies archiving and reading large objects (e.g. photos).
- Hope: someone (ideally in CS262) will build a distributed transaction service using SOR
  - Apps, Buffer Manager, Log Manager, Stable storage could all be different clusters
  - Performance: (fig 11): 1-3 orders of magnitude difference for distributed transactions

Physiological Redo (review):

- Redos are applied exactly once (using the page LSN)
- Combination of physical and logical logging
  - physical: write pre- or post-images (or both)
  - logical: write the logical operation ("insert")
- Physiological:

- redos are physical
- normal undos are like redos and set a new LSN (does not revert to the old LSN -- wouldn't work given multiple objects per page!)
- to enable more concurrency, do not undo structural changes of a B-Tree (or other index); instead of a physical undo, issue a new logical undo that is the inverse operation. Enables concurrency because we can hold short locks for the structural change rather than long locks (until the end of the transaction)

o Slotted pages: add an array of offsets to each page (slots), then store records with a slot number and use the array to look up the current offset for that record. This allows changing the page layout without any log entries.

SOR Redo:

- o Redos may applied more than once; we go back farther in time than strictly necessary
- o Redos must be physical "blind writes" -- write content that do not depend on the previous contents
- o Undos can still be logical for concurrency
- o Slotted page layout changes require redo logging

Core SOR redo phase:

- o periodically write estimated LSNs to log (after you write back a page)
- o start from disk version of the segment (or from snapshot or whole segment write)
- o replay all redos since estimated LSN (worst case the beginning of the truncated log), even though some might have been applied already
- o for all bytes of the segment: either it was correct on disk and not changed or it was written during recovery in order by time (and thus correctly reflects the last log entry to touch that byte)

Hybrid Recovery: Can mix SOR and traditional ARIES

- o Some pages have LSNs, some don't
- o Can't easily look at a page and tell! (all the bytes are used for segments)
- o Log page type *changes* and zero out the page
  - recovery may actually corrupt a page temporarily until it gets the type correct, at which point it rolls forward correctly from the all zero page.
- o Example of when to use:
  - B-Trees: internal nodes on pages, leaves are segments
  - Tables of strings: short strings good for pages especially if they change size; long strings are good for segments due to contiguous layout

Extra stuff:

Why are there LSNs on pages?

- o so that we can atomically write the timestamp (LSN) with the page
- o problem: page writes aren't actually atomic anymore
- o solution: make them atomic so that ARIES still works

- write some bits into all sectors of a page (8K page = 16 512B sectors); compare those bits with a value store somewhere else. No match => recover the page (but may match and be wrong). Assumes sectors are written atomically, which is reasonable.
- write a checksum for each page (including the LSN) and check it when reading back the page

o Both solutions impact I/O performance some

o SOR approach:

- checksum each segment (or group of segments if they are small)
- on checksum failure, can replay last redo, which can fix a torn page (and then confirm using the checksum); if that doesn't work go back to old version and roll forward
- blind writes can fix corrupted pages since they do not depend on its contents

Advanced Topics in Operating Systems, CS262a

Prof. Eric A. Brewer (with help from Joe Hellerstein)

Lightweight Recoverable Virtual Memory

Thesis (in Related Work):

> *RVM ... poses and answers the question "What is the simplest realization of essential transactional properties for the average application?" By doing so, it makes transactions accessible to applications that have hitherto balked at the baggage that comes with sophisticated transactional facilities.*
>
> **Answer:** Library implementing No-Steal, No-Force virtual memory persistence, with manual copy-on-write and redo-only logs.

Goal: allow Unix applications to manipulate persistent data structures (such as the meta data for a file system) in a manner that has clear-cut failure semantics.

Existing solutions---such as Camelot---were too heavy-weight. Wanted a "lite" version of these facilities that didn't also provide (unneeded) support for distributed and nested transactions, shared logs, etc.

Solution: a library package that provides only recoverable virtual memory.

Lessons from Camelot:

- Overhead of multiple address spaces and constant IPC between them was significant.
- Heavyweight facilities impose additional onerous programming constraints.
- Size and complexity of Camelot and its dependence on special Mach features resulted in maintenance headaches and lack of portability. (The former of these two shouldn't be an issue in a ``production" system.)

Camelot had a yucky object and process model. Its componentization led to lots of IPC. It had poorly tuned log truncation. Was perhaps too much of an embrace of Mach.

- However, note that the golden age of CMU Systems learned a lot from the sharing of artifacts: Mach, AFS, Coda...
- A lot of positive spirit in this paper.

Architecture:

- Focus on metadata not data
- External data segments: think mmap of a file into memory
- Processes map regions of data segments into their address space. No aliasing allowed.
- Changes to RVM are made as part of transactions.
- set_range operation must be called before modifying part of a region. Allows a copy of the old values to be made so that they can be efficiently restored (in memory) after an abort.
- No-flush commits trade weaker permanence guarantees in exchange for better performance. Where might one use no-flush commits of transactions?
- No-restore == no explicit abort => no undo (ever) except for crash recovery
- in situ recovery (application code doesn’t worry about recovery!)

- To ensure that the persistent data structure remains consistent even when loss of multiple transactions ex-post-facto is acceptable. Example: file system.
- no isolation, no serialization, no handling of media recovery (but these can be added). Very systems view; not popular with DB folks...

These can be used to implement a funny kind of transaction checkpoint. Might want/need to flush the write-ahead log before the transaction is done, but be willing to accept ``checkpointing'' at any of the (internal) transaction points.

- flush: force log writes to disk.
  - p6: "Note that atomicity is guaranteed independent of permanence." This confuses DB folks. In this context, it means that all of the transactions are atomic (occur all or nothing), but the most recent ones might be forgotten in a crash. I.e. they change at the time of the crash from "all" to "nothing" even though the system told the caller that the transaction committed. Flush prevents this, which only happens with no-flush transactions.
- truncate: reflect log contents to external data segments and truncate the log.

Implementation:

- Log only contains new values because uncommitted changes never get written to external data segments. No undo/redo operations. Distinguishes between external data segment (master copy) and VM backing store (durable temp copy) => can STEAL by propagating to VM without need for UNDO (since you haven't written over the master copy yet)
- Crash recovery and log truncation: see paper.
- missing "set range" call is very bad -- nasty non-deterministic bugs that show up only during some crashes... Might use static analysis to verify set range usage...
- write-ahead logging: log intent then do idempotent updates to master copy (retry the update until it succeeds)

Optimizations:

- Intra-transaction: Coalescing set_range address ranges is important since programmers have to program defensively.
- Inter-transaction: Coalesce no-flush log records at (flush) commit time.

Performance:

- Beats Camelot across the board.
- Lack of integration with VM does not appear to be a significant problem as long as ration of Real/Physical memory doesn't grow too large.
- Log traffic optimizations provide significant (though not multiple factors) savings.

3 key features about the paper:

- Goal: a facility that allows programs to manipulate persistent data structures in a manner that has clear-cut failure semantics.
- Original experience with a heavyweight, fully general transaction support facility led to a project to build a lightweight facility that provides only recoverable virtual memory (since that is all that was needed for the above-mentioned goal).
- Lightweight facility provided the desired functionality in about 10K lines of code, with

significantly better performance and portability.

A flaw: The paper describes how a general facility can be reduced and simplified in order to support a narrower applications domain.

Although it argues that the more general functionality could be regained by building additional layers on top of the reduced facility, this hasn't been demonstrated.

Also allows for "persistent errors" -- errors in a set range region aren't fixable by reboot... they last until fixed by hand.

A lesson: When building a general OS facility, pick one (or a very few) thing(s) and do it well rather than providing a general facility that offers many things done poorly.

Background on distributed transactions:

Two models for committing a transaction:

- one-phase: used by servers that maintain only volatile state. Servers only send an end request to such servers (i.e. they don't participate in the voting phase of commit).
- two-phase: used by servers that maintain recoverable state. Servers send both vote and end requests to such servers.

4 different kinds of votes may be returned:

- Vote-abort
- Vote-commit-read-only: participant has not modified recoverable data and drops out of phase two of commit protocol.
- Vote-commit-volatile: participant has not modified recoverable data, but wants to know outcome of the transaction.
- Vote-commit-recoverable: participant has modified recoverable data

Concurrency Control

CS262, Fall 2011.

Joe Hellerstein

I. Concurrency & controlling order.

- Order in a traditional programming abstraction.
- Why is concurrency hard?
- Why does order matter? What orders matter? These are deep questions.
- Step back: ignore shared state, side-channels, etc. Assume no sharing, explicit communication,

II. Inter-Agent Communication: Rendezvous/Join

The Ancient Communication Scenario: 1 sender, 1 receiver.

1. **SpatioTemporal Rendezvous**: Timed Smoke Signals on 2 mountaintops. Agent 1 has instructions to generate a puff at certain time (and place). Agent 2 has instructions to watch at same time (and place).
2. **Receiver Persist**: Smoke Signal and Watchtower. Agent 2 *waits* in watchtower for smoke-signal in the agreed-upon place. Agent 1 puffs at will. (What if too early?)
3. **Sender Persist**: WatchFires. Agent 1 can light a watchfire. Agent 2 checks for the watchfire when convenient. (What if too early?)
4. **Both Persist**: Watchfire and Watchtower. Both persist.

Upshot: (1) one-sided persistence allows asynchrony on the other side. (2) BUT persistent party must span the time of the transient party. (3) Without any temporal coordination, persistence of both sides is the way to guarantee rendezvous: second arrival necessarily overlaps first.

This is *very* much like a choice of join algorithm in a database engine!

Next Question: Multiple communications. Can we guarantee order? With temporal rendezvous, the agents coordinate ("share a clock"). With receiver persist, receiver can observe/maintain sender order. Sender persist much more common, but doesn't (organically) preserve order!

Beware: In most computing settings, this reasoning is flipped. Storage is a given, and ends up being an *implicit* comm channel. Must control operations on it across agents.

III. On State and Persistence

What is state?

- all values of memory, registers, stack, PC, etc.

What is persistent state?

- persistent state: state that is communicated across *time*. E.g. across PC settings. Write is a Send into the future, Read is Receive.
- how does DRAM persist state? sender-persist! And recall: Sender-Persist doesn't guarantee order!!
- (aside: how does OS do disk persistence?)

IV. How to control communication?

- Prevention: Change the space or the time of rendezvous!
    - writer uses private space (shadow copies, copy-on-write, multiversion, functional

programming, etc.)
    - and/or a *protocol* arranges appropriate "time" for rendezvous (e.g. agree on timing scheme via locks, or on a publication location scheme).
        - these protocols typically rely recursively on controlled communication! (e.g. callback or continuation on lock release, e.g. publication of a fwding pointer in a well-known location followed by polling, etc.)
- Detection: Identify an undesirable communication (one that violates ordering constraints), and somehow *undo* an agent.
    - Ensure that all communication it performed was invisible, or recursively undo.

V. Acceptable orders of communication

1. What are the basic operations on state? Communication — i.e. data dependencies.
2. What orders matter for these operations? It depends!
3. A client's view of the acceptable orderings: serial schedules, serializability.
4. Conflicts. Assume two clients, one server. What interleavings do not preserve client views? R-W and W-W conflicts. *Conflict Serializability is a conservative test for Serializability.*

VI. Locking as the "antijoin antechamber"

- Idea: Since communication is rendezvous, let's prevent inappropriate rendezvous — no conflicts! Result: equiv. to a serial schedule based on time-of-commit. (Dynamic!)
- consider a rendezvous in space: e.g. two spies passing a message in a locked room.
- before you can "enter" the rendezvous point:
    - check that no conflicting action is currently granted access to the rendezvous point (not-in = "antijoin").
        - if yes conflict with the granted actions (join), wait in the antechamber
        - if no conflict, mark yourself (persistently until EOT) as having been granted access. Enter the rendezvous point to do your business
    - when granted transactions depart, choose a mutually compatible group of transactions to let into the granted group.
- some games we can play with space
    - could "move" the rendezvous point in space (locking proxy? lock caching?)
    - could further delay rendezvous in time based on other considerations
- You should know the multi-granularity locks from Gray's paper! (Took the MySQL guys years to learn this.)

VII. Timestamp ordering (T/O)

Streaming symmetric join of reads and writes, outputting data and aborts.

T/O. Predeclare the schedule via timestamp (wall-clock? sequence? out-of-sequence?) at transaction birth. No antijoin = no waiting! But restart when reordering detected…

- Keep track of r-ts and w-ts for each object: *r-ts: max counter rather than persistence.*
- reject (abort) stale reads.
- reject (abort) delayed writes to objects that have later r-ts (write was already missed). Can allow (ignore) delayed writes to objects that have a later w-ts tho. (Thomas Write Rule.)

Multiversion T/O: Even fewer restart scenarios

- Keep sets of r-ts, and . This is kind of like symmetric persistence.
- Reads never rejected! (more liberal than plain T/O)
- Write rule on x: interval(W(x)) = [ts(W), mw-ts] where mw-ts is the next write after W(x) — i.e. Min_x w-ts(x) > ts(W). If any R-ts(x) exists in that interval, must reject write. I.e. that read shoulda read this write. Note more liberal than plain T/O since subsequent writes may "mask" this one for even later reads.
- GC of persistent state: anything older than Min TS of live transactions.

Note: no waiting (blocking, antijoin) in Multiversion T/O.

VIII. Optimistic concurrency: antijoin with history

Schedule predeclared by acquiring timestamp before validation.

- go through **basic OCC**
- idea: persist read history, copy on write, and try to antijoin over time window on commit to ensure "conflicts in a particular order"
- OCC antijoin predicates are complicated — transaction numbers, timestamps for read/write phases, read/write sets:
    - Valdating Tj. Suppose TN(Ti) < TN(Tj). Serializable if *for all uncommitted Ti* one of the following holds ("for all" is like anti join — i.e. notin(set-of-uncommited-Ti, NONE of the following hold))
    - Ti completes writes before Tj starts reads (prevents rw and ww conflicts out of order).
    - WS(Ti) \cap RS(Tj) = empty, Ti completes writes before Tj starts writes (no rw conflicts in order, prevent ww conflicts out of order)
    - WS(Ti) \cap RS(Tj) = empty, WS(Ti) \cap WS(Tj) = empty, Ti finishes read before Tj starts read (no ww conflicts, prevent backward rw conflicts).
    - GC?

Note: the antijoin in OCC is over all uncommitted transactions. That means that during validation, the set of uncommitted transactions must not change. I.e. only one transactions can be validating at a time. I.e. implicitly there's an X lock on the "system validating" resource.

IX. ACID

Not an axiom system. Just a mnemonic. Don't over-interpret. Remember serializability, and guarantee commit/abort.

- Atomicity: visibility of all effects at one time
- Consistency: based on state constraints
- Isolation: application writer need not reason about concurrency
- Durability: commit is a contract

X. What is all this discussion about NoSQL and Loose Consistency?

Move some coordination out of read/write storage, and into application logic.

- *ACID*: build application logic on a foundation of theory+system for controlling order.
- *Loose Consistency*: build application logic with controlled order over a loose foundation. I.e. Abandon Isolation, require programmer to worry about concurrency issues that they might care about.
- Missing from loose consistency: a foundation of theory+SW to ensure app logic meets desired constraints.
- What about Gray's transactional "degrees of consistency"?
    - A Dirty-Data Description of Degrees of Consistency Transaction T sees degree X consistency if…
        - Degree 0: T does not overwrite dirty data of other transactions
        - Degree 1: T sees degree 0 consistency, and T does not commit any writes before EOT
        - Degree 2: T sees degree 1 consistency, and T does not read dirty data of other transactions
        - Degree 3: T sees degree 2 consistency, and other transactions do not dirty any data read by T before T completes.
    - Note there were long-unresolved problems with Gray's definitions. See the Adya paper in the reading list.

