

Lecture#22: Transactions

Review -- 1 min

Naming – which blocks belong to which files?
which files belong to which names?

Directories – regular files containing name→fileID mappings

Outline - 1 min

Transactions

ACID: atomicity, consistency, isolation, durability
logging
LFS, RAIDs

Preview - 1 min

Lecture - 20 min

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?

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

1.2 multiple updates

if multiple updates needed to perform 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 transfer \$100 from your bank account to mine

- (1) Debit your account
- (2) Credit my account

to update an existing file (e.g., text editor checkpoint)

- (1) overwrite block 1 of file with new data
- (2) overwrite block 2 of file with new data
- (3) ...

or

- (1) write new file with new data
- (2) remove old file from directory
- (3) add new file to 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. Approach 1 (ad-hoc)

Common in older systems

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

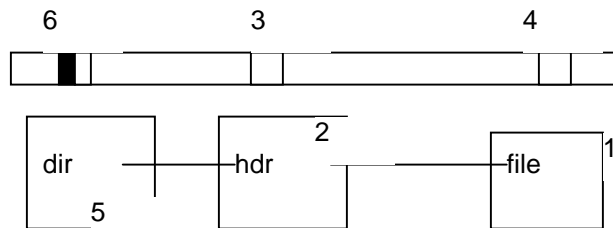
data: user bytes

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

example: for file create, first write data to file, then update file header, then mark file header “allocated” in bitmap, then mark file blocks “allocated” in bitmap, then update directory, then (if directory grew) mark new file block “allocated” in bitmap



fsck:

file header not in bitmap → only writes were to unallocated, unreachable blocks; write “disappears”

block or file header allocated, but not in bitmap → update bitmap

file created, but not yet in any directory → delete file

Challenge:

- (1) need to get ad-hoc reasoning exactly right
- (2) poor performance (synchronous writes)
- (3) slow recovery – must scan entire disk

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:

delete old file
write new file

(how vi used to work!)

now vi does the following:

write new version to temp file
move old version to other temp file
move new version to real file
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

3. Transaction

transaction – group actions together so they are:

Atomic – all or nothing. either happens or it doesn't – no partial operations

Consistent – maintains system invariants

e.g., “total deposits less total withdrawals = total accounts”

Isolated – serializable; transactions appear to happen one after another

Durable – persistent -- once it happens, stays happened

QUESTION: How does this compare to critical section?

Critical sections are atomic, serializable, consistent, but not durable

Two more terms

commit – when transaction is done (visible, durable)

rollback – “forget” uncommitted transaction (e.g. if failure occurs in middle of transaction, it didn't happen at all)

