# Lecture 19: Transactions: Reliability from Unreliable Components

## 19.0 Main Points

Relations

Transaction concept: atomicity, durability, serializability

Write ahead, write behind logging

Log structured file systems

RAIDs and availability

## 19.1 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?

### 19.1.1 Modified data kept in memory can be lost

Options for writing data:

**Write-through**: write change immediately to disk

Problem: slow! Have to wait for write to complete before you go on.

**Write-back**: delay writing modified data back to disk (for example, until replaced).

Problem: can lose data on a crash!

### 19.1.2 Multiple updates

If multiple updates needed to perform some operation, crash can occur between them!

For example, to move a file between directories:

delete file from old directory

add file to new directory

Or create new file:

allocate space on disk for header, data

write new header to disk

add the new file to directory

What if there's a crash in the middle? Even with write-through can have problems.

## 19.2 UNIX approach (ad hoc)

**Meta-data**: needed to keep file system logically consistent (directories, bitmap, file heades, indirect blocks, etc.)

**Data**: user bytes

### 19.2.1 Meta-data consistency

For meta-data, UNIX uses synchronous write-through.

Handles multiple operations by scanning disk for in-progress actions ("fsck")

For example:

file created, but not yet put in any directory => delete file

blocks allocated, but not in bitmap => update bitmap

### 19.2.2 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 are written to disk in any order.

Sometimes meta-data consistency is good enough.

For example, how should vi or emacs write save changes to a file to disk?

delete old file

write new file

Now vi does the following:

write new version in temp file
move old version to other temp file
move new version into real file
unlink old version

If crash, look at temp area; if any files out 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 it from your account.

## 19.3 Transaction Concept

**Transactions**: group actions together so that they are:

**Atomic**: either happens or it doesn't (no partial operations)

**Serializable**: transactions appear to happen one after the other

**Durable**: once it happens, stays happened

Critical sections are atomic and serializable, not durable.

Two more terms:
Commit -- when transaction is done (durable)
Rollback -- if failure during a transaction (means it didn't happen at all)

Metaphor:
Do a set of operations tentatively. If get to commit, ok.
If not, roll back the operations as if the transaction never happened.

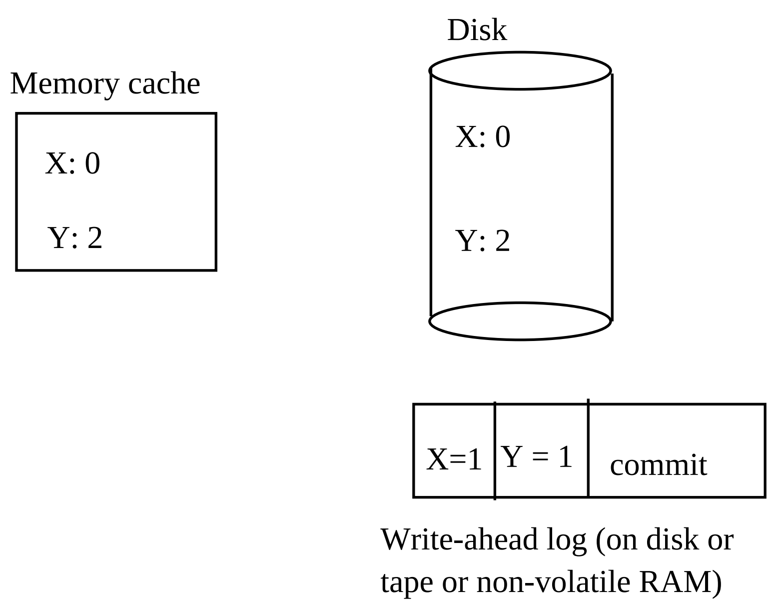
## 19.4 Transaction Implementation (one thread)

Key idea: turn multiple disk updates into a single disk write!

Illustrate with simple money transfer, from account x to account y.

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

Keep "write-ahead" (or "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 on log, transaction 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
```

What if we crash after 1?

What if we crash after 2?

What if we crash after 3 before 4 or 5?

What if we crash while we are writing "commit?" As with concurrency, need some primitive atomic operation, or else can't build anything. On most disks, 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 value along with new value. If transaction doesn't commit, "undo" change!

## 19.5 Two phase locking

What if two threads run same transaction at the same time?

Concurrency => use locks

```
Begin transaction
  Lock x, y
    x = x + 1
    y = y - 1
  Unlock x, y
 Commit
```

Problem is: B commits values for x, y, that depend on A commiting.

Solution: two-phase locking. First phase, only allowed to acquire locks. All unlocks happen at commit.

Thus, B can't see any of A's changes, until A commits and releases locks. This provides serializability!

## 19.6 Transactions in file systems

### 19.6.1 Write-ahead logging

Almost all file systems built since 1985 use write ahead logging (Windows NT, Solaris, OSF, etc). Write all changes in a transaction to log (update directory, allocate block, etc.), before sending any changes to the disk. "Create file", "delete file", "move file" are transactions.

If crash, read log:
If log isn't complete, no change!
If log is completely written, apply all changes to disk
If log is zero, then all updates have gotten to disk.

Advantage:
+ reliability
+ asynchronous write-behind
- all data written twice

### 19.6.2 Log-structured file systems

Log-structured file systems: idea is to write data only once, by having the log be the only copy of the data.
As you modify disk blocks, just store them out to disk in the log. Put everything: data blocks, file header, etc. on log.

If need to get data from disk, get it from the log -- keep map in memory to tell you where everything is (for crash recovery, have to put map on log too).

Advantages: all writes are sequential! No seeks, except for reads. But
RAM is getting cheaper => Caches getting bigger.
In extreme case (infinite size caches) -> disk I/O only for writes (only for durability of data)

Thus, optimize for writes! Logging does that.

Eventually, run out of room in the log. What happens? Have to garbage collect. Majority of files deleted in the first 5 minutes. So go back over log, and recompact pieces that are no longer in use. If disk fills up, need to clean more frequently, so keep disk under-utilized (as in DEMOS and UNIX).

Pros & cons:

+ write performance
+ reads, if file written sequentially from beginning to end
- cleaning cost (can be done in background)
- what if little updates to the middle of large files?

## 19.7 RAIDs and availability

Suppose you need to store more data than fits on a single disk (e.g., large databases or file servers). How should you arrange data across disks?

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

disk1 has blocks 1..N
disk2 has blocks N+1..2N
etc.

Another option: RAID = Redundant Arrays of Inexpensive Disks

Idea is to stripe data across disks. With k disks:

disk1 has blocks 1, k+1, 2k+1, ...
disk2 has blocks 2, k+2, 2k+2, ...
etc.

Benefits:

1. Load gets automatically balanced among disks.
2. Can transfer large file at aggregate bandwidth of all disks.

Problem: what if one disk fails?
Availability: never lose access to data. System should continue to work even if some components are not working.

In RAID, dedicate one disk to hold bitwise parity for other disks in stripe. With k+1 disks:

disk1 has blocks 1, k+1, 2k+1, ...
disk2 has blocks 2, k+2, 2k+2, ...
...
parity disk has blocks parity(1..k), parity(k+1..2k), ...

If lose any disk, can recover data from other disks plus parity:
Ex: disk1 holds 1001
disk2 holds 0101
disk3 holds 1000
parity disk: 0100

What if we lose disk2? Its contents are parity of remainder!
Thus, can lose any disk, and data is still available.

How do you update a disk block on a RAID? Have to update both data and parity.

Of course, data & parity update must appear to happen as one transaction, so need write ahead logging or log structure for RAIDs.