    - **NOTE**: if everybody is at least degree 1, than different transactions can CHOOSE what degree they wish to "see" without worry. I.e. true isolation is an option for all.

- Eventual Consistency?
    - Typically in regard to R/W reasoning on replicated state. A good reference is **Terry, et al. PDIS 1994**
    - At higher levels of abstraction than R/W, what orders matter?

XI. What orders really matter? CALM Theorem.

CALM Theorem: Consistency and Logical Monotonicity. No coordination (i.e. order control) needed for Monotonic code. Non-monotonic code needs to be protected by coordination.

CS262A: Advanced Topics in Computer Systems
Joe Hellerstein, Spring 2008
UC Berkeley

Concurrency Control: Locking, Optimistic, Degrees of Consistency

Transaction Refresher

Statement of problem:

- **Database:** a fixed set of named resources (e.g. tuples, pages, files, whatever)
- **Consistency Constraints:** must be true for DB to be considered "consistent". Examples:
  - sum(account balances) = sum(assets)
  - P is index pages for R
  - ACCT-BAL > 0
  - each employee has a valid department
- **Transaction:** a sequence of actions bracketed by begin and end statements. Each transaction ("xact") is assumed to maintain consistency (this can be guaranteed by the system).
- **Goal:** Concurrent execution of transactions, with high throughput/utilization, low response time, and fairness.

A transaction schedule:

T1	T2
read(A)	
A = A - 50	
write(A)	
	read(A)
	temp = A*0.1
	A = A - temp
	write(A)
read(B)	
B = B + 50	
write(B)	
	read(B)
	B = B + temp
	write(B)

The system "understands" only reads and writes; cannot assume any semantics of other operations.
Arbitrary interleaving can lead to:
- temporary inconsistencies (ok, unavoidable)
- "permanent" inconsistencies, that is, inconsistencies that remain after transactions have completed.

Some definitions:
- Schedule: A "history" or "audit trail" of all committed actions in the system, the xacts that performed them, and the objects they affected.
- Serial Schedule: A schedule is serial if all the actions of each single xact appear together.
- Equivalence of schedules: Two schedules S1, S2 are considered equivalent if:
    1. The set of transactions that participate in S1 and S2 are the same.
    2. For each data item Q in S1, if transaction Ti executes read(Q) and the value of Q read by Ti was written by Tj, then the same will hold in S2. [reads are all the same]
    3. For each data item Q in S1, if transaction Ti executes the last write(Q) instruction, then the same holds in S2. [the same writers "win"]
- Serializability: A schedule S is serializable if there exists a serial schedule S' such that S and S' are equivalent.

One way to think about concurrency control – in terms of dependencies:
1. T1 reads N ... T2 writes N: a RW dependency
2. T1 writes N ... T2 reads N: a WR dependency
3. T1 writes N ... T2 writes N: a WW dependency

Can construct a "serialization graph" for a schedule S (SG(S)):
- nodes are transactions T1, ..., Tn
- Edges: Ti -> Tj if there is a RW, WR, or WW dependency from Ti to Tj

Theorem: A schedule S is serializable iff SG(S) is acyclic.

Locking

A technique to ensure serializability, but hopefully preserve high concurrency as well. The winner in industry.

Basics:
- A "lock manager" records what entities are locked, by whom, and in what "mode". Also maintains wait queues.
- A well-formed transaction locks entities before using them, and unlocks them some time later.

Multiple lock modes: Some data items can be shared, so not all locks need to be exclusive.
Lock compatibility table 1:
Assume two lock modes: shared (S) and exclusive (X) locks.

	S	**X**
S	T	F
X	F	F

If you request a lock in a mode incompatible with an existing lock, you must wait.
Two-Phase Locking (2PL):
- Growing Phase: A transaction may obtain locks but not release any lock.
- Shrinking Phase: A transaction may release locks, but not obtain any new lock. (in fact, locks are usually all released at once to avoid "cascading aborts".)

Theorem: If all xacts are well-formed and follow 2PL, then any resulting schedule is serializable
(note: this is if, not if and only if!)

Some implementation background
- maintain a lock table as hashed main-mem structure
- lookup by lock name
- lookup by transaction id
- lock/unlock must be atomic operations (protected by critical section)
- typically costs several hundred instructions to lock/unlock an item
- suppose T1 has an S lock on P, T2 is waiting to get X lock on P, and now T3 wants S lock on P. Do we grant T3 an S lock?

Well, starvation, unfair, etc. Could do a little of that, but not much. So...
- Manage FCFS queue for each locked object with outstanding requests
- all xacts that are adjacent and compatible are a compatible group
- The front group is the granted group
- group mode is most restrictive mode amongst group members
- Conversions: often want to convert (e.g. S to X for "test and modify" actions). Should conversions go to back of queue?
- No! Instant deadlock (more notes on deadlock later). So put conversions right after granted group.

Granularity of Locks and Degrees of Consistency

Granularity part is easy. But some people still don't know about it (e.g. MySQL).
Be sure to understand IS, IX, SIX locks. Why do SIX locks come up?

First, a definition: A write is committed when transaction if finished; otherwise, the write is dirty.

A Locking-Based Description of Degrees of Consistency:
This is not actually a description of the degrees, but rather of how to achieve them via locking. But it's better defined.
- Degree 0: set short write locks on updated items ("short" = length of action)
- Degree 1: set long write locks on updated items ("long" = EOT)
- Degree 2: set long write locks on updated items, and short read locks on items read
- Degree 3: set long write and read locks

A Dirty-Data Description of Degrees of Consistency
Transaction T sees degree X consistency if...
- Degree 0: T does not overwrite dirty data of other transactions
- Degree 1:
  - T sees degree 0 consistency, and
  - T does not commit any writes before EOT
- Degree 2:
  - T sees degree 1 consistency, and
  - T does not read dirty data of other transactions
- Degree 3:
  - T sees degree 2 consistency, and
  - Other transactions do not dirty any data read by T before T completes.

Examples of Inconsistencies prevented by Various Degrees
Garbage reads:

> T1: write(X); T2: write(X)
> Who knows what value X will end up being?
> *Solution:* set short write locks (degree 0)

Lost Updates:

T1: write(X)
T2: write(X)
T1: abort (physical UNDO restores X to pre-T1 value)
At this point, the update to T2 is lost
Solution: set long write locks (degree 1)

Dirty Reads:

T1: write(X)
T2: read(X)
T1: abort

Now T2's read is bogus.
Solution: set long X locks and short S locks (degree 2)
Many systems do long-running queries at degree 2.

Unrepeatable reads:

T1: read(X)
T2: write(X)
T2: end transaction
T1: read(X)

Now T2 has read two different values for X.
Solution: long read locks (degree 3)

Phantoms:

T1: read range [x - y]
T2: insert z, $x < z < y$
T2: end transaction
T1: read range [x - y]

Z is a "phantom" data item (eek!)
Solution: ??

NOTE: if everybody is at least degree 1, than different transactions can CHOOSE what degree they wish to "see" without worry. I.e. can have a mixture of levels of consistency.

Oracle's Snapshot Isolation
A.K.A. "SERIALIZABLE" in Oracle. Orwellian!

Idea: Give each transaction a timestamp, and a "snapshot" of the DBMS at transaction begin. Then install their writes at commit time. Read-only transactions never block or get rolled back!

Caveat: to avoid Lost Updates, ensure that "older" transactions don't overwrite newer, committed transactions.

Technique: "archive on write". I.e. move old versions of tuples out of the way, but don't throw them out.

- On write, if there's space on the page, they can move to a different "slot". Else move to a "rollback segment"
- Readers see the appropriate versions (snapshot plus their own updates). Non-trivial: needs to work with indexes as well as filescans.
- Writes become visible at commit time.
- "First committer wins": At commit time, if T1 and T2 have a WW conflict, and T1 commits, then T2 is aborted. Oracle enforces this by setting locks. Pros and Cons??

A snapshot isolation schedule:

```
T1: R(A0), R(B0),                          W(A1), C
T2:                R(A0), R(B0), W(B2), C
```

Now, B2 = f(A0, B0); A2 = g(A0, B0). Is this schedule serializable? Example:

- A is checking, B is savings. Constraint: Sum of balances > $0.
- Initially, A=70, B = 80.
- T1 debits $100 from checking.
- T2 debits $100 from savings.

"Write Skew"! There are subtler problems as well, see O'Neil paper.
Still, despite IBM complaining that these anomalies mean the technique is "broken", Snapshot Isolation is popular and works "most of the time" (and certainly on leading benchmarks.)

Question: How do you extend Snapshot Isolation to give true serializable schedules? Cost?

Optimistic Concurrency Control

Attractive, simple idea: optimize case where conflict is rare.
Basic idea: all transactions consist of three phases:

1. Read. Here, all writes are to private storage (shadow copies).
2. Validation. Make sure no conflicts have occurred.
3. Write. If Validation was successful, make writes public. (If not, abort!)

When might this make sense? Three examples:

1. All transactions are readers.
2. Lots of transactions, each accessing/modifying only a small amount of data, large total amount of data.
3. Fraction of transaction execution in which conflicts "really take place" is small compared to total pathlength.

The Validation Phase

- Goal: to guarantee that only serializable schedules result.
- Technique: actually find an equivalent serializable schedule. That is,
  1. Assign each transaction a TN during execution.
  2. Ensure that if you run transactions in order induced by "<" on TNs, you get an equivalent serial schedule.
  3.

Suppose TN(Ti) < TN(Tj). Then if one of the following three conditions holds, it's serializable:

1. Ti completes its write phase before Tj starts its read phase.
2. WS(Ti) ∩ RS(Tj) = ∅ and Ti completes its write phase before Tj starts its write phase.
3. WS(Ti) ∩ RS(Tj) = ∅ and WS(Ti) ∩ WS(Tj) = ∅ and Ti completes its read phase before Tj completes its read phase.

Is this correct? Each condition guarantees that the three possible classes of conflicts (W-R, R-W, W-W) go "one way" only: higher transaction id (j) depends on lower (i), but not vice versa. (When we speak of conflicts below, they are implicitly ordered i then j.)

1. For condition 1 this is obvious (true serial execution!)
2. Interleave Wi/Rj, but ensure no WR conflict. RW and forward WW allowed.
3. interleave Wi/Rj or Wi/Wj. I.e. forbid only Ri/Wj interleaving, AND ensure all conflicts are RW

Assigning TN's: at beginning of transactions is not optimistic; do it at end of read phase. Note: this satisfies second half of condition (3).

Note: a transaction T with a very long read phase must check write sets of all transactions begun and finished while T was active. This could require unbounded buffer space.
Solution: bound buffer space, toss out when full, abort transactions that could be affected.

- Gives rise to starvation. Solve by having starving transaction write-lock the whole DB!

Serial Validation
Only checks properties (1) and (2), since writes are not going to be interleaved.
Simple technique: make a critical section around <get xactno; validate (1) or (2) for everybody from your start to finish; write>. Not great if:

- write takes a long time
- SMP – might want to validate 2 things at once if there's not enough reading to do

Improvement to speed up validation:

```
repeat as often as you want {
      get current xactno.
      Check if you're valid with everything up to that xactno.
}
<get xactno; validate with new xacts; write>.
```

Note: read-only xacts don't need to get xactnos! Just need to validate up to highest xactno at end of read phase (without critical section!)

Parallel Validation
Want to allow interleaved writes.
Need to be able to check condition (3).

- Save active xacts (those which have finished reading but not writing).
- Active xacts can't intersect your read or write set.
- Validation:
  <get xactno; copy active; add yourself to active>
  check (1) or (2) against everything from start to finish;
  check (3) against all xacts in active copy
  If all's clear, go ahead and write.
  <bump xact counter, remove yourself from active>.

Small critical section.
Problems:

- a member of active that causes you to abort may have aborted
  - can add even more bookkeeping to handle this
  - can make active short with improvement analogous to that of serial validation

Does this make any sense?
Ask yourself:

- what's the running state of the approach, what bounds that state?
- what happens on conflict and how does that affect performance?

Advanced Topics in Computer Systems, CS262A

Prof. Eric Brewer

Experience With Processes and Monitors in Mesa

I. Experience With Processes and Monitors in Mesa

Focus of this paper: light-weight processes (threads in today's terminology) and how they synchronize with each other.

History:

- 2nd system; followed the Alto.
- planned to build a large system using many programmers. (Some thoughts about commercializing.)
- advent of things like server machines and networking introduced applications that are heavy users of concurrency.

Chose to build a single address space system:

- single user system, so protection not an issue. (Safety was to come from the language.)
- wanted global resource sharing.

Large system, many programmers, many applications:

- Module-based programming with information hiding.

Since they were starting "from scratch", they could integrate the hardware, the runtime software, and the language with each other.

Programming model for inter-process communication: shared memory (monitors) vs. message passing.

- Needham & Lauer claimed the two models are duals of each other.
- Chose shared memory model because they thought they could fit it into Mesa as a language construct more naturally.

How to synchronize processes?

- Non-preemptive scheduler: tends to yield very delicate systems. Why?
  - Have to know whether or not a yield might be called for *every* procedure you call. Violates information hiding.
  - Prohibits multiprocessor systems.
  - Need a separate preemptive mechanism for I/O anyway.
  - Can't do multiprogramming across page faults.
- Simple locking (e.g. semaphores): too little structuring discipline, e.g. no guarantee that locks will be released on every code path; wanted something that could be integrated into a Mesa language construct.

Lecture 19

- Chose preemptive scheduling of light-weight processes and monitors.

Light-weight processes:

- easy forking and synchronization
- shared address space
- fast performance for creation, switching, and synchronization; low storage overhead.

Monitors:

- monitor lock (for synchronization)
  - tied to module structure of the language: makes it clear what's being monitored.
  - language automatically acquires and releases the lock.
- tied to a particular ***invariant***, which helps users think about the program
- condition variable (for scheduling)
- Dangling references similar to those of pointers. There are also language-based solutions that would prohibit these kinds of errors, such as do-across, which is just a parallel control structure. It eliminates dangling processes because the syntax defines the point of the fork and the join.
- Monitors (and Mesa in particular) led to several aspects of Java. Java's synchronized

Monitors	Java Synchronized Objects
external	public
internal	private synchronized
entry	public synchronized

objects are the object-oriented programming version on monitors, and they are a better solution than monitored records. (Each instance has its own lock rather than just each element of an array.)

Changes made to design and implementation issues encountered:

- 3 types of procedures in a monitor module:
  - entry (acquires and releases lock).
  - internal (no locking done): can't be called from outside the module.
  - external (no locking done): externally callable. Why is this useful?

- allows grouping of related things into a module.
- allows doing some of the work outside the monitor lock.
- allows controlled release and reacquisition of monitor lock.

- Notify semantics:
  - Cede lock to waking process: too many context switches. Why would this approach be desirable? (Waiting process knows the condition it was waiting on is guaranteed to hold.)

- Notifier keeps lock, waking process gets put a front of monitor queue. Doesn't work in the presence of priorities.
- Notifier keeps lock, wakes process with no guarantees => waking process must recheck its condition.

What other kinds of notification does this approach enable?

Timeouts, broadcasts, aborts.

