Motivation for Transactions

Ensuring Reliability from Unreliable Components

For example, file systems have lots of data structures that the OS caches to get good performance, but we need to keep them correct even in case of failures.

- Bit map of free blocks
- Directories
- File headers
- Indirect blocks
- Data blocks

Keeping them accurate is easy to do if we just read these structures, harder when we write them.

Modified data kept in memory can be lost!

Write-through (now) - write change immediately back to disk. But this is slow since we have to wait for the write to hit the disk and get an interrupt.

Write-back (later) - delay writing the modified data to disk until for example, the page is replaced in memory (or the line in cache). But this can result in inconsistencies since the data can be lost in a crash.

Reliability From Unreliable Components

Multiple Updates - If multiple updates are needed to perform some operation, the crash can occur between them. For example:

- move a file between directories
  1. add the file to the new directory, and
  2. delete the file from the old directory
- create a new file
  1. allocate space on disk for the header and data,
  2. write header to disk
  3. add new file to the directory
  4. write data to disk

UNIX Approach to Reliability (ad hoc)

Break down of requirements into meta-data and regular data.

Meta-Data: is the data needed to keep the file system logically consistent (directories, bit maps, headers, indirect blocks, etc.)

- To keep this data consistent, UNIX uses synchronous write-through.
- If UNIX needs to perform multiple updates, it does them in a specific order, so that if there is a crash, it can run a special program "fsck" that scans the entire disk for internal consistency to check for any "in progress" operations, and then fixes them up.

Example:

1. If the OS created a file, but did not put it in a directory, then UNIX deletes the file.
2. If the OS allocated blocks to a file, but did not change the bit map, then UNIX updates the bit map.

UNIX Approach to Reliability (ad hoc)

User Data Consistency: In UNIX, user data uses write back for performance, but every 30 seconds the OS writes all modified blocks back to disk (or the user can issue a "sync" to force the OS to send all outstanding writes to disk immediately).

- This method does not guarantee blocks are written to disk in any particular order.
- User programs that care about consistency and reliability can do things like the following example:
  - To write a file 'name' in "vi":
    1. write new version to file 'tmp1'
    2. move old version to file 'tmp2'
    3. move new version into file 'name'
    4. remove 'tmp2' and 'tmp1'
  - If there is a crash, look at the tmp area; if any files are in there, send an email to the user that there might be a problem.
- If the user program wants to have multiple file operations happen as a unit (e.g., ATM gives you $100 and debits your account), this is not good enough!

Transaction Terminology

Transactions: group actions together so that the are

Atomic: they all happen or they all don't.

Serializable: transactions appear to happen one after the other.

Durable: once it happens, it sticks.

Critical sections give us atomicity and serializability, but not durability. To get durability, we need to be able to:

Commit - indicate when a transaction is finished.

Roll back - if we have a failure in the middle of a transaction, we need to be able to undo what we have done so far (make it all not happen).

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

Transaction Implementation Using One Thread

Idea: Fix the problem of multiple updates to disk by turning multiple updates into a single disk write. For example, on a transfer of money from one account to another:

```
begin transaction
  x = x + 100
  y = y - 100
Commit
```

- Keep "write-ahead" or "redo" log on disk of all changes in the transaction.
- If we do not erase the log, it records everything the thread does (or tries!) to do.
- Once the OS writes both changes on the log, the transaction is committed.
- Then "write behind" changes to disk, logging all writes.
- If the crash comes after a commit, the log is replayed.

Transaction Implementation Using One Thread

We need atomic operations. We assume that writing a single sector to disk is atomic.

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 to log
4. write x to disk, record on log
5. write y to disk, record on log
6. reclaim space on log

Where can we crash?

1. After 1 or 2: Since no commit, no changes are on disk so we can ignore the log.
2. After 3: Commit is written to log, so do 4, 5, and 6.
3. After 4: Do 5 and 6.
4. After 5: Do 6.

Could we write x back to disk before commit? Only if we keep a log of the old value with the new value and if we do not get a commit, undo the change.

Transaction Implementation with Two Phase Locking

How do we handle multiple threads trying to access the same data at the same time? Consider the following:

```
begin transaction
  lock x, y
    x = x + 100
    y = y - 100
  unlock x, y
Commit
```

Given two threads A & B that execute the above code in the following sequence

1. A gets the lock, reads and modifies x and y, writes to the log, and unlocks.
2. Then B grabs the lock before A commits.
3. B reads A's modifications, then modifies x and y, writes to the log, unlocks, and commits.
4. Then the system crashes before A commits

Problem: B commits values for x and y that depend on A, but A never committed.

Solution: Two phase locking requires that in the first phase threads may only acquire locks. All unlocks happen at the commit.

$\Longrightarrow$ B cannot see any of A's changes until A commits and releases the locks. This protocol provides serializability.

Transaction in File Systems

Write-ahead Logging

- Almost all file systems built since 1985 use write ahead logging (Solaris, OSF, Windows NT, etc.).
- Write all file changes to a transaction log (update directory, allocate block, create file, move file, etc.) before sending any changes to disk.
- This strategy eliminates any need for file system check (fsck) after a crash.
- If there is a crash, read the log.
  1. If log is zero, then all updates made it to disk, do nothing.
  2. If the log is not complete (no commit), do nothing.
  3. If the log is completely written, apply changes that are left to disk

Advantages: reliability, asynchronous write-behind

Disadvantages: all data is written twice (other approaches try to eliminate this overhead, but they have other problems.)