4. Implementation (one thread)

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

Illustrate with simple money transfer from acct x to acct y

Begin transaction

$x = x + 1$

$y = y - 1$

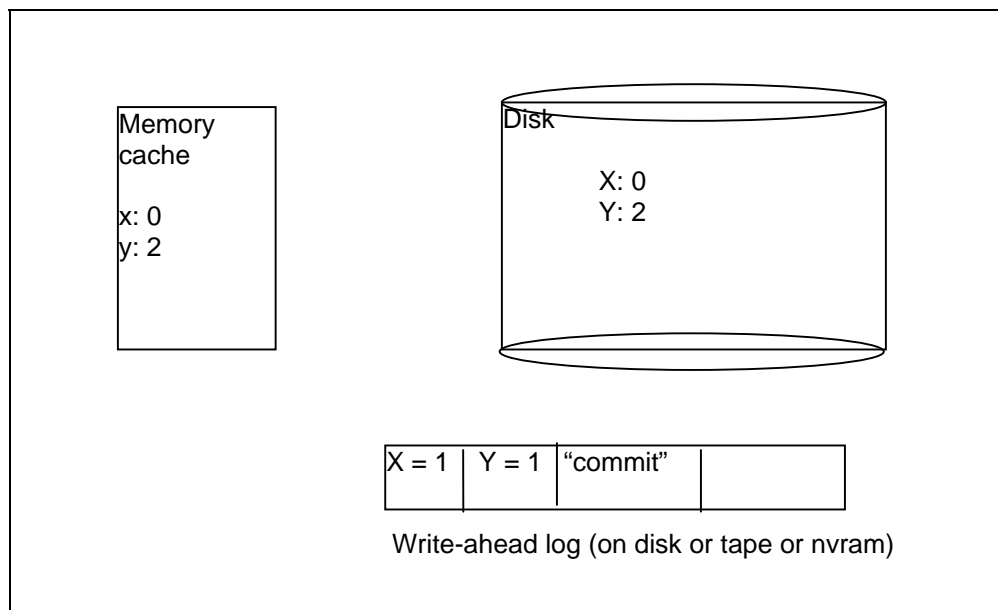
Commit

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 (and cache)
- 2) write new value of y to log (and cache)
- 3) write “commit” to log
- 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!

4.1.1 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?

4.1.2 Typical log record format:

Name of transaction:

[Previous log record of this transaction:]

[Next log record of this transaction:]

[Time:]

Type of operation:

Object of operation:

Old value:

New value:

5. Concurrency

Transaction A	transaction B
W1	
	W2
	R1

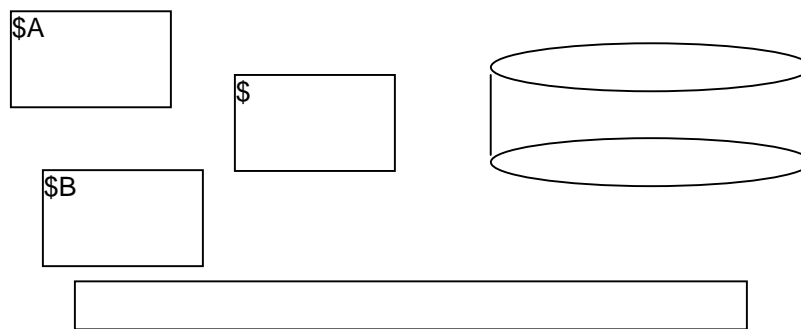
R2

W3
Commit

6. Concurrency: Reads

Requirement: Need to ensure *isolation* – transaction 1 cannot see the result of transaction 2 until transaction 2 commits.

6.1 Option 1: Per-transaction views



Store uncommitted writes with transaction
 Store committed writes in shared cache (asynchronously write to disk...once they are safely on disk, mark cache entry as “clean/replacable”)

Note for redo log

Uncommitted \rightarrow committed, cached, dirty \rightarrow committed
 cached, clean \rightarrow committed uncached

6.2 Option 2: 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. Concurrency: Writes

Requirement: durability -- once a transaction completes, its effects remain

Issue: ordering on recovery. Suppose two concurrent transactions T1 and T2 write to same location; T2 “wins” (is later) and is observed by T3; now we reboot, and during recovery T1 “wins”. E.g., log:

W2 W1 Commit1 Commit2

Solution 1: Always make sure recovery writes happen in commit order

Solution 2: Two-phase locking; still need to make sure recovery happens in commit order; but this guaranteed to also match write order in log so may be simpler

e.g.,

W2 Commit2 W1 Commit1

Admin - 3 min

Lecture - 35 min

8. Transactions in file systems

8.1 write-ahead logging

Almost all file systems built since 1985 use write-ahead logging (windows NT, solaris, OSF, Linux JFS, SGI XFS, etc)

Idea: write all changes in a transaction log (update directory, allocate blocks, etc) before sending any changes to disk

→ “create file”, “delete file”, “move file” etc are transactions

eliminates need to “fsck” after crash

If crash

read log

if log isn't committed, no change

if log is committed, apply all changes to disk

if log is zero, then all updates have gotten to disk

Advantage:

+ reliability

+ asynch write-behind (seeks)

DA: all data written twice (→ often, only log metadata)

8.2 Log-structured file system

Idea: write data only once by having log be only copy on disk

→ as you modify disk blocks, just store them out on disk in the log.