- Abort: a nice request to abort -- allows the target process to reach a wait or monitor exit, and then it voluntarily aborts. No need to re-establish the invariant -- as compared to just killing the process outright!
- Deadlocks: Wait only releases the lock of the current monitor, not any nested calling monitors. This is a general problem with modular systems and synchronization: synchronization requires *global* knowledge about locks, which violates the information hiding paradigm of modular programming. Why is monitor deadlock less onerous than the yield problem for non-preemptive schedulers?
  - Want to generally insert as many yields as possible to provide increased concurrency; only use locks when you want to synchronize.
  - Yield bugs are difficult to find (symptoms may appear far after the bogus yield)
- Basic deadlock rule: no recursion, direct or mutual
- Lock granularity: introduced monitored records so that the same monitor code could handle multiple instances of something in parallel.
- Interrupts: interrupt handler can't block waiting to acquire a monitor lock.
  - Introduced *naked* notifies: notifies done without holding the monitor lock.
  - Had to worry about a timing race: the notify could occur between a monitor's condition check and its call on Wait. This a "time of check to time of use" (toctou, pronounced "tock-too") bug -- the condition of the test no longer applies at the time of use. In particular, the sequence: 1) check for waiters == false, 2) naked notify, 3) go to sleep is a toctou bug. Added a *wakeup-waiting* flag to condition variables; naked notify sets the wakeup waiting flag if the target is awake, and before that process goes to sleep it checks this bit, and if set, resets it, and stays awake to check for notifies again.
  - What happens in general with a message handlers that needs to acquire a lock? (use the interrupt to queue the task, and then acquire locks in a process context instead)
- Priority Inversion
  - high-priority processes may block on lower-priority processes
  - a solution: temporarily increase the priority of the holder of the monitor to that of the highest priority blocked process (somewhat tricky -- what happens when that high-priority process finishes with the monitor? You have to know the priority of the next highest ⇒ keep them sorted or scan the list on exit)
  - The Mars rover stalled due to this kind of bug and had to be debugged and fixed from earth!
- Exceptions: must restore monitor invariant as you unwind the stack. What does Java do? (you must use a sequence of try-finally blocks)
  - The idea that you can just kill a process and release the locks is naive -- each lock protects some invariant that really needs to be restored before you can release the lock.
  - Entry procedures that have an exception, but no exception handler DO NOT release the monitor lock. This ensures deadlock and a trip into the debugger, but at least it maintains the invariant.

Lecture 19

Hints vs. Guarantees:

- Notify is only a hint.
- ⇒ don't have to wake up the right process, don't have to change the notifier if we slightly change the wait condition (the two are decoupled).
- ⇒ easier to implement, because it's always OK to wake up too many processes. If we get lost, we could even wake up everybody (broadcast)
- Enables timeouts and aborts
- General Principle: use hints for performance that have little or better yet no effect on the correctness. Inktomi uses hints for fault tolerance: if the hint is wrong, things we'll timeout and we'll use a backup strategy ⇒ performance hit for incorrect hint, but no errors.

Performance:

- Context switch is very fast: 2 procedure calls.
  - Ended up not mattering much for performance:

  - ran only on uniprocessor systems.

  - concurrency mostly used for clean structuring purposes.
- Procedure calls are slow: 30 instrs (RISC proc. calls are 10x faster). Due to heap-allocated procedure frames. Why did they do this?
  - Didn't want to worry about colliding process stacks.
  - Mental model was "any procedure call might be a fork": xfer was basic control transfer primitive.
- Process creation: ~ 1100 instrs.
  - Good enough most of the time.
  - Fast-fork package implemented later that keeps around a pool or "available" processes.

3 key features about the paper:

- Describes the experiences designers had with designing, building and using a large system that aggressively relies on light-weight processes and monitor facilities for all its software concurrency needs.
- Describes various subtle issues of implementing a threads-with-monitors design in real life for a large system.
- Discusses the performance and overheads of various primitives and three representative applications, but doesn't give a big picture of how important various things turned out to be.

Some flaws:

- Gloss over how hard it is to program with locks and exceptions sometimes. (Not clear if there are better ways).
- Performance discussion doesn't give the big picture.

Lecture 19

A lesson: The light-weight threads-with-monitors programming paradigm can be used to successfully build large systems, but there are subtle points that have to be correct in the design and implementation in order to do so.

Advanced Topics in Computer Systems, CS262A
Prof. Eric Brewer

SEDA
Capriccio

I. Background

Threads:

- The standard model (similar to Mesa)
- Concurrent threads with locks/mutex/etc. for synchronization
- Blocking calls
- May hold locks for long time
- Problems with more than 100 or so threads due to OS overhead (why?)
  - see SEDA graph of throughput vs. number of threads
- Strong support from OS, libraries, debuggers, ...

Events:

- Lots of small handlers that run to completion
- Basic model is event arrives and runs a handler
  - state is global or part of handler (not much in between)
- An event loops runs at the core waiting for arrivals, then calls handler
- No context switch, just procedure call
- Threads exist, but just run event loop -> handler -> event loop
  - stack trace is not useful for debugging!
  - typically one thread per CPU (any more doesn't add anything since threads don't block)
  - Sometimes have extra threads for things that may block; e.g. OSs that only support synchronous disk reads
- Natural fit with finite-state machines (FSMs)
  - arrows are handlers that change states
  - blocking calls are split into two states (before and after the call)
- Allows very high concurrency
  - multiplex 10,000 FSMs over a small number of threads

II. Cooperative Task Scheduling

Tasks

Week 4

- **preemptive**: tasks may be interrupted at any time
  - must use locks/mutex to get atomicity
  - may get pre-empted while holding a lock -- others must wait until you are rescheduled
  - might want to differentiate short and long atomic sections (short should finish up work)
- **serial**: tasks run to completion
  - basic event handlers, which are atomic
  - not allowed to block
  - what if they run too long? (not much to do about that, could kill them; implies might be better for friendly systems)
  - hard to support multiprocessors
- **cooperative**: tasks are not pre-empted, but do yield the processor
  - can use stacks and make calls, but still interleaved
  - yield points are not atomic: limits what you can do in an atomic section
  - better with compiler help: is a call a yield point or not?
  - hard to support multiprocessors
- Note: pre-emption is OK if it can't affect the current atomic section. Easy way to achieve this is data partitioning! Only threads that access the shared state are a problem!
  - Can pre-empt for system routines
  - Can pre-empt to switch to a different process (with its own set of threads), but assumes processes don't share state

Split-phase actions: how do you implement a split-phase action?

- Threads: not too bad -- just block until the action completes (synchronous)
  - Assumes other threads run in the meantime
  - Ties up considerable memory (full stack)
  - Easy memory management: stack allocation/deallocation matches natural lifetime
- Events: hard
  - Must store live state in a continuation (on the heap usually). Handler lifetime is too short, so need to explicitly allocate and deallocate later
  - Scoping is bad too: need a multi-handler scope, which usually implies global scope
  - Rips the function into two functions: before and after
  - Debugging is hard
  - Evolution is hard: adding a yielding call implies more ripping to do; converting a non-yielding call into a yielding call is worse -- every call site needs to be ripped and those sites may become yielding which cascades the problem

Atomic split-phase actions (really hard):

Week 4

- Threads -- pessimistic: acquire lock and then block
- Threads -- optimistic: read state, block, try write, retry if fail (and re-block!)
- Events -- pessimistic: acquire lock, store state in continuation ; later reply completes and releases lock. (seems hard to debug, what if event never comes? or comes more than once?)
- Events -- optimistic: read state, store in continuation ; apply write, retry if fail
- Basic problem: exclusive access can last a long time -- hard to make progress
- General question: when can we move the lock to one side or the other?
- One strategy:
  - structure as a sequence of actions that may or may not block (like cache reads)
  - acquire lock, walk through sequence, if block then release and start over
  - if get all the way through then action was short (and atomic)
  - This seems hard to automate! Compiler would need to know that some actions mostly won't block or won't block the second time... and then also know that something can be retried without multiple side effects...

Main conclusion (for me): compilers are the key to concurrency in the future

III. SEDA

Problem to solve:

- 1) need a better way to achieve concurrency than just threads
- 2) need to provide graceful degradation
- 3) need to enable feedback loops that adapt to changing conditions
- Internet makes the concurrency higher, requires high availability and ensures that load will exceed the target range.

Graceful degradation:

- want throughput to increase linearly and then stay flat as you exceed capacity
- want response time to be low until saturation and then linearly increase
- want fairness in the presence of overload
- almost no systems provide this
- key is to drop work early or to queue it for later (threads have implicit queues on locks, sockets, etc.)
- virtualization makes it harder to know where you stand!

Problems with threads (claimed):

Week 4

- threads limits are too small in practice (about 100); some of this is due to linear searches in internal data structures, or limits on kernel memory allocation
- claims about locks, overhead, TLB and cache misses are harder to understand -- don't seem to be fundamental over events
  - do events use less memory? probably some but not 50% less
  - do events have fewer misses? latencies are the same, so only if working set is smaller
  - is it bad to waste VM for stacks? only with tons of threads
  - is there a framentation issue with stacks (lots of partially full pages)? probably to some degree (each stack needs at least one page. If so, we can move to a model with non-contiguous stack frames (leads to a different kind of fragmentation. If so, we could allocate subpages (like FFS did for fragments), and thread a stack through subpages (but this needs compiler support and must recompile all libraries).
- queues are implicit, which makes it hard to control or even identify the bottlenecks
- key insight in SEDA: no user allocated threads: programmer defines what can be concurrent and SEDA manages the threads. Otherwise no way to control the overall number or distribution of threads

Event-based approach has problems too:

- debugging
- legacy code
- stack ripping

Solution:

- use threads within a stage, events between stages
- stages have explicit queues and explicit concurrency
- threads (in a stage) can block, just not too often
- SEDA will add and remove threads from a stage as needed
- simplifies modularity: queues decouple stages in terms of performance at some cost to latency
- threads never cross stages, but events can be pass by value or pass by reference.
- stage scheduling affects locality -- better to run one stage for a while than to follow an event through multiple stages. This should make up for the extra latency of crossing stages.

Feedback loops:

- works for any measurable property that has smooth behavior (usually continuous as well). Property typically needs to be monotonic in the control area (else get lost in local minima/maxima)
- within a stage: batching controller decides how many events to process at one time
  - balance high throughput of large batches with lower latency of small batches -- look for point where the throughput drops off
- thread pool controller: find the minimum number of threads that keeps queue length low
- global thread allocation based on priorities or queue lengths...

Performance is very good, degrades more gracefully, and is more fair!

- but huge dropped requests to maintain response time goal
- however, can't really do any better than this...

Events vs. threads revisited

- how does SEDA do split-phase actions?
- Intra-stage:
  - threads can just block
  - multiple threads within a stage, so shared state must be protected. Common case is that each event is mostly independent (think http requests)
- Inter-stage:
  - rip action into two stages
  - usually one-way: no return (equivalent to tail recursion). This means that the continuation is just the contents of the event for the next stage.
  - loops in stages are harder: have to manually pass around the state
  - atomicity is tricky too: how do you hold locks across multiple stages? generally try to avoid, but otherwise need one stage to lock and a later one to unlock

IV. Capriccio

Idea: instead of switching to events, let's just fix threads

- leverage async I/O
- scale to 100,000 threads (qualitative difference: one per connection!)
- enable compiler support and invariants

Why user-level threads? (mostly same args from scheduler activations)

- easy, low-cost synchronization (but not for I/O which is slow anyway)
- *control* over thread semantics, invariants
- enable application-specific behavoir (e.g. scheduling) and optimizations
- enable compiler assistance (e.g. safe stacks)

But, still has problems:

- still have two schedulers
- async I/O mitigates this by eliminating largest cause of blocking (in kernel)
- still can block unexpectedly -- page faults or close() example

- current version actually stops running in such cases (so must be rare)
- can't schedule multple processes at user-level, only threads within a process (not a problem for dedicated machines like servers)

Specific:

- POSIX interface for legacy apps, but now at user level with a runtime library
- make all thread ops O(1)
- deal with stack space for 100,000 threads (can't just give each 2MB)
  - this uses less stack space
  - and is faster!
  - and is safer! (any fixed amount might not be enough)

Async I/O

- allows lots of user threads to map to small number of kernel threads
- allows better disk throughput
- differnt mechanisms for network (epoll) and disk (AIO), but this is hidden from users

Resource-aware scheduling:

- not well developed yet
- goal: transparent but application specific (!)
- note: lots of earlier work on OS extensibility that was hard to use and not well justified
- here we are not requiring work on the part of the programmer, and we are focused on servers, which actually do need different support

CS262A: Advanced Topics in Computer Systems
Joe Hellerstein, Spring 2008
UC Berkeley

Concurrency Control and Performance

Agrawal/Carey/Livny: Locking vs. Optimistic

Previous work had conflicting results:
- Carey & Stonebraker (VLDB84), Agrawal & DeWitt (TODS85): blocking beats restarts
- Tay (Harvard PhD) & Balter (PODC82): restarts beat blocking
- Franaszek & Robinson (TODS85): optimistic beats locking

Goal of this paper:
- Do a good job modeling the problem and its variants
- Capture causes of previous conflicting results
- Make recommendations based on variables of problem

Methodology
- simulation study, compare Blocking (i.e. 2PL), Immediate Restart (restart when denied a lock), and Optimistic (a la Kung & Robinson)
- pay attention to model of system:
  - database system model: hardware and software model (CPUs, disks, size & granule of DB, load control mechanism, CC algorithm
  - user model: arrival of user tasks, nature of tasks (e.g. batch vs. interactive)
  - transaction model: logical reference string (i.e. CC schedule), physical reference string (i.e. disk block requests, CPU processing bursts).
  - Probabilistic modeling of each. They argue this is key to a performance study of a DBMS.
- logical queueing model
- physical queueing model

Measurements
- measure throughput, mostly
- pay attention to variance of response time, too
- pick a DB size so that there are noticeable conflicts (else you get comparable performance)

Experiment 1: Infinite Resources
- as many disks and CPUs as you want
- blocking thrashes due to transactions blocking numerous times
- restart plateaus: adaptive wait period (avg response time) before restart
  - serves as a primitive load control!
- optimistic scales logarithmically
- standard deviation of response time under locking much lower

Experiment 2: Limited Resources (1 CPU, 2 disks)
- Everybody thrashes
- blocking throughput peaks at mpl 25
- optimistic peaks at 10
- restart peaks at 10, plateaus at 50 – as good or better than optimistic
- at super-high mpl (200), restart beats both blocking and optimistic

    - but total throughput worse than blocking @ mpl 25
    - effectively, restart is achieving mpl 60
    - load control is the answer here – adding it to blocking & optimistic makes them handle higher mpls better

Experiment 3: Multiple Resources (5, 10, 25, 50 CPUs, 2 disks each)
- optimistic starts to win at 25 CPUs
    - when useful disk utilization is only about 30%, system begins to behave like infinite resources
- even better at 50

Experiment 4: Interactive Workloads
Add user think time.
- makes the system appear to have more resources
- so optimistic wins with think times 5 & 10 secs. Blocking still wins for 1 second think time.

Questioning 2 assumptions:
- fake restart – biases for optimistic
    - fake restarts result in less conflict.
    - cost of conflict in optimistic is higher
    - issue of $k > 2$ transactions contending for one item
    - will have to punish k-1 of them with real restart
- write-lock acquisition
    - recall our discussion of lock upgrades and deadlock
    - blind write biases for restart (optimistic not an issue here), particularly with infinite resources (blocking holds write locks for a long time; waste of deadlock restart not an issue here).
    - with finite resources, blind write restarts transactions earlier (making restart look better)

Conclusions
- blocking beats restarting, unless resource utilization is low
- possible in situations of high think time
- mpl control important. admission control the typical scheme.
    - Restart's adaptive load control is too clumsy, though.
- false assumptions made blocking look relatively worse

Final quote by Wulf!

Advanced Topics in Computer Systems, CS262A

Prof. Eric Brewer

Lottery Scheduling

I. Lottery Scheduling

Very general, proportional-share scheduling algorithm.

Problems with traditional schedulers:

- Priority systems are ad hoc at best: highest priority always wins
- "Fair share" implemented by adjusting priorities with a feedback loop to achieve fairness over the (very) long term (highest priority still wins all the time, but now the Unix priorities are always changing)
- Priority inversion: high-priority jobs can be blocked behind low-priority jobs
- Schedulers are complex and difficult to control

Lottery scheduling:

- Priority determined by the number of tickets each process has: priority is the relative percentage of all of the tickets competing for this resource.
- Scheduler picks winning ticket randomly, gives owner the resource
- Tickets can be used for a wide variety of different resources (uniform) and are machine independent (abstract)

How fair is lottery scheduling?

