Review session's from 7:30-9:30 tonight, in 34-101. Notes are gonna be here. You know the deal. Final exam Thursday: two hours. Designed to take 90 minutes. Always to your benefit to guess

RAID

Previous Q2 problems relating to RAID: 2011 P2, 2010 P2, 2009 P3, 2007 P3*

- CPU performance was scaling really quickly, but disk performance wasn't scaling as quickly (RPM).
- Goal
  - [ ] keep doing work rather than being stalled
- Instead of buying more expensive disks, use RAID
  - Link together cheap disks to increase throughput
    - But with lots of cheap disks, there are more frequent failures - RAID solves that

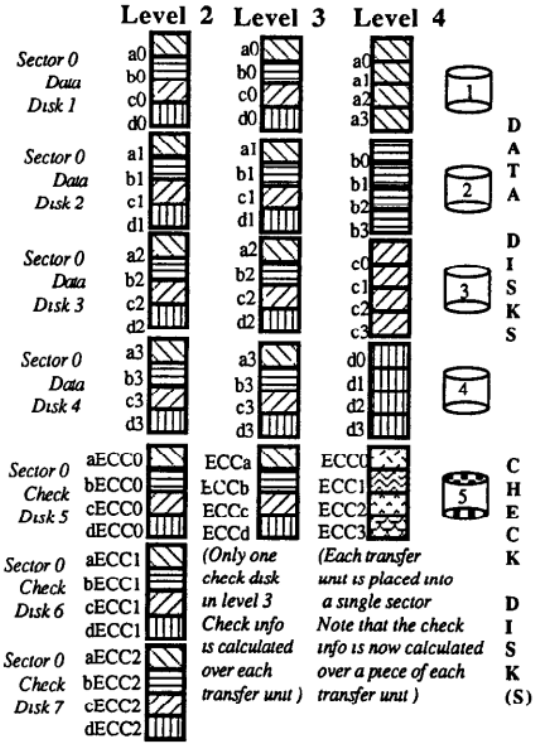


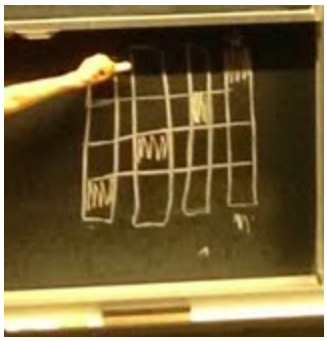
Figure 3 *Comparison of location of data and check information in sectors for RAID levels 2, 3, and 4 for G=4 Not shown is the small amount of check information per sector added by the disk controller to detect and correct soft errors within a sector Remember that we use physical sector numbers and hardware control to explain these ideas but RAID can be implemented by software using logical sectors and disks*

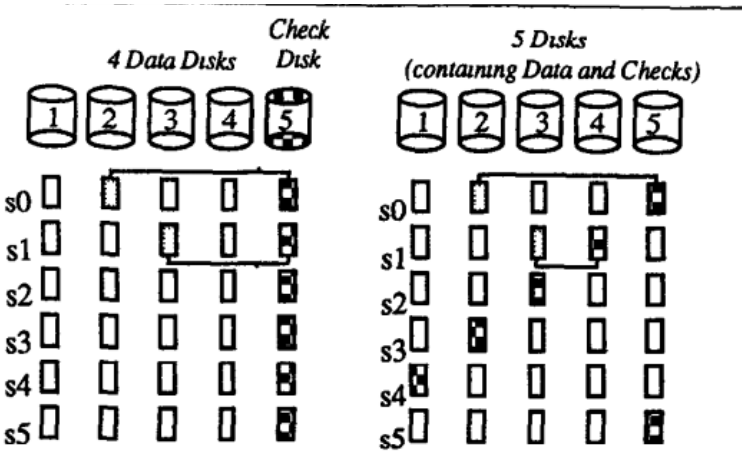
- Five different RAID levels
  - Only 1 and 5 are used in practice
    - Level 1-3 (Disk storage)
    - Level 4-5 (performance)
  - Raid level 1:

