28 File Locking

When files are shared among processes, concurrent accesses can lead to race conditions, similar to those that can occur in shared memory. To allow cooperating processes to coordinate access to shared files, filesystem provides lock primitives. Dimensions in file locking:

- Granularity
  - file lock: can lock entire file only
  - record lock: can lock arbitrary region of file (locks held on non-overlapping regions of a file do not conflict)
- Semantics
  - shared lock: multiple processes can acquire shared locks for the same (region of) a file
  - exclusive lock: if a process hold an exclusive lock on a (region of) a file, no other process can hold a shared or exclusive lock on the same (region of a) file.
- Enforcement
  - mandatory lock: OS prohibits access to files by processes that do not hold the appropriate lock.
  - advisory locks: OS does not restrict access to files based on lock ownership. Processes can chose to ignore locking.

29 Filesystem Reliability

Loss of data in a filesystem can have catastrophic effects (worse than failure of hardware.) Need to ensure reasonable level of safety against data loss in the event of system failures. Threats:

1. media (disk) failure
2. accidental or malicious deletion of data by users
3. system crash during filesystem modifications, leaving data on disk in an inconsistent state
4. during a security compromise, an attacker can delete or tamper with data

Backup: copy entire filesystem onto low-cost media (tape) at regular intervals (e.g., once a day). Keep backup media off-line, where they can't be accessed during a security incident. In the event of a failure of the primary media (disk), accidental deletion, corruption due to crash, or security compromise, can restore data from backup media. Amount of data loss is limited to modifications that occured since the last backup. (1, 2, 3, 4)

Mirrored disks: multiple copies of the filesystem are maintained on independent disks. Disk writes update all the redundant disks in parallel. Used in applications that cannot tolerate any data loss (e.g., banking). (1)

RAID (Redundant Array of Inexpensive Disks): use multiple parallel disk drives for higher throughput and increased reliability. For example (RAID-5): store a different 512 byte chunk of a 4KB block on each of 8 disks. A ninth disk stores a 512 byte *parity chunk*. Parity chunk is the XOR of the 8 data chunks. As a result, can recover the data if any one of the nine disks fails at the cost of 8/9th the capacity. (1) (Also get 8 times the sequential r/w bandwidth of a single disk; but: small writes of less than 4KB are expensive.)

Versioning filesystem: filesystem creates a new version every time a file is modified. Old versions are kept until explicitly deleted. Can use copy-on-write to reduce storage requirements. (1, 2)

Snapshoting filesystem: filesystem takes a snapshot of the entire filesystem at fixed intervals (e.g., daily). Snapshots are kept until explicitly deleted. Can use copy-on-write to reduce storage requirements. (1, partly 2)

After a system crash in the middle of a filesystem operation, filesystem metadata may be in an inconsistent state. (Invariants of the on-disk metadata may not hold.) Example: a file was deleted, but its disk blocks have not yet been added to the freelist. (3)

Solution 1: run a program during system startup that examines the entire filesystem, detects inconsistencies, and restores the invariants. In Unix, that programs is called fsck. Some of the conditions fsck checks and repairs:

- correct i-node reference counts
- missing blocks in the freelist
- blocks that are both in the freelist and part of a file
- incorrect information in the superblock
- out-of-range block and i-node numbers
- disconnected (unreachable) files and directories

This used to be the preferred solution, but today's filesystems have become so big that running fsck could take many hours, during which the filesystem would remain unavailable.

Solution 1a: Soft updates. Allows delayed writes of metadata blocks to disk, but tracks dependencies among metadata updates and ensures they are written in a particular order that ensures recoverability using fsck in the case of a failure.

Solution 2: maintain a log for metadata updates (journaling filesystem)

- log is a circular buffer stored on disk (or second disk/SSD)
- during each (sequence of) operation(s) that modify the metadata:
    1. write modified metadata blocks into log, followed by a "commit" record
    2. write the blocks into their actual locations in the filesystem
    3. add a "done" record indicating the operation was completed

- after a crash:
    1. check the log
    2. if there are incomplete operations (a "commit" not followed by a matching "done" record), copy the modified blocks into their actual locations in the filesystem
- before over-writing blocks/records in the log, make sure they are followed by a "done" record

A: Metadata updates are effectively atomic wrt to crashes. Recovery time is proportional to the number of operations not yet commmited to disk, thus very short (compared to an fsck).
D: Each modified metadata block is written twice; fortunately, log writes are efficient because they are sequential.

Solution 3: maintain only a log (log-structured filesystem)

- write updates (data and metadata) only into the log
- writes are very efficient (sequential)
- must read log in reverse to find most recent version of each disk block
- $\rightarrow$ requires a large cache for efficient reads
- when the log wraps, need to check if existing entries are still live
- if so, need to compact live entries (log cleaning)

30 Transactions

Application programs have their own invariants that must hold about the state (contents) of their files. After a system crash, the system cannot restore these invariants, since it has no knowledge of them. Need tools that allow applications to maintain their invariants on persistent data despite crashes.

Transactions are a high-level mechanism that allows applications to specify transitions from one consistent state to another. They can be used by the system to both ensure synchronization (in the event of concurrent access), and all-or-nothing semantics (in the event of system crashes.)

Transaction Primitives:

```
tid = startTransaction();
{sequence of read/write operations to one or more files}
if (anythingGoesWrong)
    abortTransaction(tid);
else {
    {more read/write operations}
    commitTransaction(tid);
}
```

Properties of transactions (ACID):

- Atomicity: either all or no part of a transaction completes
- Consistency: transactions take files from one consistent state to another
- Isolation: concurrent transactions do not interfere with each other
- Durability: once a transaction commits, the changes are permanent

Implementation of transactions:

- atomicity, durability:

- Sol 1: Shadowing. Keep separate, private copies of each modified file blocks, leaving the original file unchanged. When the transaction commits, atomically copy the shadow pages to the original pages. (Difficult to make this work for transactions that write multiple files/tables.)
- Sol 2: Transaction log (write ahead logging). Modify original files directly, but write first into the log all changes that were made (with the original values) . If the transaction aborts or the system fails before the transaction commits (i.e., no commit record in the log upon reboot), use that log to undo all the changes that were made ("rollback").

- Consistency, Isolation: Two-phase locking (2PL)

  In phase 1, locks are acquired and none released. In phase 2, locks are released and none acquired.

  Locks can be shared (read lock) or exclusive (write lock).

  Guarantees *Serializability*: The effect of a set of transactions is as if the transaction had been executed serially in some order.

There is much more to transactions. Transactions play a central role in databases.