- If client has probability p of winning, then the expected number of wins (from the binomial distribution) is np.
- Variance of binomial distribution: $\sigma^2 = np(1-p)$
- Accuracy improves with $\sqrt{n}$
- Geometric distribution used to find tries until first win
- Big picture answer: mostly accurate, but short-term inaccuracies are possible; see Stride scheduling below.

Ticket Transfer: how to deal with dependencies

- Basic idea: if you are blocked on someone else, give them your tickets
- Example: client-server
  - Server has no tickets of its own
  - Clients give server all of their tickets during RPC
  - Server's priority is the sum of the priorities of all of its active clients
  - Server can use lottery scheduling to give preferential service to high-priority clients
- Very elegant solution to long-standing problem (not the first solution however)

Scheduling

Ticket inflation: make up your own tickets (print your own money)

- Only works among mutually trusting clients
- Presumably works best if inflation is temporary
- Allows clients to adjust their priority dynamically with zero communication

Currencies: set up an exchange rate with the base currency

- Enables inflation just within a group
- Simplifies mini-lotteries, such as for a mutex

Compensation tickets: what happens if a thread is I/O bound and regular blocks before its quantum expires? Without adjustment, this implies that thread gets less than its share of the processor.

- Basic idea: if you complete fraction f of the quantum, your tickets are inflated by 1/f until the next time you win.
- Example: if B on average uses 1/5 of a quantum, its tickets will be inflated 5x and it will win 5 times as often and get its correct share overall.
- What if B alternates between 1/5 and whole quantums?

Problems:

- Not as fair as we'd like: mutex comes out 1.8:1 instead of 2:1, while multimedia apps come out 1.92:1.50:1 instead of 3:2:1
- Practice midterm question: are these differences statistically significant? (probably are, which would imply that the lottery is biased or that there is a secondary force affecting the relative priority)
- Multimedia app: biased due to X server assuming uniform priority instead of using tickets. Conclusion: to really work, tickets must be used everywhere. Every queue is an implicit scheduling decision... Every spinlock ignores priority...
- Can we force it to be unfair? Is there a way to use compensation tickets to get more time, e.g., quit early to get compensation tickets and then run for the full time next time?
- What about kernel cycles? If a process uses a lot of cycles indirectly, such as through the ethernet driver, does it get higher priority implicitly? (probably)

Stride Scheduling: follow on to lottery scheduling (not in paper)

- Basic idea: make a deterministic version to reduce short-term variability
- Mark time virtually using "passes" as the unit
- A process has a stride, which is the number of passes between executions. Strides are inversely proportional to the number of tickets, so high priority jobs have low strides and thus run often.
- Very regular: a job with priority p will run every 1/p passes.
- Algorithm (roughly): always pick the job with the lowest pass number. Updates its pass number by adding its stride.
- Similar mechanism to compensation tickets: if a job uses only fraction f, update its pass number by $f \times stride$ instead of just using the stride.
- Overall result: it is far more accurate than lottery scheduling and error can be bounded

Scheduling

absolutely instead of probabilistically

CS262, Fall 2008 : Parallel DBs and MapReduce

- **Parallel Databases**
  - History of Database Machines vs. Commodity HW.
  - Side benefit of relational model: parallelism.
- Basic concepts
  - Pipeline vs. partition parallelism
  - Speedup (fixed problem size) vs. Scaleup (problem and HW grow)
  - Barriers to parallelism: Startup, Interference and Skew
    - Note in paper about interference: 1% slowdown limits scaleup to 37x
  - Shared mem & disk revisited
    - Inevitably you want caching, leading to processor affinity. Why not code it up?
- DB dataflow models: iterators and pipelines. more on this next time!
  - Hashjoin and sort algorithms.
- Sort benchmarks, balancing the HW pipeline
  - Ramification for manycore? datacenters?
  - The "hard stuff": DB layout, query optimization, mixed workloads, UTILITIES!
- **MapReduce**
  - Goals
    - automatic parallelization/distribution
    - fault-tolerance
    - I/O scheduling
    - status/monitoring
  - Structure
    - Pasted Graphic
    - map (k1, v1) -> list(k2, v2)
    - reduce(k2, list(v2)) -> list(whatever)
- Platform:
  - Commodity PCs (dual processor), commodity NW, 100's/1000's of machines, cheap IDE disks
  - GFS for reliable file storage
  - job = {tasks}, passed to scheduler
- Basic Execution:
  - data "automatically" partitioning into M "splits". Reduce done by hashing mod R. M and R specified by the user.
  - Splits are of a single "file" into physical chunks (# of Bytes)
  - Mappers write to memory buffers. Periodically, buffers are flushed locally. Location sent to master.
  - Master notifies Reduce workers about flushed results of Mappers. Fetches them via RPC.
  - Reduce then performs Sort-based groupBy. Output appended to final output file in GFS for this reduce partition.
  - When all M's and R's done, Master wakes up user program to "return" from the MapReduce call. R Reduce files are left in place, possibly to be reused in another MapReduce stage.
- FT:

CS262, Fall 2008: Parallel DBs and MapReduce

- Master pings workers for failure.
- Completed map tasks are reset as "idle" (i.e. not yet started) because they're inaccessible on the failed node's disk. In-Progress Maps and Reduces also set as "idle". Completed Reduces are safely in GFS.
- All reducers need to be told of a failure, due to "pull" model.
- Master failure can be handled by checkpointing of worker state.
- To ensure correctness, need ATOMIC commit of map and reduce. So tentatively write to private temp files (R for M's, 1 for R's).
- When Mapper completes, it sends message to master with the names of the R files, which it records in its data structure. Subsequent such messages ignored.
- When Reduce completes, worker renames temp output to final output name. Atomic rename in GFS ensures that only one of possibly many redundant reducers "wins".
- Clearly, non-deterministic functions provide loose semantics.

- Locality:
  - Master assigns Mappers with an understanding of where the GFS blocks of the input are. Best on a machine with a replica, else "close" in the network (same switch).
- Granularity:
  - Many more M's and R's than machines. Good for load-balancing, and you need to LB after failures. Since master has to bookkeep the M's and R's -O(M*R) state and O(M+R) scheduling decisions, don't want this insanely big. Also, may not want too many R's cos result is spread into many files.
  - Rule of thumb: choose M to be about 16MB-64MB, R a small multiple (e.g. 100) of the #machines. 2K machines, M=200K, R=5K.
- Stragglers:
  - Note causes: faulty though correctable disks, shared resource utilization, bugs in code.
  - One solution: competition. Redundant execution of the last in-progress tasks. When useful, when not?
- Combiner function: for commutative/associative Reduce
  - partial pre-aggregation at the mapper. output to the local intermediate file to be sent to reducer.
- Other stuff
  - Side effects: up to the programmer to make atomic and idempotent.
  - Optionally skip bad input records: signal handler sends a UDP packet with seqno to Master. If it sees >1 such, skips it next time.
  - Sequential local version for debugging.
  - HTTP server in master for status, stderr, stdout
  - Running aggregation of "counters" for sanity check
- Performance
  - 1800 machines, 2GHz Xeons, 4GB RAM< 160GB disk (!!), Gb Ethernet. Two level tree switched net with 100-200 Gbps aggregate at root
  - Grep experiment: see startup cost in graph -1 minute startup, 150 seconds total! Startup includes code propagation, opening 1000 files in GFS, getting GFS metadata for locality optimization. WOW.

CS262, Fall 2008: Parallel DBs and MapReduce

- Sort: 891 seconds, 1800 machines, 3600 disks. 1057 seconds was TeraSort benchmark. (2006 was 435 Secs on **80 Itanium machines** with 2520 disks, 2000 was 1952 IBM SP machines with 2168 disks!)

- Note FLuX project (ICDE 03, SIGMOD 04)
  - Fully pipelined
  - Process pairs + Partitioning
  - quite complex, but more powerful
- Big picture questions
  - when is complexity merited? Architectural elegance in MapReduce?
  - Programming models

Advanced Topics in Computer Systems
Eric Brewer (based on notes from Joe Hellerstein)

SQL Query Optimization

Basics

Given: A query joining *n* tables
The "Plan Space": Huge number of alternative, semantically equivalent plans.
The Perils of Error: Running time of plans can vary by many orders of magnitude
Ideal Goal: Map a declarative query to the most efficient plan tree.
Conventional Wisdom: You' re OK if you avoid the rotten plans.
Industrial State of the art: Most optimizers use System R technique and work "OK" up to about 10 joins.

> Wide variability in handling complex queries with aggregation, subqueries, etc. Wait for rewrite paper.

Approach 1: The Optimization Oracle

(*definitely* not to be confused with the company of the same name)
You' d like to get the following information, but in 0 time:

- Consider each possible plan in turn.
- Run it & measure performance.
- The one that was fastest is the keeper.

Approach 2: Make Up a Heuristic & See if it Works

University INGRES

- always use NL-Join (indexed inner whenever possible)
- order relations from smallest to biggest

Oracle 6

- "Syntax-based" optimization

OK,OK. Approach 3: Think!

Three issues:

- define plan space to search
- do cost estimation for plans
- find an efficient algorithm to search through plan space for "cheapest" plan

Selinger & the System R crowd the first to do this right. The Bible of Query Optimization.

SQL Refresher

```
SELECT {DISTINCT} <list of columns>
FROM <list of tables>
{WHERE <list of "Boolean Factors" (predicates in CNF)>}
{GROUP BY <list of columns>
{HAVING <list of Boolean Factors>}}
{ORDER BY <list of columns>};
```

Semantics are:

1. take Cartesian product (a/k/a cross-product) of tables in FROM clause, projecting to only those columns that appear in other clauses
2. if there's a WHERE clause, apply all filters in it
3. if there's a GROUP BY clause, form groups on the result
4. if there's a HAVING clause, filter groups with it
5. if there's an ORDER BY clause, make sure output is in the right order
6. if there's a DISTINCT modifier, remove dups

Of course the plans don't do this exactly; query optimization interleaves 1 & 2 into a plan tree. GROUP BY, HAVING, DISTINCT and ORDER BY are applied at the end, pretty much in that order.

Plan Space

All your favorite query processing algorithms:

- sequential & index (clustered/unclustered) scans
- NL-join, (sort)-merge join, hash join
- sorting & hash-based grouping
- plan flows in a non-blocking fashion with get-next iterators

Note some assumptions folded in here:

- selections are "pushed down"
- projections are "pushed down"
- all this is only for single query blocks

Some other popular assumptions (System R)

- only left-deep plan trees
- avoid Cartesian products

Cost Estimation

The soft underbelly of query optimization.

Requires:

- estimation of costs for each operator based on input cardinalities
  - both I/O & CPU costs to some degree of accuracy
- estimation of predicate selectivities to compute cardinalities for next step
- assumption of independence across predicates

- decidedly an inexact science.
- "Selingerian" estimation (no longer state of the art, but not really so far off.)
    - # of tuples & pages
    - # of values per column (only for indexed columns)
        - These estimations done periodically (why not all the time?)
    - back of envelope calculations: CPU cost is based on # of RSS calls, no distinction between Random and Sequential IO
    - when you can't estimate, use the "wet finger" technique
- New alternative approaches:
    - Sampling: so far only concrete results for base relations
    - Histograms: getting better. Common in industry, some interesting new research.
    - Controlling "error propagation"

Searching the Plan Space

- Exhaustive search
- Dynamic Programming (prunes useless subtrees): System R
- Top-down, transformative version of DP: Volcano, Cascades (used in MS SQL Server?)
- Randomized search algorithms (e.g. Ioannidis & Kang)
- Job Scheduling techniques
- In previous years we read many of these (they're fun!), but it is arguably more relevant to talk about query rewriting.

The System R Optimizer's Search Algorithm

Look only at left-deep plans: there are $n!$ plans (not factoring in choice of join method)
Observation: many of those plans share common prefixes, so don't enumerate all of them
Sounds like a job for … Dynamic Programming!

1. Find all plans for accessing each base relation
    - Include index scans when available on "SARGable" predicates

2. For each relation, save cheapest unordered plan, and cheapest plan for each "interesting order". Discard all others.
3. Now, try all ways of joining all pairs of 1-table plans saved so far. Save cheapest unordered 2-table plans, and cheapest "interesting ordered" 2-table plans.
    - note: secondary join predicates are just like selections that can't be pushed down

4. Now try all ways of combining a 2-table plan with a 1-table plan. Save cheapest unordered and interestingly ordered 3-way plans. You can now throw away the 2-way plans.
5. Continue combining k-way and 1-way plans until you have a collection of full plan trees
6. At top, satisfy GROUP BY and ORDER BY either by using interestingly ordered plan, or by adding a sort node to unordered plan, whichever is cheapest.

Some additional details:

- don't combine a *k*-way plan with a 1-way plan if there's no predicate between them, unless all predicates have been used up (i.e. postpone Cartesian products)
- Cost of sort-merge join takes existing order of inputs into account.

Evaluation:

- Only brings complexity down to about $n2^{n-1}$, and you store choose(n, ceil(n/2)) plans
- But no-Cartesian-products rule can make a big difference for some queries.
- For worst queries, DP dies at 10-15 joins
- adding parameters to the search space makes things worse (e.g. expensive predicates, distribution, parallelism, etc.)

Simple variations to improve plan quality:

- bushy trees: (2(n-1))! / (n-1)! plans, DP complexity is $3^n - 2^{n+1} + n + 1$ need to store 2^n plans (actually it's worse for subtle reasons)
- consider cross products: maximizes optimization time

Subqueries 101: Selinger does a very complete job with the basics.

- subqueries optimized separately
- uncorrelated vs. correlated subqueries
    - uncorrelated subqueries are basically constants to be computed once in execution

      correlated subqueries are like function calls

CS262A: Advanced Topics in Computer Systems
Eric Brewer, based on notes by Joe Hellerstein

Volcano & the Exchange Operator

There are a host of techniques for parallelizing particular query operators (e.g. hash join, sorting, etc.), but what you really need is to parallelize your query *engine* in a clean, uniform way.

Volcano's Solution: encapsulate the parallelism in a query *operator* of its own, not in the QP infrastructure.

Overview: kinds of intra-query parallelism available:

- pipeline
- partition, with two subcases:
    - intra-operator parallelism (e.g. parallel hash join, or parallel sort)
    - inter-operator parallelism -- bushy trees

We want to enable all of these -- including setup, teardown, and runtime logic -- in a clean encapsulated way.

The **exchange** operator: an operator you pop into any single-site dataflow graph as desired -- anonymous to the other operators.

Implementation:

- Note: Volcano was done with processes, but today you'd use threads
- splits the graph into two threads. The lower thread has an X-OUT iterator at the top. The upper thread has an X-IN iterator at the botton.
- X-OUT is a driver for the lower iterator. Says next() a bunch of times, constructs a packet, *pushes* that packet via IPC or network comm onto a queue in X-IN's "port". X-IN responds to next() when it has tuples in its queue.
- Flow control: semaphore on the port dictates the maximum degree to which the producer can get ahead of the consumer. This is akin to a bounded queue.
- Note that introducing a queue allows a push producer to work with a pull consumer. The queue allows a *bounded drift* in their rates of production, beyond which one side is blocking/polling.