        - keep multiple copies of data;
        - every time you want to do a read, it is very quick.
        - Writes are slow because you have to write to both disks.
        - The storage overhead is also very high, as you have to explicitly write both copies.
    - Level 2 - bit-level ECC (error-correcting code)
        - Pro: Fewer disks dedicated to backup and it's possible to correct errors.
        - Con: In practice using the recovery mechanism isn't practical.
        - read all the sectors in all the disks

data disk	data disk	data disk	redundant disk	redundant disk

    - Level 3
        - Instead of multiple redundancy disks, get rid of all but one. Make that disk a parity disk
        - Errors are detected using an in-built disk checking mechanism
        - Wins the disk count/storage tradeoff
    - Level 4
        - Stripe data blocks/sectors across disks one sector at a time (i.e., consecutive bits of same transfer unit are stored on the same disk)
        - Instead of calculating parity one bit at a time, does a sector at a time
        - XOR first three sectors together, bitwise, to get your parity sector
        - "Writes are really bad because if you're doing a small write, you get terrible performance because every time you do a write, you have to do a change to the redundancy."
        - **Performance is bad because the redundancy disk is congested - bottleneck**
    - Level 5:
        - Instead of consistently using 1 disk for redundancy, we have four disks and four sectors that contain redundancy information
            - Which disk has redundancy info changes on a sector-by-sector basis





Raid 4 Raid 5

- This way, you can write to multiple disks and check disks in parallel
  - check disks were bottleneck in previous RAID levels
- RAID 5 is much better than RAID 1! Only has one redundancy disk.

Interleaving → vv Redundancy	none	bit	sector
replication	RAID 1		
ECC		RAID 2	
Parity		RAID 3	RAID 4/5
none			RAID 0

- **Redundancy**
  - RAID implies redundancy

Problems about RAID (quiz two from previous terms):
- 2011 problem 2
- 2010 problem 2
- 2009 problem 3
- 2007 Problem III

System R - logging and isolation

- One of the oldest relational database systems - early implementation SQL
- Showed benefit of transaction processing
- Goals:3
- Transactions: Operations you can perform on your data
    - Example: Banking
    - Multiple, parallel transactions need *isolation*
        - One solution: serializability - everything occurs in some serial order to avoid parallel transactions
            - Enforce this via 2-phase locking: acquire ALL your locks before releasing them
                - “Growing phase” - where you acquire all the locks you need
                - “Shrinking phase” - where you release all the locks that you’ve acquired
            - What about deadlock? We can avoid that by acquiring locks in the same order. (there are other solutions too)
- Recoverability
    - Transactions must either *commit* or *abort*
    - If the system crashes between two sub-operations, none of them must go through
    - Solution 1: Shadow Files
        - writes happen to a copy of the shadow file (a page table), then at file save, the shadow pointer points to the new page table
            - only the modified pages are copied
        - old page table is deleted
        - very efficient - changes only one pointer, de-allocates two blocks - not too many write operations were needed!
        - file_restore points both current and shadow file to the previous page table, deletes the newer page table
        - file level granularity, not transaction level
    - Solution 2: Logging
        - Write-ahead logging: log before you write
        - something about undos - many is the same as one....?
        - something similar about redos
        - REDO RIGHT UNDO LEFT
    - There are a lot of System R questions. Review them.

“[RE: System R] It’s also archiving. This is related but different from probability. What if your disk crashes? What if the whole datacenter is on fire? You need a recovery mechanism; essentially, you need multiple copies of your data - you need to periodically *archive* your data. Under the carpet is efficiency. You don’t want 3x, 10x overhead.”

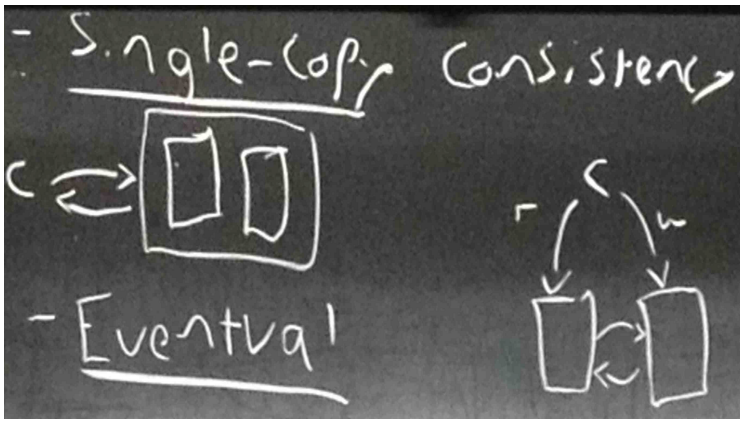
“Let me talk about transactions. They’re essentially operations that you can perform on your data; the classic example is the banking example. Let’s look at this transfer function: Transfer(A, B, $10). It begins with a “begin” command and ends with a “commit” (or “abort”) command. If you remember your hands-on on databases, you do this using two SQL commands - subtract (A=A-10) and add (B=B+10). These are the two actions in your transaction. All of these slides will be put online, so you don’t need to take pictures or copy the slides verbatim.”

“Multiple transactions need isolation. Let’s say there’s a transaction: A to B, ten dollars. I’m going to add interest of 10% to both accounts. [Interleaving requires judicious care.]”

