The Transcation Concept

CS380L

1 Preliminaries

1.1 Review

- File system basics – what are conclusions?

1.2 Outline

- Motivation/goal
- Transaction concept: ACID semantics
- Logging, checkpoints
- Two-phase commit
- Two-phase locking
- Scalability
- Nested transactions
- Long-lived transactions
- Subsets of ACID

1.3 Preview

2 Motivation/goal

2.1 Motivation

- File systems have lots of data structures
  - bitmap of free blocks

  - directory
  - file header
  - indirect blocks
  - data blocks

- For performance, all must be cached!
- Ok for reads, but what about writes?

2.1.1 Modified data in memory (”cached writes”) can be lost

- Options for writing data
  - write through - write changes immediately to disk
  - problem: slow! Have to wait for each write to complete before going on.
  - Write back - delay writing modified data back to disk (for example, until replaced). Problem: can lose data on a crash

2.1.2 multiple updates

- if multiple updates needed to performe some operation, crash can occur between them!
- For example, to move a file between directories:
  1. delete file from old directory
  2. add file to new directory
- to create new file
  1. allocate space on disk for header, data
  2. write new header to disk
  3. add new file to directory
- What if there is a crash in the middle, even with write-through have a problem

2.2 Unix approach (ad-hoc)

- metadata: needed to keep file system logically consistent (directories, bitmaps, file headers, indirect blocks, etc.)
- data: user bytes

2.2.1 Metadata consistency

- For metadata, UNIX uses synchronous write through
- If multiple updates needed, does them in specific order so that if a crash occurs, run special program "fsck" that scans entire disk for internal consistency to check for "in progress" operations and then fix up anything in progress
  - Exokernel allows guest file systems to enforce order by not writing "tainted" blocks to disk
- example:
  - file created, but not yet in any directory → delete file
  - blocks allocated, but not in bitmap → update bitmap
- Challenge:
  1. need to get ad-hoc reasoning exactly right (3 exokernel rules from Ganger's earlier dissertation help)
  2. poor performance (synchronous writes)
  3. slow recovery - must scan entire disk

2.2.2 User data consistency

- what about user data?

→ write back, forced to disk every 30 seconds (or user can call "sync" to force to disk immediately)

- No guarantee blocks written to disk in any order
- can lose up to 30 seconds of work
- Still, sometimes metadata consistency is enough

- e.g. how should vi or emacs write changes to a file to disk?
- option 1:
  1. delete old file
  2. write new file
- (how vi used to work!)
- now vi does the following:
  1. write new version to temp file
  2. move old version to other temp file
  3. move new version to real file
  4. unlink old version
- If a crash, look in temp area, if any files there, send e-mail to user that there might be a problem

• But what if user wants to have multiple file operations occur as a unit?

• Example: bank transfer
  - ATM gives you $100
  - debits your account

• must be atomic

2.2.3 General's paradox

• Want to be able to reliable update state on in two different locations (possibly on two different machines)
  - e.g., move file from directory A to directory B
  - e.g., create file: update free list, directory, inode, data block
  - e.g., atomically move $100 from my account to Visa account
  - e.g., atomically move directory from file server A to file server B

• Challenge:
  - machines can crash
  - messages can be lost

• General's paradox

  - Can I use messages and retries over an unreliable network to synchronize two machines so that they are guaranteed to do same op at same time?
  - Remarkably, no. Even if all messages end up getting through

- General's paradox: two generals on separate mountains. Can only communicate via messengers; the messengers can get lost or be captured

  - Need to coordinate the attack; if they attack at different times, then they all die. If they attack at same time, they win.
  - 1 → 2: Let's attack at 9
  - 2 → 1: OK. 9 it is.
  - 1 → 2: Check. 9 it is.
  - 2 → 1: Gotcha. 9 it is.
  - ...