Benefits of exchange:

1. opaquely handles setup and teardown of clones (in an SMP...for shared-nothing, would need to have daemons at each site, and a protocol to request clone spawning)
2. at the top of a local subplan, allows pipeline parallelism: turns iterator-based, unithreaded "pull" into network-based, cross-thread "push".
    - Why is push beneficial?
    - at the top of a local subplan, allows decoupling of children's scheduling.
    - inside a subplan, can mix pull and push to get the best of both

"Extensibility" features of Volcano and exchange:

- operators don't interpret records, *support functions* do; goes for partitioning as well

There were a couple subsequent extensions to Exchange:

- Graefe has another paper on exchange in Transactions on Software Engineering which provides more gory details on startup and teardown of clones. It's not all that pretty, unfortunately.
- The River project at Berkeley revisited partitioned parallelism with an eye toward adaptive load balancing for state-agnostic operators (each data item can go into any consumer partition).
- FLuX (Fault-tolerant, Load-Balancing eXchange) was an effort at Berkeley to extend Exchange to add what the name says.
- Google MapReduce basically applies partition parallelism to a simple dataflow pipeline, and demonstrates broad applicability in their workloads. It also includes simple fault-tolerance and load balancing techniques -- simpler than River or Flux, yet effective enough for their workloads. A related paper on Sawzall proposes a "little language" for this environment, which should be contrasted with a dataflow or query language.

Food for thought:

- As we'll see again, encapsulating communication/flow details is a good programming paradigm. Is there a broader lesson here for asynchronous programming models, and particularly distributed or parallel programming models? Volcano was in fact supposed to be targeted at more general parallel programming, though they only made a few steps in that direction.
- Note that an optimizer chooses where to pop exchange ops into a plan. What does this suggest in concert with the above point? If we program right, can this kind of decision be made by an optimizer in more general programming tasks? Can query optimization be brought to bear in more generic contexts?
- What about eddies and exchange -- can we make the use of exchange operators dynamic, and dynamically control the points and degrees of parallelism?

Eddies

Starting point: observes that a query optimizer is an adaptive system with a very slow feedback loop:

1. Observe environment: daily/weekly (runstats)
2. Use observations to choose behavior: query optimization
3. Take action: query execution

There are reasons to believe this is way too slow. People have looked at more intelligent things (see survey article for more detail):

- **Per-query adaptivity:** piggyback statistics-gathering on query execution. [Chen/Roussopoulos 94].

- **Runtime sampling:** Take samples of the database right at runtime to estimate costs. (many papers starting in the late 80's)
- **Runtime "competition":** for initial phase of a query, try multiple plans, and then choose the best alternative. [Antoshenkov, DEC RDB, 96]. Only used for base-table access method selection.
- **Inter-operator adaptivity**: Place a materialization operator in the plan. Re-optimize after the materialization operator runs. e.g. [Kabra/DeWitt98]
- **Adaptive operators:** Some good work was done on making big operators (e.g. hashjoin, sort) adaptive to changing memory availability. e.g. [Pang/Carey/Livny 93]. Also some work on making join algorithms for interactive query systems that favor one input or the other based on user feedback [Haas/Hellerstein 97]
- **Adaptive partitioning:** River [Arpaci/etal. 99] adaptively decided how to partition a dataflow a la Exchange.

Eddies were an effort to subsume a bunch of this stuff using the design spirit of Exchange: encapsulate the decisions in a dataflow operator. An eddy allows for adaptive reordering of a subtree of dataflow operators on a *tuple by tuple basis* (or slower, of course). Here's the idea:

- The eddy is "wired up" so that its inputs are the inputs to the subtree, its output is the output of the subtree, and all the operators in the subtree are both inputs and outputs of the eddy. Note that an eddy can service *all* the operators in a plan, or it can just provide flexibility for a subset of operators.
- The eddy is parameterized by a *partial ordering* on the operators, which tells it which operators must precede which in the dataflow, and which ones can be mutually reordered.
- If all the operators are pipelining (in the "Joe Hellerstein rule" sense of producing tuples while consuming), then the eddy gets to:
    1. Observe the rates of production/consumption for operators
    2. Choose the order of operators that each tuple visits.
- In essence, the choice of the dataflow graph edges has been replaced by a "routing policy".
- The eddy operator can therefore serve as a single encapsulated place for control logic (in the sense of control theory).
- To ensure that each tuple visits each operator at most once and in an order consistent with the partial order given, a "steering vector" of ready/done bits is attached to each tuple to guide

Note a vague similarity to INGRES' optimization scheme, which also could change join orders "per tuple" in some sense. At the architectural level, that's all fine and dandy. But many questions remain. Some basic ones:

- Many people with competing schemes accused eddies of overkill and efficiency: isn't the overhead of all this tuple massaging too high? You can't really want or need to adapt on a per-tuple basis? This was addressed with a simple batching scheme described in a short paper with a performance study in Postgres.
- What is an optimal routing policy? How do you even define the problem? This is tricky. It's useful to start with the simpler problem of eddies over unary filters -- i.e. selections or key-based index joins (e.g. web lookups). Even here the problem is tricky, and depends how you define it. For a stable data distribution, an

approximation algorithm was developed There's a natural though imperfect analogy to "n-arm bandit" problems. There are also some complexity results and worst-case bounds for different models of correlation among predicates. An interesting heuristic appeared in VLDB this year [Babu 2005]

More complicated questions revolving around joins remain:

- Is it always OK to mess with tuple routing among joins, or can that give you wrong answers? *Moments of symmetry* was an intuitive, informal handle on that.
- Join output requires feeds on both inputs. The initial delay problem in the paper.
- Can we enable the join *algorithm* to change, or are we limited to the join ordering alone?
- Joins carry a "burden of history": once potentially joinable tuples have been sent to separate join operators, there is no way to create *any* output product using those tuples with a join order that combines them first. E.g. the initial delay problem is like this: S tuples were sent to the join of R and S, T tuples were sent to the join of S and T. If S tuples could be easily filtered by T and are expensive to join with R, once the R's come in it's too late to change your mind.

Many of these problems were subsequently addressed by choosing a different granularity of dataflow operator. Instead of using eddies and joins, you expose the "state modules" ("STeMs") (hashtables, b-trees) from the join directly to the eddy -- in essence you expose the join algorithm's internals to the eddy. This idea led to:

- competition and hybridization of multiple join algorithms (hash join and index join) at runtime [Raman, 2003]
- user-controllable partial results from queries[Raman, 2002]
- solutions to the initial delay and "burden of history" problems (via STAIRS) [Deshpande, 2004]

The end result of this was that we tore apart traditional relational query processing and optimization and reexamined it. However, we certainly did not put it back together (yet)! The set of new variables exposed introduces a bunch of complexity, and naturally reopens buried chestnuts like dealing with dependencies in data and predicates. *Much remains to be done here!* The question is relevance: one can come up with many scenarios where adaptivity helps a lot, but are any of them enough to rearchitect a DBMS?

My take: maybe not in the traditional DBMS market. Maybe in the brave new world of software dataflow for other tasks, e.g. network routing!

Click Modular Router

I. Click Modular Router

Key ideas:

- Static graph of elements -- can verify well-formed connections
- Extenibility through interposition or redefinition -- every wire and every element is a place for possible extension
- Both push and pull processing
- Kinds of state: local, global (poor), configurations (static) and flow based
- Flexibility with only minor performance hit (mostly due to virtual functions)

Push vs. Pull:

- in/out can be push, pull or agnostic
- pull is call/return from destination, just like database query processing
- push is driven by the sender, like upcalls
- pull is good for polling and scheduling
- push is good for arrivals (and events in general)
- a queue is a push to pull converter
- Volcano's "exchange operator" is a pull to push converter, not clear if Click has an analog, although you could build one. Might be useful for multiprocessor version?

Graph properties:

- connections must match: pull to pull, push to push, or either to agnostic
- statically checked, and all agnostic ports are resolved (to push/pull) statically
- elements have configuration parameters and initilization (like "open")
- no obvious equivalent for "close", although there is an atomic update to the whole graph
- push outputs have one wire (otherwise implies duplicating output)
- pull inputs have one wire (otherwise have to decide which one to call, or merge them)
- push inputs and pull outputs can have multiple wires, first-come first served
- declarative language (for the graph) => room for optimization (see his PhD)

Scheduling:

Click Router

- single thread (!)
- only some elements need to be scheduled (tasks) -- those that push or pull independent of a caller
- most just execute when called (pushed or pulled)
- stride scheduling for tasks (why is this a good idea?)
- tasks must yield the thread by returning -- not clear how long tasks should run or how it affects future scheduling decisions
- deadlock? likely if any element can block, since there is no pre-emption and no other threads. exception: block for arrival might work (since that is a form of concurrency)
- Polling vs. Interrupts: polling is much faster (8x?); processor stays in the sweet spot (no pipeline flushes), better caching, and easy to batch arrivals for lower overhead (and even more locality)
- "receiver livelock" -- problem in which interrupts are arriving faster than the handling rate, which prevents any real work from getting done. Polling doesn't have this problem, but you must drop packets at poll time to keep it in the operating range. Note how flat the throughput curves are when the reach saturation -- excellent "graceful degradation", much like SEDA
- Note the use of shared memory to get packets, much like U-Net.

Flow Context:

- Queried at initialization time to figure out properties about your context (neighbors)
- Example properties: nearest queue, how many queues downstream or upstream, what interface is this from
- Enables elements to behave differently based on context
- Best example is looking at the total downstream queue length for congestion estimation
- Side note: RED is "random early detection" (Floyd et al. 93). Key idea is to drop packets with (linearly) higher probability as the queue length gets longer, starting at some min. Attacks congestion proactively and earlier, hoping to keep the delays low.
- In a DBMS, these kinds of things are done by the query optimizer, which typically has lots of specialized operators (and parameters) to use in different contexts

Initialization:

- broken into stages to avoid cycle dependencies
- not clear who decides what goes in each stage; presumably the programmer has to participate

Annotations:

- Idea: add metadata to packet to carry information downstream
- Statically defined fields

Click Router

- In theory every producer of an annotation should have exactly one consumer on all possible paths (where discard counts as a consumer). This does not appear to be enforced (or enforcable), since the annotations are not part of the language.
- This is a bit of a hack: it is its own system that can't be type checked or verified, but it still required for correct operation.
- Also no support for global information. Not clear why this was avoided. (In general, larger scopes are bad, but if you need one, it is better to create one than to abuse other mechanisms like configurations or annotations.)

Peformance:

- overall excellent
- polling really helps
- flexibility costs about 1us per packet -- mostly due to virtual functions (in C++). This is because every element is a subclass with lots of virtual functions, so every push/pull call is a virtual function call with a dynamic level of indirection. However, since the graph is really static, you can flatten the virtual functions either statically or (in theory) at initialization time.
- Supports its claims of extensibility and flexibility quite well
- Supports its claim of low overhead for this flexibility

Are Virtual Machine Monitors Microkernels Done Right? Xen and the Art of Virtualization

I. Background

Virtual machines:

- Observation: instruction-set architectures (ISA) form some of the relatively few well-documented complex interfaces we have in world
- Machine interface also includes the meaning of interrupt numbers, programmed I/O, DMA, etc.
- Anything that implements this interface can execute the software for that platform
- A *virtual machine* is a software implementation of this interface (often using the same underlying ISA, but not always)

Examples:

- IBM initiated VM idea to support legacy binary code; i.e. support the interface to an old machine on a newer machine
- Apple does this to run old Mac programs on newer machines (with a different ISA)
- MAME is an emulator for old arcade games (5800+ games!!) that actually execute the game code straight from a ROM image.
- Modern VM research started with the Disco project at Stanford, which ran multiple virtual machines on a large shared-memory multiprocessor (since normal OS's couldn't scale well to lots of CPUs). This led to VMWare.
- VMWare: lots of uses including customer support (support many variations/versions on one PC using a VM for each), web hosting, app hosting (host many independent low-utilization servers on one machine -- "server consolidation")

VM Basics:

- The real master is no longer the OS but the "Virtual machine monitor" (VMM) or "hypervisor" (hypervisor > supervisor)
- OS no longer runs in the most privileged mode (which is reserved for the VMM)
- But OS thinks it is running in most privileged mode and still issues those instructions?
  - Ideally, such instructions should cause traps and the VMM then emulates the instruction to keep the OS happy
  - But in x86, some such instructions fail silently! VMWare solution: dynamically rewrite binary code to replace those instructions with traps; Xen solution: rewrite the OS slightly to avoid them
  - x86 has four privilege "rings" with ring 0 having full access. VMM = ring 0, OS = ring, app = ring 3 (x86 rings come from Multics, as do x86 segments)

Virtual Machines

How accurate is the emulation?

- Original paper on whether or not a machine is virtualizable: Gerald J. Popek and Robert P. Goldberg (1974). "Formal Requirements for Virtualizable Third Generation Architectures". *Communications of the ACM* 17 (7): 412 –421.
- Disco/VMWare/IBM: Complete: runs unmodified OSs and applications
- Dinali: introduced the idea of "paravirtualization" -- change interface some to improve VMM performance/simplicity
  - Must change OS and some apps (e.g. those that use segmentation)
  - But can support 1000s of VMs on one machine...
  - Great for web hosting
- Xen: change OS but not applications (support the full *application binary interface* (ABI)
  - Faster than full VM -- supports ~100 VMs per machine
- Moving to a paravirtual VM is essentially porting the software to a very similar machine

II. Disco: Emulate the MIPS interface

1) emulate R10000

- simulate all instructions: most are directly executed, privileged instructions must be emulated, since we won't run the OS in privileged mode (Disco runs privileged, OS runs supervisor mode, apps in user mode)
- an OS privileged instruction causes a trap which causes Disco to emulated the intended instruction
- map VCPUs onto real CPU: registers, hidden registers

2) MMU and physical memory

- virtual memory -> virtual physical memory -> machine memory
- VTLB is a Disco data structure, maps VM -> PM
- TLB holds the "net" mapping from VM -> MM, by combining VTLB mapping with Disco's page mapping, which is PM -> MM
- on TLB instruction on the VCPU, Disco gets trap, updates the VTLB, computes the real TLB entry by combined VTLB mapping with internal PM->MM page table (taking the permission bits from the VTLB instruction)
- Must flush the real TLB on VM switch
- Somewhat slower:
  - OS now has TLB misses (not direct mapped)
  - TLB flushes are frequent
  - TLB instructions are now emulated
- Disco maintains a second-level cache of TLB entries: this makes the VTLB seem larger

than a regular R10000 TLB. Disco can thus absorb many TLB faults without passing them through to the real OS

3) I/O (disk and network)

- emulated all programmed I/O instructions
- can also use special Disco-aware device drivers (simpler)
- main task: translate all I/O instructions from using PM addresses to MM addresses

Optimizations:

- larger TLB
- copy-on-write disk blocks
  - track which blocks already in memory
  - when possible, reuse these pages by marking all versions read-only and using copy-on-write if they are modified
  - => shared OS pages and shared executables can really be shared.
- zero-copy networking along fake "subnet" that connect VMs within an SMP. Sender and receiver can use the same buffer (copy on write)

III. Xen: emulate x86 (mostly)

Key idea: change the machine-OS interface to make VMs simpler and higher performance

- Called *paravirtualization* (due to Denali project)
- Pros: better performance on x86, some simplifications in VM implementation, OS might want to know that it is virtualized! (e.g. real time clocks)
- Cons: must modify the guest OS (but not its applications!)
- Aims for performance isolation (why is this hard?)
- Philosophy: divide up resources and let each OS manage its own; ensures that real costs are correctly accounted to each OS (essentially zero shared costs, e.g., no shared buffers, no shared network stack, etc.)

x86 harder to virtualize than Mips (as in Disco):

- MMU uses hardware page tables
- some privileged instructions fail silently rather than fault; VMWare fixed this using binary rewrite --- Xen by modifying the OS to avoid them

Step 1: reduce the privilege of the OS

- "hypervisor" runs with full privilege instead (ring 0), OS runs in ring 1, apps in ring 3
- Xen must intercept interrupts; converts them to events posted to shared region with OS
- need both real and virtual time (and wall clock)

Virtualizing virtual memory:

Virtual Machines

- x86 does not have software TLB
- good performance requires that all valid translations should be in HW page table
- TLB not “tagged”, which means address space switch must flush TLB
- 1) map Xen into top 64MB in all address spaces (and limit guest OS access)
- 2) guest OS manages the hardware page table(s), but entries must be validated by Zen on updates; guest OS has read-only access to its own page table
- Page frame states: PD=page directory, PT=page table, LDT=local descriptor table, GDT=global descriptor table, RW=writable page. The type system allows Xen to make sure that only validated pages are used for the HW page table.
- Each guest OS gets a dedicated set of pages, although size can grow/shrink over time
- Physical page numbers (those used by the guest OS) can differ from the actual hardware numbers; Xen has a table to map HW->Phys; and each guest OS has a Phy->HW map. This enables the illusion of physically contiguous pages.