PNUTS (PLATFORM FOR NIMBLE UNIVERSAL TABLE STORAGE)

Replicated state machines (RSM) is directly related

- Goal: build a reliable system from unreliable components
    - **consistent** replicas of data across a series of servers



- Consistency, form 1: Single-copy consistency:
    - multiple replicas of the servers
    - client is able to interact with these multiple replicas as if its one server?

- Consistency, form 2: Eventual consistency:

    - takes time to get to a consistent datathing
    - Will eventually come to consistency but might see some stale data while servers are coming to consistency
    - copies talk to one another, resolve conflicts.
        - Client will not see changes if it talks a replica that isn’t updated
        - Data can be stale
- A solution - implement a view server that can keep track of who is primary, backup, who has failed (?)

    - But what if the view server fails?
    - So what if we just distribute consensus between primary and backup? (i.e., get rid of view server) -- explained in 6.824---- >_> like always

PNUTS - all questions are from 6.824 (PNUTS has never been covered in 033 before)

- Replicated storage system, used at yahoo (called SHERPA nowadays)
- Latency is an issue - servers are very far apart (e.g., SF/VA/TX)
    - which is why PNUTS does NOT provide single-copy consistency : too slow
    - but, PNUTS also does NOT provide eventual consistency : geographical constraints
    - so what does PNUTS provide? **per-record timeline consistency.**
        - All updates to a record happen in the same order on every single replica
        - How?!?!
            - Each record has a *master region*
                - every write must come here first
            - message broker ensures that every other replica receives the updates in the same order
            - if you're not local to your master region, your writes would be super slow
            - if majority of writes come from a particular region, the master region switches
            - ALL THE UPDATES GO TO THE MASTER FIRST. The master has its own reliability guarantees.
            - you can assume that once an update is written to the message broker it has been "committed"
        - `read-any(key)` - This is **fast** because it's close to your local region. If Alice's mom calls read-any, she might see the previous version of the profile, but never anything out of order.
        - `read-critical(key, version_number)` - use if your writes depend on a certain version - **maybe fast**, depending on whether or not your region has the required version
        - `read-latest(key)` - just give the latest version of the data
        - `test-set-write(key, value, version_number)` - only writes your data if the version number at the master region == the version number you provide
            - can we use this for bank records? no because it it per record
            - atomic update to one record
            - RM rejects if version number not equal to current version

Security

- security policy: details security goals (e.g., only allow faculty to access grades)
- threat model : who are the adversaries

- guard model: how to protect data
    - authentication
        - Prove you are who you say you are
        - Username/pass
- Passwords:
    - Only users with passwords can access data
    - Compromises should be locally contained
    - Straw-Man I
        - Keep database mapping names to passwords
        - User sends you username/password,verify
        - Not good enough if cleartext
            - Adversary can see all passwords once hacks into the system
    - Straw-Man 2
        - Have a good hash function (e.g. SHA-256, AES, FNV; cf. http://www.larc.usp.br/~pbarreto/hflounge.html)
        - Keep a database mapping usernames to hash(password)
        - User sends username/password. Verify if hash(password) matches, I guess
    - “Good?”
        - Have a good hash function, h
        - Have a random, per-user salt, s
        - Map each username to (h(password + s), s)
        - Verify if h(password + s) matches
    - Use encryption

Buffer Overruns (or how C has cost people millions of dollars)

- Don’t use C because fuck you, “but you kind of need to use C or C++ sometimes”

- Real programmers use assembly anyways +1
    - Or Lisp (like PAUL I-WROTE-IN-LISP GRAHAM)
- Heap - needs to be manually allocated and cleared
    - Store dynamic memory.
- Stack -
    - }local vars, function params
    - stack frames only live as long as the function

- blah blah stack pointer/base pointer woohoo 004

push pop push pop

grows
downwards
(towards lower mem addresses) overrun!
r
e
f
f
u
b
upwards
grows
heap

Trust

Key ideas:

- difficult to know what the software you use actually does
    - natural solution is to write everything yourself! but not everyone is swarun
    - no choice but to trust software you download
- compiler
    - translates code from one language (e.g., C) to machine code
- where do compilers come from?
    - write compiler for a new language C.2 in an old language C.1
    - chicken-and-egg problem: where did the first compiler come from? written in assembly! (super minimal C compiler)
    - C.2 can now compile itself after a compiler has been written for it
    - so, can we discard the source code for the old compiler written in C.1?
        - no! because the new compiler may contain a hidden backdoor (e.g., a master password)
        - why not patch in C.3, etc? → C.2 can infect later versions of the compiler
- how can we detect trojans?
    - two different compilers may both be legit (same functionality) but have different optimizations

- so, we run another program through the two output compilers - the results must be identical, since the two compilers were expected to be functionally identical