- Even if all messages are delivered, can't coordinate (B/c a chance that the last message doesn't get through). Can't simultaneously get two machines to aggre to do something at same time
- No solution to this - one of the few things in CS that is just impossible.
- Proof: by induction

3 Transaction concept: ACID semantics

- Solve weaker problem: 2 operations will both happen/not happen (but not necessarily happen at same time)
- Transaction concept: give one entity the power to say "yes" or "no" for all entities

  - Local transaction: one disk update (e.g., write "commit" to log) irrevokably triggers several updates
  - Distributed transaction (2 phase commit): one machine can decide for all machines; all machines agree to go along with decision

- **ACID semantics**

  - Atomic – all updates happen or none do

- Consistent – after each update, system invariants maintained
- Isolated – no one out side of transaction sees any updates until they can see them all
- Durable – once it is done it stays done

- Gray argues ACID is right software building block for reliable systems
  - Application of end-to-end principle – you need to handle this case anyhow; handle it in a clean and correct way and get the side benefit of also solving (rare) deadlocks, (less rare) programming restrictions, etc.
  - Today: Widely accepted in databases

- Are subsets ever appropriate?
  - What would "ACI" be and when might it be useful?
  - What would "ACD" be and when might it be useful?
  - Any others?
    * Satya: "Isolation-only transactions"

4 Implementation (one thread): Logging, checkpoints

- Key idea - fix problem of how you make multiple updates to disk atomically, by turning multiple updates into a single disk write!

- **PICTURE: disk, log**

- Illustrate with simple money transfer from acct x to acct y

```
Begin transaction
x = x + 1
y = y - 1
Commit
```

- Keep "write-ahead" log ("redo log") on disk of all changes in transaction

- A log is like a journal - never erased, record of everything you've done

- Once both changes are in log, write is committed

- Then can "write behind" changes to disk - if crash after commit, replay log to make sure updates get to disk.

- Sequence of steps to execute transaction
  1. write new value of x to log
  2. write new value of y to log
  3. write ”commit”
  4. write x to disk
  5. write y to disk
  6. reclaim space on log
- QUESTION: what if we crash after 1?

→ no commit, nothing on disk, so ignore changes

- what if after 2?

→ ditto

- what if after 3, before 4 or 5?

→ commit written to log, so replay those changes back to disk

- What if we crash while writing commit?

→ As with concurrency, need some primitive atomic operation, or else can't build anything else.

  – Writing a single sector on disk (with a CRC) is atomic!

- can we write x back to disk before commit?
  – Yes: keep an ”undo log” - save old values along with new value
  – If transaction doesn't commit, ”undo” change!
- QUESTION: can we do transaction with just undo log?
- Just redo log?

5 Admin

Feedback

Getting back on schedule:

- Today: transaction
- W: Advanced file systems – LFS (optional: XFS, netapp)
- Next week: distributed and replicated file systems

6 Two-phase locking

- What if two threads run same transaction at same time?
- Concurrency → use locks

```
Begin transaction
lock x, y
x = x+1
y = y-1
Unlock x, y
commit
```

- What if A grabs locks, modifies x, y, writes to log, unlocks, and right before committing, then B comes in, grabs lock, writes x, y, unlocks, does commit;
- Then A crashes before commit

→ problem. B commits values for x, y that depend on A committing

- Solution: two-phase locking
  - Phase 1: only allowed to acquire lock
  - Phase 2: All unlocks happen at commit
- Thus, B can't see any of A's changes until A commits and releases locks → provides serializability
- Also note - gives us a way to avoid deadlock
- What happens if you try to grab a lock and it is already held?
  - (or what if you wait on a lock for ¿ 1 second, or....)

→ abort transaction!