Network:

- Model: each guest OS has a virtual network interface connected to a virtual firewall/ router (VFR). The VFR both limits the guest OS and also ensure correct incoming packet dispatch.
- Exchange pages on packet receipt (to avoid copying); no frame available => dropped packet
- Rules enforce no IP spoofing by guest OS
- Bandwidth is round robin (is this “isolated”?)

Disk:

- virtual block devices (VBDs): similar to SCSI disks
- management of partitions, etc. done via domain0
- could also use NFS or network-attached storage instead

Domain 0:

- nice idea: run the VMM management at user level
- easier to debug and hypercall checks catch potential errors (narrow interface!)

IV. Microkernels vs VMMs

Microkernel idea: create a small OS kernel and then run other aspects of the traditional OS, such as the file system, in other processes.

- Simple modular OS components
- Requires good IPC performances (to communicate among the now separate parts of the OS)
- Mach is one version, Windows started this way as well (due to Mach), but slowly moved pieces back into one monolithic OS (e.g. graphics)

Virtual Machines

- Small TLBs also hurt microkernels, since more processes need to be resident at once (but benchmarks were done with few processes so this didn't affect architecture much!)
- VMM view: divide up into essentially *non-communicating* pieces and switch among them -- no need for good IPC performance and *no dependencies* among the pieces
- Interprocess dependencies reduce reliability in practice: who is responsible for all of these modules? can you really make your own module effectively in practice?
- Xen: focus thus on *dividing* up resources, not managing them!
- Parallax is a file system that runs in another domain, more like a mounted file system

Live Migration of Virtual Machines
ReVirt: Virtual-Machine Logging and Replay

Goal of this lecture: illustrate some of value of leveraging the VMM interface for new properties; we cover two here (migration and exact replay), but there are many others as well including debugging and reliability.

I. Live Migration

Migration is useful:

- Load balancing for long-lived jobs (why not short lived?)
- Ease of management: controlled maintenance windows
- Fault tolerance: move job away from flaky (but not yet broken hardware)
- Energy efficiency: rearrange loads to reduce A/C needs
- Data center is the right target

Two basic strategies:

- Local names: move the state physically to the new machine
  - Local memory, CPU registers, local disk (if used -- typically not in data centers)
  - Not really possible for some physical devices, e.g. tape drive
- Global names: can just use the same name in the new location
  - Network attached storage provides global names for persistent state
  - Network address translation or layer 2 names allows movement of IP addresses

Historically, migration focused on processes:

- Typically move the process and leave some support for it back on the original machine
  - e.g. old host handles local disk access, forwards network traffic
  - these are “residual dependencies” -- old host must remain up and in use
  - Hard to move exactly the right data for a process -- which bits of the OS must move?
  - E.g. hard to move TCP state of an active connection for a process
- See Zap paper for best of process-based migration

VMM Migration:

- Move the whole OS as a unit -- don’t need to understand the OS or its state
- Can move apps for which you have no source code (and are not trusted by the owner)
- Can avoid residual dependencies in data center thanks to global names
- Non-live VMM migration is also useful:

Virtual Machines 2

- migrate your work environment home and back: put the suspended VMM on a USB key or send it over the network
- Collective project, “Internet suspend and resume”

Goals:

- Minimize downtime (maximize availability)
- Keep the total migration time manageable
- Limit the impact of migration on both the migratee and the local network

Live migration approach:

- Allocate resources at the destination (to ensure it can receive the domain)
- Iteratively copy memory pages to the destination host
  - Service continues to run at this time on the source host
  - Any page that gets written will have to be moved again
  - Iterate until a) only small amount remains, or b) not making much forward progress
  - Can increase bandwidth used for later iterations to reduce the time during which pages are dirtied
- Stop and copy the remaining (dirty) state
  - Service is down during this interval
  - At end of the copy, the source and destination domains are identical and either one could be restarted
  - Once copy is acknowledged, the migration is *committed* in the transactional sense
- Update IP address to MAC address translation using “gratuitous ARP” packet
  - Service packets starting coming to the new host
  - May lose some packets, but this could have happened anyway and TCP will recover
- Restart service on the new host
- Delete domain from the source host (no residual dependencies)

Types of live migration:

- Managed migration: move the OS without its participation
- Managed migration with some paravirtualization
  - Stun rogue processes that dirty memory too quickly
  - Move unused pages out of the domain so they don’t need to be copied
- Self migration: OS participates in the migration (paravirtualization)
  - harder to get a consistent OS snapshot since the OS is running!

Excellent results on all three goals:

- downtimes are very short (60ms for Quake 3 !)
- impact on service and network are limited and reasonable
- total migration time is minutes
- Once migration is complete, source domain is completely free

Virtual Machines 2

II. ReVirt

Idea: use VM interface to replay non-deterministic attacks exactly

The overall problem:

- Many ways to take over a machine; even the kernel has many bugs
- The number of ways and sophistication is increasing (CERT advisories)
- The PC is too complex to be correct
- So, can't really prevent problems, but we can eliminate holes as we find them
- Goal: make it much easier to find exploited hole after an attack

Basic approach: log the events that took place (audit trail) so that we can reconstruct the attack

- Problem 1: integrity: attacker can change/remove the logs
- Problem 2: the logs are incomplete and may miss key events
- Problem 3: the events may not be enough to recreate non-deterministic bugs; can't decrypt past traffic either since keys are one thing that is non-deterministic

ReVirt approach:

- 1) Log in the VMM to ensure integrity
  - Compromised OS does not have access to VMM logs, even if logger runs in another domain (rather than in the VMM proper)
  - Small code base reduces bugs in VMM
- 2) record non-deterministic events using logs and checkpoints and replay everything **exactly** to pinpoint the exploit
- Narrow VMM interface makes it possible to log all events well -- no need to understand drivers, hardware, etc.

Alternative solutions:

-

OS on OS:

- User mode linux: run linux OS as an "app" inside the real OS
- This is a kind of paravirtualization
- Must map all low-level events to Unix signals
- Use host OS devices instead of raw devices
- ReVirt would work better on top of Xen (which is newer), which is both faster and has a much smaller "trusted computing base"

General replay approach:

- start from a safe checkpoint, then roll forward using the log, watching for the exploit
- May not find it on the first pass, but should learn something. So restart and roll forward again based on new information; repeat. In theory, could implement a "step backward" debugging option, which would restore the checkpoint and roll forward *almost* all the

way to the present (except for the last "step")

- Insight: only need to log non-deterministic events and inputs, the rest by definition are deterministic and will replay naturally
  - Time: must record the exact time of each event and replay it at the same (virtual) time. For example, replay an interrupt during the exact same *instruction* as before.
  - Input (keyboard, mouse, network packet, etc.): must log the exact input as well as its time. This is easy except for packets, which can consume much space.

ReVirt logging specifics:

- Copy disk image as first checkpoint
- Log events in the VMM into a circular buffer, then periodically move to disk
- For events, log the instruction and the # of branches since the last interrupt; together these define the exact time of the event
- For input, log data from virtual devices: disk, network, real-time clock, keyboard, mouse, CD-ROM, floppy. May not need to log the actual data, if you expect it to be around for replay (e.g. disk blocks)
- For non-deterministic x86 instructions, can trap and emulate to ensure deterministic results (e.g. read cycle counter). (ReVirt actually used paravirtualization instead of emulation.)

ReVirt playback specifics:

- 1) Prevent new events from disturbing playback
- 2) Load checkpoint
- 3) Deliver each event at the precise time. For interrupts: set branch counter to trap every 128 branches until you get close to the right time, then set breakpoint on specific instruction. On each break, check the branch counter to see if this is the right one. If so, issue interrupt. (Breakpoints are slower, so only use them when you get close.)
- Most of playback is near real time.
- Can also replay on a different machine! (similar to live migration)

Tools:

- Continuing from replay: tool can start live from the middle of the replay. This is useful for examining the state of the machine using traditional tools
- Offline tools examine the state of the machine while it is paused. These tools do NOT depend on the OS being correct (unlike first case), and do not interfere with further replay
- X proxy allows replay of the screen

Runtime overhead is fine, typically 1-60% with UM Linux and would be lower with Xen

Replaying complex programs does in fact produce the same results

Final thought: a VMM is also a very effective place for an attacker to reside... install a VMM under the OS and then take over at will, but remain largely undetectable.

Advanced Topics in Computer Systems, CS262A
Prof. Eric Brewer

Lessons from Giant-Scale Services

Experience paper on how to build and operate very large Internet sites...

Background:

- CAP Theorem: can pick any two of Consistency, Availability, Partition-tolerance
- ... but at most two.
- Clusters pick C and A; disconnected operation and leases picks AP, locks pick CP
- Availability defined by the view at the data center -- if you can't connect that is outside the scope!

Key ideas:

- Load management
- Partitioning vs. Replication, load redirection
- Availability metrics: yield and harvest, MTTR emphasis
- Online evolution
- Graceful degradation
- the DQ principle

Basics:

symmetry

data center

backplane

Load Management

smart clients

disaster recovery

Availability Metrics: uptime, MTBF, MTTR, yield, harvest

DQ Principle

Giant Scale

Replication vs. Parititioning

- replication maintains D but not Q
- partitioning maintains Q but not D
- which is better?

Load Redirection problem: replicating the data is not enough -- must replicate the DQ access to get to the data

Graceful Degradation

- major drop in DQ
- typically try to maintain Q by reducing D significantly
- but many more sophisticated options: skip hard queries, turn on non-critical services, cache more (with stale data)

Disaster Tolerance

- loss of many replicas plus graceful degradation

Online evolution

- need a process
- staging: maintain two full versions, fast swap among them
- three ways to upgrade

Moving sites!

- possible but not easy...

Advanced Topics in Computer Systems, CS262

Prof Eric A. Brewer

Cluster-Based Network Services

I. Network Services:

- 24x7 operation
- huge scale (unprecedented)
- personalization
- no distribution problem (vs products)

Basic Advantages of Clusters:

- Absolute scale (larger systems than any single computer)
- High Availability -- but must tolerate partial failures
- Commodity building blocks => cost, service and support, delivery time, alternate suppliers, trained employees

Challenges:

- Hard to administer: single system image? ease of global view?
- Partial failure brings new problems: must tolerate failures, can't just reboot
- hard to have shared state (no shared address space)

ACID vs. BASE:

Idea: focus on HA with looser semantics rather than ACID semantics

- ACID => data unavailable rather than available but inconsistent
- BASE => data available, but could be stale, inconsistent or approximate
- Real systems use BOTH semantics
- Claim: BASE can lead to simpler systems and better performance (hard to prove)
  - Performance: caching and avoidance of communication and some locks (e.g. ACID requires strict locking and communication with replicas for every write and any reads without locks)
  - Simpler: soft-state leads to easy recovery and interchangable components
- BASE fits clusters well do to partial failure and lack of a (natural) shared namespace

TACC Model:

- Restartable Workers
  - can run anywhere (even on overflow nodes)

- Worker must handle it's own restart (easy with soft state workers, or workers that interface to an extern al database)
- Load balancing and worker creation/deletion is handled by SNS layer
- Fault tolerance = restart/migrate failed workers

o Four kinds of workers:

- Caching: stores post-transform, post-aggregation, and WAN content
- Transformation: one-way conversion of data, including format changes (eg MIME type), resolution, size, quality, color map, language, etc.
- Aggregation: combination of data from multiple sources; eg. movie info from different theaters, company info from multiple sites (analgous to a "join" for internet content)
- Customization: support for personalization/localization based on persistent profiles

o Question: is there a data independent "query" language analogous to SQL?

o Starfish fault tolerance:

- idea: any alive piece can regrow (restart) the whole system
- need to track only "aliveness" not remote state (no state mirroring, since all state is soft)
- multicast to regenerate/update state (there is no difference)
- Manager watches front ends and vice versa

Burstiness and Overflow

o Problem: peaks >> average => hard to plan capacity

o General solutions:

- caching absorbs some spikes, especially if it can be more aggressive during overload
- admission control (especially of "hard" queries)
- overflow nodes

o Burstiness is real: a side effect of humans in the loop? or just natural?

o Overflow nodes:

- Idea: exploit nodes that normally have another purpose (such as desktop machines)
- Not really tried in practice so far with few exceptions, eg. Pratt & Witney run simulations on desktops at night, but not really an "overflow"
- Similar to another real world phenomenon (apocryphal?): Schwab uses managers to answer customer calls during an overflow; they are all trained but only work during overflow

Chord

Goal: build a “peer to peer” hash table for the wide area.

- DHT work started in response to Napster, which was a centralized search engine but p2p distribution of files
- DHTs aim to make the search engine decentralized: given a key, find a node with that key/value pair and return the value
- But... nodes come and go relatively quickly (“churn”)

Consistent Hashing

- key k assigned to node equal or just following k -- its "successor"
- gives load balancing (1+\epsilon)K/N keys each, cheap join/leave O(K/N). Based on random hashing. \epsilon is O(\log N)
- if you run O(\log N) virtual nodes per real machine, you can reduce this arbitrarily

Routing: the Ring of successor pointers is key the base case

- CHORD: number of fingers equal to number of bits in the ID space. successor of (NodeID + 2^i). First finger is the successor.
- "recursive" routing in log N steps. Can also do "iterative". tradeoffs?
- What is this: arithmetic!
  - Connection to Group Theory: Cayley and Coset graphs.Deconstructing DHTs

Join (the DHT kind, not the database kind):

- correctness invariants are on data placement and successors. rest of finger table is "just" hints for performance.
  - 1) initialize new node's state (pred and fingers). based on lookups in existing ring. do bulk lookups to save cases where finger[i] and finger[i+1] are the same, makes things scale with net size, not address space size. Also, could ask neighbor for his finger table and pred to bootstrap, reducing init time to log(N)
  - 2) update fingers and pred of existing nodes to point to new node. We basically know who shold point to us: the node at n-2^(i-1) and a contiguous run of its predecessors
  - 3) move state (or ask app to do so). this is dodged. can you make this atomic in a p2p system?

Concurrent operation!

- how does this work when lots of this is happening at once?
- lookup cases: fast, slow (bad fingers), and wrong (bad successors). Can you detect "wrong"?
- stabilization: separate correctness and performance maintenance. Stabilization updates successors. Works if network remains connected. Idea: "fix forward": periodically check your successor's pred to see if you've got the wrong successor, and if so notify new successor. Note that join can now simply point to successor, does not set up anyone

else anymore -- stabilization does that!

- theorems: once linked in, always linked in. eventual consistency of successors with quiescence (no more joins).
- fixing fingers: well, notice that a single join doesn't mess up fingers much. if finger fixing goes faster than NETWORK DOUBLING IN SIZE, things remain log lookup. So fix fingers lazily, on error perhaps.

Failure:

- to deal with this, maintain r = O(log n) successors under stabilize, not just one. upon failure, skip the successor. stabilize will take care of the rest.
- A challenge: network locality. Many many schemes proposed for this, some quite fancy. Basic idea is to choose fingers with some randomness, and maintain multiple alternatives via measurement (where distance typically equals latency).

Advanced Topics in Computer Systems, CS262B

Prof Eric A. Brewer

Distributed Data Structures

Gribble et al.