Put everything: data blocks, file headers, etc. on log

If need to get data from disk, get it from the log

- can store data blocks, indirect blocks, etc anywhere on disk, so no problem to put in log

- put inodes in log, too

→ need some way to find them

- imap is array of pointers to inodes

inodes no longer in fixed location, but imap is in fixed location (actually two fixed locations called “checkpoints”)

“apply changes to disk” now means update on-disk imap
 “replay log after crash” now means apply changes of committed transactions to imap.

Advantages

- all writes are sequential!

No seeks, except for reads, but

- RAM getting bigger → caches getting bigger
 - in extreme case (infinite cache) → disk I/O only for writes (only for durability of data)
- conclude, optimize for writes. LFS does that

Cleaning

Eventually, log wraps around – run out of room

→ have to garbage collect.

Majority of files deleted within first 5 minutes, so go back over log, and compress pieces that are no longer in use

As disk gets full, need to clean more frequently, so keep disk under-utilized

Pros & cons

- + write performance
- + read performance (if write order predicts read order)
- cleaning cost (off-line?)
- bad if disk full, random updates to files

9. RAIDS and availability

Suppose you need to store more data than fits on a single disk? How should you arrange data across disks?

One option: treat disk as huge pool of disk blocks so that:

disk 1 has blocks 1, $k+1$, $2k+1$, ...

Disk 2 has blocks 2, $k+2$, ...

...

Benefits

- load gets balanced automatically across disks
- can transfer large files at aggregate BW of all disks

Problem: what if one disk fails?

Big problem: for k disks k times as likely to have a failed disk at any given time

Availability – never lose access to data; system should continue working even if some components are not working

In RAID, dedicate one disk to hold parity for other disks in stripe

disk 1 has blocks 1, $K+1$, $2K+1$, ...

disk 2 has blocks 2, $K+2$, $2K+2$, ...

...

parity disk has blocks $\text{parity}(1\dots k)$, $\text{parity}(K+1\dots 2K)\dots$

If lose any disk, can recover data from other disks plus parity

ex:

disk 1 has 1 0 0 1

disk 2 has 0 1 0 1

disk 3 has 1 0 0 0

parity has 0 1 0 0

What if we lose disk 2? Its contents are parity of remainder!

Thus can lose any disk, and data is still available

Details:

- disk failures are “fail-stop” – disks tell you when they fail
- update – read-modify-write data and parity *atomically*
 - solution – write-ahead logging or log-structure

why does this work?

Suppose MTTF = 100K hours (11.5 years)

Department has 100 disks → 100K/100 until first failure = 1000 hours
= lose data every 41.66 days!

Suppose MTTR = 10 hours and we arrange disks as 99 disks + 1 parity

Assuming independent failures(*) – need to get unlucky and have a second failure before the first disk is fixed

e.g., 41 days until the first failure happens, then race to fix disk before next one fails. Since I fix the disk in 10 hours and the next disk is expected to fail in 1000 hours, I win this race 99 out of 100 times

→ MTDDL = 100 * 1000 = 100K hours

1% reduction in effective space gets 100x improvement in reliability!

Of course, I can improve this further by (1) improving repair time or (2) using more redundancy (e.g., 1 parity per 10 rather than 1 per 100)

$MTDDL = MTTF^2_{\text{disk}} / N * (G-1) * MTTR_{\text{disk}}$

e.g., 100 disks in groups of 9 data + 1 parity

$100K^2 / (100 * 9 * 10) = 1.1M \text{ hours } (>100 \text{ years})$

NOTE: independent failure assumption is way too optimistic

(1) “bathtub” lifetime – quoted MTTF only valid during intended service lifetime (e.g., first 3-5 years of services possibly not including burn-in)

(2) environmental correlation – power surges, vibration, manufacturing defect, ...

Summary - 1 min

Key idea: log