→ avoids "no-revocation" condition of deadlock

- Generalization: readers/writers locks

7 Two-phase commit

- What if we want two machines to do an atomic update?
- example: my account is at NationsBank, yours is at Wells Fargo. How to transfer $100 from you to me? (Need to guarantee that both banks agree on what happened).
- Example: file system - move a file from directory A on server a to directory B on server b
- One machine must make irrevokable decision and then reliably inform others of decision
- Abstraction - distributed transaction - two machines agree to do something or not do it, atomically (but not necessarily at exactly the same time)
- Two phase commit
  - Phase 1: Everyone gives master machine power
  - Phase 2: Master decides and tells everyone whether commit happened or not
- Phase 1: coordinator requests
  1. coordinator sends REQUEST to all participants
     - e.g. C→S1 ”delete foo from /”, C→S2 ”add foo to /”
  2. participants recv request, execute transaction locally, write VOTE_COMMIT or VOTE_ABORT to local log, and send VOTE_COMMIT or VOTE_ABORT to coordinator

Failure case	Success case
S1 decides OK, writes ”rm /foo; VOTE_COMMIT” to log, and sends VOTE_COMMIT S2 decides no space on device and writes and sends VOTE_ABORT	S1 and S2 decide OK and write updates and VOTE_COMMIT to log, send VOTE_COMMIT

- Phase 2: coordinator decides
  1. 3
     - case 1: coordinator recv VOTE_ABORT or timeout

     → coordinator write GLOBAL_ABORT to log, and send GLOBAL_ABORT to participants

- case 2: coordinator recvs VOTE_COMMIT from all participants

→ coordinator write GLOBAL_COMMIT to log, and send GLOBAL_COMMIT to participants

2. 4 participant receives decision; write GLOBAL_COMMIT or GLOBAL_ABORT to log

- What if
  - Participant crashes at 2? Wakes up, does nothing. Coordinator will timeout, abort transaction, retry
  - Coordinator crashes at 3? Wakes up,
  - Case 1: no GLOBAL_* in log → Send message to participants "abort"
  - Case 2: GLOBAL_ABORT in log → send message to participants "abort"
  - Case 3: GLOBAL_COMMIT in log → send message to participants "commit"
  - Participant crashes at 4? → On recovery, ask coordinator what happened and commit or abort
- This is another example of the idea of a basic atomic operation. In this case - commit needs to "happen" at one place
- Limitation of 2PC - what if coordinator crashes during 3 and doesn't wake up? All nodes block forever
  - What if participants times out waiting in step 4 for coordinator to say what happened. It can make some progress by asking other participants
    1. if any participant has heard "GLOBAL_COMMIT/ABORT", we can safely commit/abort
    2. if any participant has said "VOTE_ABORT" or has made no vote, we can safely abort
    3. if all participants have said "VOTE_COMMIT" but none have heard "GLOBAL_*", can we commit? A: no - coordinator might have written "GLOBAL_ABORT" to its disk (e.g., local error or timeout)
  - Turns out - 2PC always has risk of indefinite blocking
  - Solve with 3 phase commit
    * See "distributed computing" – 3PC, Paxos

  - In practice 2PC usually good enough - but be aware of the limits

- If you come to a place where you need to do something across multiple machines, don't hack
  - use 2PC (or 3PC)
  - if 2PC, identify circumstances under which indefinite blocking can occur (and decide if acceptable engineering risk)

8 Scalability

9 Nested transactions

- Issue: Interact with multiple organizations; each interaction is a "transaction" to each organization; all interactions together are a "transaction" to you
- (travel agent example)
- Proposed solution?
  - View transaction as collection of:
    * actions on unprotected objects
    * protected actions that my be undone or redone
    * real actions that may be deferred but not undone
    * nested transactions tht may be undone
  - Nested transaction returns name and parameters of compensating transaction
  - Parent includes compensating transaction in log of parent transaction
  - Invoke compensating transactions from log if parent transaction aborted
  - "Not satisfying, but better than etnirely manual procedures that are in common use today"
  - Consistent, atomic, durable, but not isolated – "others can see the uncommitted updates of nested transactions; these updates may subsequently be undone by compensation"
  - Question: how to adapt 2 phase locking to restore isolation?

10 Long-lived transactions

11 Subsets of ACID