Goal: new persistent storage layer that is a better fit for Internet services

- separate service logic from durability, availability, consistency issues
- interface is that of a data structure, not a SQL query (intentionally navigational)
- automate replication and partitioning across the cluster
- automate recovery and high availability
- CAP: CA (not P) -- depends on a machine room with a redundant network

Nit: abstract's perf numbers have too much precision! (should have been limited to 2-3 significant digits)

Alternatives for persistent storage:

- DBMS: chooses C over A, interface is inherently slow: parse and plan each query.
- Enterprise Java Beans (EJB): map Java objects onto tables. Typical use has an array of objects map to a table with one row for each object; row stores the serialized bytes for that object => expensive copy to get the object in and out of the database. Also, storage is never local and there is little control over partitioning and replication (which nodes). Should be 10x or more slower, but no reliable data.
- File system: also has an expensive interface and very hard to provide fine-grain atomicity. (primary atomic operation is file rename, which implies copying the whole file). Fine choice for large objects, but not for small ones...

Internet requirements:

- extreme scale: billion request/day
- high availability
- unknown but potential fast growth -- must be able to add capacity quickly
- overload will occur -- need graceful degradation

Cluster properties:

- Incremental scalability -- add nodes over time
- Potential for high availability -- but needs to be written!
- Natural parallelism for both I/O and CPUs
- High BW, low-latency, *partition free* network (CAP)
- Machine room properties: security, administration, reliable power, AC, networking (these generally don't apply to P2P systems in practice)

DDS design decisions:

- hash table interface: 64-bit keys, byte array values, put/get/delete elements, create/destroy hash tables

- operations are atomic, sequences of operations (trans-actions) are not
- two-phase commit across all replicas for consistency of writes
- read any replica (=> higher read throughput than write throughput)
- event-driven design for high concurrency. this was a mistake and led to Capriccio. It is being rebuilt from scratch in C over Capriccio. (see Java problems below)
- assume no network partitions (CAP)

Partitions:

- key idea: break hash tables into many small fixed size "partitions"
- replicate partitions (not tables); replica groups are a set of partitions with the same data
- recovery: recover each partition independently
- small partitions mean that you can lock the whole partition during recovery (other partitions operate independently)
- also implies that you can lazily recover each partition, or you can be proactive.

Fault tolerance

- DDS library: handles two-phase commit across replica group.
  - commit occurs when at least one replica receives a commit message
  - failure before this and all replicas will time out and discover there was no commit by contacting each other
  - failure after this and the replicas will learn about the commit from the member that recorded it, and will then all commit
- Brick:
  - failure before commit will cause 2PC to fail and everyone aborts
  - failure after agreement to commit, but before commit: other replica's commit, this node will get the new value during recovery
- Other service code:
  - if all durable state is in the DDS, this is recovered automatically on restart
  - soft state can be rebuilt (e.g. caches)
  - if local durable state (not in DDS), then service is on its own (e.g. might store some data in a DBMS also)

Data Partitioning Map:

- maps HT key to partition id
- uses a trie so that it is easy to split or merge partitions as needed (increment data repartitioning!)
- replicated on all DDS clients, eventual consistency (stale data causes a repair and a retry); staleness is detected by comparing a hash of the maps to the current values.

Replica Group Map:

- maps partition id to the set of replicas
- replicated on all DDS clients, eventual consistency as above

- writes are 2PC to all replicas
- reads go to one replica. which one? want to spread out reads in the same way to maximize effective cache space!

Recovery:

- Node failure takes down all bricks on that node; brick failure takes down all partitions on that brick
- a recovering brick must catch up all its partitions
- key idea 1: allow a brick to say "no" -- this simplifies the code greatly! DDS library will retry in a bit. "NO" means that brick can be inconsistent for a while (but not too long)
- all committed operations must use up-to-date maps (DP and RG), else retry
- bricks catch up one partition at a time: just copy that partition from another replica.
- Updates to a partition stop during recovery, but this is OK but partitions are small!
- Must decide how proactive to be in recovery; OK to be completely lazy (wait for partition to be accessed, but hurts latency)
- see recovery graph (figure 8)

Performance:

- great scalability, availability, graceful degradation for reads (less so for writes)
- Problems: Java GC can screw up performance! One a node gets behind it needs more memory (for growing queues), so it tends to get worse and worse! Need admisstion control, control over GC, or probably both. SEDA would probably fix this...
- Events make this much worse! Event driven systems (in Java) create a lot of garbage, as events are just passed up through layer and then GC'd. Threads would have most of this state on the stack and thus create very little gargage... (this would also be a bad place for functional languages, which need to copy a great deal)
- serious problems with Java: GC issues, extra copying, asynchronous I/O... We later fixed async I/O, but the other two issues remain. In general, Java is too high level for this kind of work...
-

Dynamo

I. Background

Highly available key-value store (hash table)

- S3 is a highly available file system
- EC2 is a highly available VM hosting service

Goals:

- HA despite high load, many faults
- Internal service => non-malicious clients (other Amazon services only)
  - each client service runs its own dynamo instances
- enable tradeoffs among consistency, availability, cost and performance
- simple primary key operations only
- no multi-key atomic operations; no isolation except single key updates
- incremental scalability & heterogeneity
- minimal management required
- eventual consistency
- replace DBMS for many needs
  - expensive to buy; expensive to operate/manage
  - tends to pick C over A
- 99.9 percentile metrics

Techniques:

- consistent hashing to avoid heavy repartition with varying participants
- versioning for consistency
- quorom-like consistency protocol
- decentralized replica synchronization
- gossip-based failure detection

Design issues

- choose A, P, and then work on conflict resolution
- when to resolve conflicts? on reads or on writes? (dynamo picks reads; writes always work)

Dynamo

II. Architecture

- get(key) -> {list of conflicting versions}, context
  - key is array of bytes
  - Internal keys are 182-bit MD5 hash of the key
- put(key, context, object)
  - object is array of bytes

Context is a complex version number, covered below.

Partition key among nodes via consistent hashing:

- oversample by some factor to get more even distribution; ie. each node supports 16 virtual nodes, so variation in load reduced by sqrt(16) = 4
- virtual nodes can be spread unevenly to support heterogeneity of nodes

N-way replication for each key-value pair

- n-1 successors also store a replica
- ... but would like other nodes to be able to know all n nodes, not just the first one (in case that one is down); more on this later
- but want independent failure, so use first n-1 *distinct* nodes (since one real node may support multiple virtual nodes among the the n)

Versioning

- choose A, so updates work all the time
- given a partition, the two sides may have different versions
- eventually both visible (and both returned by get()); **this is exposed in the API. This is the right call as there is no generic way to hide inconsistent versions well.**
- add/delete are both updates and both increment version number
- Vector clocks:
  - list of (node, counter) pairs
  - given two VCs, v1 and v2, v1 is newer if it has a superset (or the same set) of nodes and for every node in common, the counter value for v1 is >= that of v2
  - usually only one node entry, but expands in the presence of faults/partitions
  - scalability limited by # of distinct nodes

Operation:

- use load balancer or smart client approach to find server node
- typically start at first of N replicas, called coordinator; if guess was wrong (ie. not a top N node), then forward to correct node
-
- put (write): need W replicas to participate; R+W > N
  - generate new vector clock (given context and node)
  - write new version locally

Dynamo

- send object and VC to N highest ranked reachable nodes
- wait for W-1responses
- return to client

o get (read): need R replicas to participate

- request versions from N best successors
- wait for R reponses
- find current versions (remove duplicates, dominated VCs)
- return current versions (client may do conflict resolution)

Hinted handoff:

- o may need to send to non-optimal node due to failures/partitions
- o once partition fixed we can forward local versions to their correct home

N vs R vs. W

Replica synchronization:

- o Merkle trees: hierachical hash function; leaves are the keys and a parent node is the hash of its children
- o To compare replicas, just compare their trees top down. If root is the same, then whole tree is the same. Walk down the path that doesn't match to find the mismatched keys
- o one tree for each virtual node
- o churn => some changes in virtual nodes and therefore a need to recalculate trees sometimes

Explicit join/leave of nodes

- o other "leaves" are temporary, likely due to faults
- o having explicit join/leave differentiates between temporary and permanent. The latter implies repartitioning for example.

CS262 Lecture Notes
Eric Brewer

BigTable

Goal: a general-purpose data-center storage system

- big or little objects
- ordered keys with scans
- notion of locality
- very large scale
- durable and highly available
- hugely successful within Google -- very broadly used

Data model: a big sparse table

- rows are sort order
  - atomic operations on single rows
  - scan rows in order
  - locality by rows first
- columns: properties of the row
  - variable schema: easily create new columns
  - column families: groups of columns
    - for access control (e.g. private data)
    - for locality (read these columns together, with nothing else)
    - harder to create new families
- multiple entries per cell using timestamps
  - enables multi-version concurrency control across rows

Basic implementation:

- writes go to log then to in-memory table “memtable” (key, value)
- periodically: move in memory table to disk => SSTable(s)
  - “minor compaction”
    - frees up memory
    - reduces recovery time (less log to scan)
  - SSTable = immutable ordered subset of table: range of keys and subset of their columns
    - one locality group per SSTable (for columns)
  - tablet = all of the SSTables for one key range + the memtable
    - tablets get split when they get too big
    - SSTables can be shared after the split (immutable)
  - some values may be stale (due to new writes to those keys)
- reads: maintain in-memory map of keys to {SSTables, memtable}
  - current version is in exactly one SSTable or memtable
  - reading based on timestamp requires multiple reads
  - may also have to read many SSTables to get all of the columns
- scan = merge-sort like merge of SSTables in order

        - easy since they are in sorted order
    - Compaction
        - SSTables similar to segments in LFS
        - need to “clean” old SSTables to reclaim space
            - also to *actually* delete private data
        - Clean by merging multiple SSTables into one new one
            - “major compaction” => merge all tables

Bloom Filters

- goal: efficient test for set membership: member(key) -> true/false
- false => definitely not in the set, no need for lookup
- true => probably is in the set
    - so do lookup to make sure and get the value
- generally supports adding elements, but not removing them
    - but some tricks to fix this (counting)
    - or just create a new set once in a while
- basic version:
    - m bit positions
    - k hash functions
    - for insert: compute k bit locations, set them to 1
    - for lookup: compute k bit locations
        - all = 1 => return true (may be wrong)
        - any = 0 => return false

    - 1% error rate ~ 10 bits/element
        - good to have some a priori idea of the target set size
- use in BigTable
    - avoid reading all SSTables for elements that are not present (at least mostly avoid it)
    - saves many seeks

Three pieces to the implementation:

- client library with the API (like DDS)
- tablet servers that serve parts of several tables
- master that tracks tables and tablet servers
    - assigns tablets to tablet servers
    - merges tablets
    - tracks active servers and learns about splits
    - clients only deal with master to create/delete tables and column family changes
    - clients get data directly from servers

All tables part of one big system

- root table points to metadata tables
    - never splits => always three levels of tablets
- these point to user tables

Tricky bits:

- SSTables work in 64k blocks
    - pro: caching a block avoid seeks for reads with locality
    - con: small random reads have high overhead and waste memory
        - solutions?
- Compression: compress 64k blocks
    - big enough for some gain
    - encoding based on many blocks => better than gzip
    - second compression within a block
- Each server handles many tablets
    - merges logs into one giant log
        - pro: fast and sequential
        - con: complex recovery
            - recover tablets independently, but their logs are mixed...
                - solution in paper: sort the log first, then recover...
            - long time source of bugs
    - Could we keep the logs separate?
- Strong need for monitoring tools
    - detailed RPC trace of specific requests
    - active monitoring of all servers

Brewer/Hellerstein CS262 Spring 2008: 2PC and Paxos

- A theme: two-phase protocols
  - Courtesy Jim Gray:
    - Marriage Ceremony: "Do you?" "I do!" "I now pronounce you..."
    - Theater: "Ready on the set?" "Ready!" "Action!"
    - Contract Law: Offer. Signature. Deal/lawsuit.
  - Actually these protocols are pretty simple
    - Fussy to prove they're safe/correct
    - Even fussier to tune them and maintain proofs, and that's where much of the sweat goes.
- Two Phase Commit and Logging in R*
  - Setup
    - Roles
      - coordinator (transaction manager or TM)
      - subordinate (resource manager, or RM)
    - Goal: All or nothing agreement on commit (single subordinate veto is enough to abort).
      - Also, integrate properly with log processing and recovery.
    - Assumptions
      - Update in place, WAL
      - batch-force log records
    - Desired characteristics
      - guaranteed xact atomicity
      - ability to "forget" outcome of commit ASAP
      - minimal log writes and message traffic
      - optimized performance in no-failure case (the "fast path")
      - exploitation of completely or partially R/O xacts
      - maximize ability to perform unilateral abort
    - In order to minimize logging and comm:
      - rare failures do not deserve extra overhead in normal processing
      - Hierarchical commit better than 2P
  - The basic 2PC protocol with logging (normal processing):

Coordinator Log	Messages	Subordinate Log
	PREPARE	
		prepare*/abort*
	VOTE Y/N	
commit*/abort*		
	C/A	
		commit*/abort*
	ACK	
end		

    - Rule: never need to ask something that you used to know! Log before ACKing.
      - Since subords force abort/commit before ACKing, they never need to ask coord to remind them about final outcome.
    - Costs:
      - subords: 2 forced log-writes, 2 msgs
      - coord: 1 forced log write, 1 async log write, 2 msgs per subord
      - total: 4n messages, 2N+1 log writes. Delays: 4 message delays, 3 sync writes.
        - we'll tune this down below
  - 2PC and failures
    - Note: 2PC systems are *not available* during a coordinator failure! Yuck!! (See Paxos Commit, below, for discussion)
      - what about subordinate failure?
    - Recovery process protocol:
      - 1 On restart, read log and accumulate committing xacts info in main mem
      - 2 if you discover a local xact in the prepared state, contact coord to find out fate
      - 3 if you discover a local xact that was not prepared, UNDO it, write abort record, forget
      - 4 if a local xact was committing (i.e. this is the coord), then send out COMMIT msgs to subords that haven't ACKed Similar for aborting.
    - Upon discovering a failure elsewhere

Brewer/Hellerstein CS262 Spring 2008: 2PC and Paxos

        - If a coord discovers that a subord is unreachable...
            - while waiting for its vote: coord aborts xact as usual
            - while waiting for an ACK: coord gives xact to recovery mgr
        - If subord discovers that coord is unreachable...
            - if it hasn't sent a YES vote yet, do unilateral abort
            - if it has sent a YES vote subord gives xact to recovery mgr
        - If a recovery mgr receives an inquiry from a subord in prepared state
            - if main mem info says xact is committing or aborting, send COMMIT/ABORT
            - if main mem info says nothing...?
    - An aside: Hierarchical 2PC
        - If you have a tree-shaped process graph
            - root (which talks to user) is a coord
            - leaves are subords
            - interior nodes are both
                - after receiving PREPARE, propagate to children.
                - vote after children. any NO below causes a NO vote (this is like stratified aggregation!)
                - after receiving COMMIT record, force-write log, ACK to parent, and propagate to children. similar for ABORT.
    - Tuning approach I: Presumed Abort
        - recall... if main-mem says nothing, coord says ABORT
        - SO... coord can forget a xact immediately after deciding to abort it! (write abort record, THEN forget)
        - abort can be async write
            - no ACKS required from subords on ABORT
            - no need to remember names of subords in abort record, nor write end record after abort
            - if coord sees subord has failed, need not pass xact to recovery system; can just ABORT.
        - Look at R/O xacts:
            - subords who have only read send READ VOTEs instead of YES VOTEs, release locks, write no log records
                - logic is: READ & YES = YES, READ & NO = NO, READ & READ = READ
                - if all votes are READ, there's no second phase
                - commit record at coord includes only YES sites
                - Tallying up the R/O work: N+1 msgs, no disk writes. Delays: 1 msg delay.
    - Tuning approach II: Presumed Commit
        - Should be the fast path, can we do it fast?
        - Inverting the logic:
            - require ACK for ABORT, not COMMIT!
            - subords force abort* record, not commit
            - no info? presume commit!
        - Problem!
            - subord prepares
            - coord crashes
            - on restart, coord aborts and forgets
            - subord asks about the xact, coord says "no info = commit!"
            - subord commits, but everybody else does not.
        - Solution:
            - coord records names of subords on stable storage before allowing them to prepare ("collecting" record)
            - then it can tell them about aborts on restart
            - everything else analogous (mirror) to P.A.
            - Tallying up R/O work: N+1 msgs, 2 diskwrites (collecting*, commit), Delays: 1 diskwrite delay, 1 msg delay.
    - Costs of the variants
        - 2PC commit: 2N+2 writes, 4N messages. Delays: 3 write delays, 4 msg delays
        - PA commit: 2N+2 writes, 4N messages. Delays: 3 write delays, 4 msg delays
        - PC commit: 2N+2 writes, 3N messages. Delays: 3 write delays, 3 msg delays.
        - PA *always* beats plain 2PC
        - PA beats PC for R/O transactions
        - for xacts with only one writer subord, PC beats PA (PA has an extra ACK from subord)
        - for n-1 writer subords, PC much better than PA (PA forces n-1 times at subords on commits, sends n extra msgs)

Brewer/Hellerstein CS262 Spring 2008: 2PC and Paxos

- choice between PA and PC could be made on a xact-by-xact basis!
  - "query" optimization? Overlog?

- Paxos
  - Setup
    - 3 roles being played
      - A single Proposer ("Leader"), proposes "values"
        - Leader-election protocol is well-known and predates this work
      - Acceptor, part of protocol to decide on "choosing" values
      - Learner, hears about "chosen" values
    - Goal: majority agreement to "choose" a proposed value
      - Imagine a single Consensus Box. Now emulate that with a distributed set of machines that can tolerate failure.
      - Non-triviality: only proposed values can be learned
      - "Consistency": 2 learners cannot learn different values
      - Liveness: if value C has been proposed, and enough processes are alive, eventually each learner will learn some value
    - Assumptions
      - Async machines
      - Independent, fail-stop failures
        - will tolerate F/(2F+1) nodes failing *simultaneously*.
        - vs. 2PC. vs. Byzantine Agreement.
      - msgs lost, delayed, reordered, but not corrupted.
  - The basic Paxos protocol
    - Proposer | Acceptors | Learner
    - prepare(n) → |
    - |← promise (m,w) |
    - Accept(n,v) → |
    - |← accepted → |
    - | | broadcast →
    - notes:
      - acceptors only promise(m,w) if m < n and they haven't promised something higher than n already
        - w is the last value *accepted* (or null)
      - proposer only issues accepts if a majority promised. if all acceptor returned null w's, proposed gets to choose v (the *free* case). else v is the w it received with the highest associated m (the *forced* case).
        - why should a proposer bother accepting if it is forced by a non-null w?
    - Costs
      - 4F messages, 4 message delays.
  - Paxos with failures
    - Acceptor failures
      - First, note that all majorities overlap by 1
        - Whenever a majority of acceptors is non-failed in future, previously accepted values will be stored with associated numbers.
      - Second, note how promises help
    - Learner failures
      - trivial
    - Proposer failures
      - Leader-election will replace proposer on failure
      - Proposer can fail any time before accept with no confusion
      - Fail after Accept msg sent out causes trouble: dueling proposers
        - new leader will be elected, and if old leader recovers she won't know she's no longer leader
        - prepare(n) will fail
        - new leader may try to restart with prepare(n+1)
          - gets promises
        - old leader recovers and tries to restart with prepare(n+1)
          - gets NACKs
        - old leader tries prepare(n+2)
          - gets promises
        - new leader tries to accept(n+1)

Brewer/Hellerstein CS262 Spring 2008: 2PC and Paxos

        - gets NACKs
      - etc,.
      - Leader-election will eventually solve this
  - Many variants -- see Wikipedia entry
    - Multi-Paxos: for continuous stream of consensus tasks. Skips Phase 1.
      - Very typical implementation
      - (Actually, we can always skip Phase 1, even without multi)
    - Cheap Paxos: let F of the 2F+1 machines be slow
    - Fast Paxos: skip phase 1, let clients initiate phase 2 via broadcast to proposer and acceptors
    - Byzantine Paxos: allows for nodes to be malicious.
  - Paxos and distributed state machines
    - A nice model (the usual model!) for reasoning about fault-tolerant systems is the distributed state machine
      - multiple clients
      - server implemented by multiple nodes running redundant copies of the same deterministic state machine
      - how do we ensure that each machine runs the same commands in the same order?
        - a Paxos leader (proposer) serializes all client requests.
        - it uses Paxos to get consensus on the content of the n'th request
      - if leader fails, leader election picks a new one. recovery works out pretty well:
        - even if we have dueling leaders!
          - Phase 1 of Paxos is used to get one of the leaders to "win" the nth Paxos round
          - Only in Phase 2 does that leader actually issue the command.
            - the command for for round *n* is only chosen after Phase 2 for round *n-1* completes
            - hence to choose a command, you have to be all caught up on history, and hence choose the "right" one.
        - how does a new leader "catch up"
          - well, it had been a listener, so it has a partial view of history
          - start by issuing Phase 1 requests for any gaps in history, and *all "future" rounds* (expained below)
            - will learn the history from the Promise responses
          - run Phase 2 for all the promises that responded with a value
            - at minimum local execution of the commands
            - to complete the sequence of historical commands, replace any remaining gap commands with no-op proposals
          - what does it mean to do phase one for all future rounds (infinitely many)?
            - propose a *single sequence number* in one message, representing an unbounded number of rounds
              - acceptor can simply say OK
- Paxos Commit
  - Gray & Lamport 2006!! (from a 2004 TR)
  - History: Skeen's Non-Blocking (3-Phase) Commit
    - Handle the case of a failed transaction coordinator
      - multiple coordinators and failover
      - nobody every nailed this down (specific algorithm with correctness proofs)
  - Paxos makes this really simple
    - we can have multiple coordinators (transaction managers), and their decisions on commit are handled by Paxos
      - client issues "prepare" to multiple coordinators
      - subordinates respond "prepared" to all coordinators
      - Paxos used to deal with coordinator decisions if any of the coords fail.
        - Note -- still unanimous decision by subordinates! Majority used at coordinators.
      - Same logging all around
      - A version of this due to Mohan in 1983 (with a slower consensus protocol)
      - Paxos Commit also includes an optimization over the Mohan solution
        - coordinator need not be the Paxos proposer!
        - subordinates don't respond to coordinator prepare. instead, they serve as Paxos proposers for their own status
        - coordinators are Listeners on those proposals, and can issue commits upon getting a majority for *each* subordinate
        - saves one round of messages
        - Acceptors in Paxos must log each accepted message before sending it.
    - Total cost (with all optimizations): (N-1)(2F+3) msgs, N+F+1 writes. 4 message delays, 2 write delays.
  - Full paper is (typically) complex and full of fussy detail

K42

I. Background

Several assumptions in 1996:

- *Windows would dominate everything except very high end (so focus there)*
  Not true; shifted their strategy to promote/leverage Linux in 2000
  Decided not to support multiple personalities => less need for customizability
- *Multiprocessors would become ubiquitous; especially NUMA*
  Manycore is here, but arrived slower than expected
  NUMA not true yet: hardware trying to make systems mostly uniform; more clusters of UMA than NUMA shared memory
- *Maintenance/development cost would dominate*
  Generally true: still not very modular, which hinders development. Places that are modular, kernel-loadable modules, seem to have more innovation
- *Will need to be customizable/extensible*
  Has been useful, but complex to implement; VMs change the picture some as do the ability of clusters to handle the availability issues (so you can take down a node easily). IBM developed its own hypervisor for fault containment and to co-exist as a guest OS
- *All machines moving to 64-bit*
  Still coming, but really only at the high end so far. K42 spent a huge amount of time creating the 64-bit open source community and it still limits them some

Basics:

- Aimed for small kernel, with much functionality in user-level libraries
  - enables customization/extensibility
  - enable multiple “personalities” over the same core (but now less needed with rise of Linux, VMs
- Extensive use of OOP
- Aim for scalability to many cores/CPUs with shared memory
  - avoid global locks!
- Multiplex multiple OSs in time -- seems like a bad idea compared to VMs

II. Scalability

Two broad approaches:

- fine-grain locking (not generally true for other OSs)
- memory locality

- some per-CPU memory

Approaches:

- protected procedure call (PPC):
  - cross-address space (client to server)
  - both sides run on the same processor (for memory locality)
  - each client request spawns a server thread (EB: might want to limit this with a thread pool); client thread blocks
- Locality aware memory allocation (think free list for each processor)
- Use of local, fine-grain objects to ensure fine-grain locking (per object); also enables customizability
- Cluster objects (covered below)
- In general, don't block in the kernel (like capriccio)

Memory techniques:

- Partition state among CPUs; enables scalability and locality
- Push page faults up to app (app blocks, but OS is event driven)
- Processor specific memory (used for clustered objects among other things)

Clustered Objects:

- Basic idea: a set of objects, one per processor, that work together to implement a service
- Mechanism: indirect call using COIDs (clustered object IDs) -- each CPU has a COID to function pointer table to find the local object
- Objects could be different, generally different instances of the same class; must at least have the same interface
- 0<= # objects <= #CPUs
  - 0 because objects can be created lazily; invoking the object causes it to be created
- Local object called the "rep"
- Easy part: scales well, has fine-grained locking
- Hard part: clustered objects must manage shared state among themselves
  - reps have pointer to "root" object that manages single-copy shared state
- Nice point: can vary the # of objects over time based on load

III. User-Kernel interface

Scheduling:

K42

- kernel schedule address spaces
- user-level schedules threads
- process = address space + one or more dispatchers
- multiple dispatchers for multiple cores or for different priorities/QoS
- threads can block for page faults (for example), but dispatcher retains control of the core
  - page fault => dispatcher receives upcall
  - halts offending thread
  - runs something else
- similarly, systems calls can block the thread without blocking the dispatcher
- priorities first, then lottery within one priority
- Posix can be on top of dispatchers

Message passing

- both sync and async messages between cores
- server process can export an object, and clients can call its methods via messages
- async calls have no reply and don't block the caller
- soft interrupt can be used to notify other dispatchers in the same address space (process)
-

Speculative Execution

I. Background

Long tradition of research on speculation for computer architecture:

- branch prediction
- value prediction (!)

But these speculations have limited upside... what about speculation at the application level?

Insight: OS controls side effects, so OS can limit the impact of speculation!

- E.g., delay printf until the computation is really going to happen for sure
- If cancelled, we just delete it

Start with the file system: it is often synchronous for actions that work as expected such as cache hits

- we can speculate that we will hit, and then recover if we are wrong
- synchronization delays are the cost of consistency
- distributed file systems (DFS) have to choose between these delays and weaker consistency: they generally weaken consistency

NFS provides "close-to-open" consistency (weaker):

- when you open a file you will see all changes from others that have closed that file
- plus a 30-second delay!
- do not see changes until the file is closed, which reduces the communication overhead

Strict synchronization is pessimistic:

- most file operations do not conflict
- Speculator is optimistic

Basic operation:

- when the OS reaches a blocking operation
- 1) checkpoint state (to enable recovery)
- 2) speculate on the answer
- 3) continue execution immediately using this answer
- 4) when real answer comes in, either continue along, or abort and restart from the checkpoint

Speculator

II. Speculator

Key invariants:

- Know when a process is speculative
- Speculative processes cannot externalize output (just queue the output)
- Track speculative operations as they affect other processes! (they become speculative too)
  - fork, exit, signals, pepes, fifos, sockets, local file, distributed files
  - other forms of IPC just block until speculation is resolved
- No modification of applications!

Performance: 2x over local networks, 10x over wide-area networks

NFS normal behavior:

- async writes of the data to the server
- sync commit RPC (=> round trip plus disk write)
- speculator guesses that all writes succeed and so does the commit
- on open, speculator guesses that cached copy is still valid

Three characteristics:

- ***Speculation is usually right:*** for DFS, only wrong in the case of conflicts, which are rare
- ***checkpointing is faster than remote I/O***
  - copy-on-write fork operation is fast; most pages not copied
  - child only executes if speculation was wrong.
  - take and discard a checkpoint for a small process = 52us !
  - big process = 6ms
- ***Spare CPU for speculation*** (low opportunity cost)

III. Implementation

Linux 2.4.21, 7500 lines of C

- create_speculation -> spec_id
- commit/fail calls

Creating a speculation

- copy-on-write fork (save the child for abort)
- save the state of open file descriptors, copy pending signals (to replay them)
- child is not runnable

Speculator

- simply discard the child (reclaim pages) if speculation commits
- aborted processes are killed next time they try to run
- child process *assumes the identity* of the failed process
  - process id, thread group id
  - restore file desciptors and signals
  - reexcute system call that caused speculation (would it speculate again?)

Two new data structures in the kernel:

- 1) track set of kernel objects that depend on speculation (so you can reclaim them)
- 2) an undo log for each kernel object
- On a nested speculation, you can reuse the undo log (but may undo more work than necessary)
  - if more than 500ms has passed create a new undo log, so that you won't undo too much work
  - if the first operation has side effects (like mkdir), then better to create a second log. If the first operation succeeds its side effects are visible, so we don't want to undo them!

Single process correctness:

- Invariant: speculative state never visible to external device or user
  - screen output, network packets, IPC, etc.
- Invariant: a process should never see speculative state unless it is explicitly dependent on that state (or its producer).
  - If it does see the state, and that speculation fails, then this process must fail too
  - Easiest solution is to block new process until speculation resolves (but this limits parallelism)
  - Basic version: make a special syscall vector table that blocks all system calls that would externalize output
  - V2: allows syscalls that are read only
  - V3: allows syscalls that modify only private state to this process (example: dup2)
  - V4: allows syscalls to speculative file systems
  - V5: allows read/write to speculative files
  - V6: buffer write call output until speculation resolves (to screen or network)

Multi-process speculation:

- DFS: failure => invalidate cached copy
- Ramdisk: rmdir => keep around old version so you can put it back
- ext3: never write speculative data to disk (no steal!). On failure, can just delete this buffer page rather than write it out (no undo log for the disk)
- problem with no steal: some pages may never get written back (e.g. superblock)
  - solution: use redo/undo functions and keep two version (spec and nonspec). update non-spec on commits and only write it out
- reorder compound ext3 transactions so that speculative actions are postponed

Speculator

- pipes: need to track create/delete as well as read/write
- signals: can't checkpoint the receiver at an arbitrary point (why?)
  - instead: cause receiver to checkpoint soon
  - deliver a "pending" signal, but take no action
  - when process returns from the kernel, the pending signals become real and are handled
  - checkpoint is taken a this time (which is safe)
- exit: keep around pid until process exit is confirmed

IV. Using speculation

General case:

- speculate on something in cache (if not in cache, then don't speculate)
- convert sync call to verify cache entry into async call
- start speculation
- on reply to async call, decide commit/fail

Harder for mutating operations!

- side effects of speculation could become visible to others
- key solution: server need not speculate! It knows if the hypothesis is wrong and can act accordingly
- for each RPC, include list of hypotheses for server to validate

Group commit:

- speculation leads to many outstanding "synchronous" operations
- can do one write for the group, rather than one write each!
- just delay a bit and the client will send you more work, which you can amortize with one actual write
  - great for journals, does it matter for traditional file systems? (which have writes to different parts of the disk?)

V. Evaluation

Very FAST!

Rollback cost is low

Group commit helps a great deal for journaled file system (and some for NFS)