Discussion 9: Reliability

April 15, 2022

Contents

1 Reliability

Availability is the probability that the system can accept and process requests. This is measured in "nines" of probability (e.g. 99.9% is said to be "3-nines of availability").

Durability

Durability is the ability of a system to recover data despite faults (i.e. fault tolerance). It's important to note that durability does not necessarily imply availability.

When making a file system more durable, there are multiple levels which we need to concern ourselves with. For small defects in the hard drive, Reed-Solomon error correcting codes can be used in each disk block. When using a buffer cache or any other delayed write mechanism, it's important to make sure dirty data gets written back to the disk. To combat unexpected failures or power outages, the computer can be built with a special, battery-backed RAM called non-volatile RAM (NVRAM) for dirty blocks in the buffer cache.

To make sure the data survives in the long term, it needs to be replicated to maximize the independence of failures. **Redundant Array of Inexpensive Disks (RAID)** is a system that spreads data redundantly across multiple disks in order to tolerate individual disk failures. RAID 1 will **mirror** a disk onto another "shadow" disk. Evidently, this is a very expensive solution as each write to a disk actually incurs two physical writes. RAID 5 will stripe data across n multiple disks to allow for a single disk failure. Each stripe unit consists of $n - 1$ blocks and one **parity block**, which is created by XOR-ing the $n - 1$ blocks. To recover from a disk failure, the system simply needs to XOR the remaining blocks.

Reliability

Reliability the ability of a system or component to perform its required functions under stated conditions for a specified period of time. This means that the system is not only up (i.e. availability), but also performing its jobs correctly. Reliability includes the ideas of availability, durability, and security.

One approach taken by FAT and FFS (in combination with `fsck`) is **careful ordering and recovery**. For instance, creating a file in FFS may be broken down into the following steps

1. Allocate data block.
2. Write data block.
3. Allocate inode.
4. Write inode block.
5. Update free map.
6. Update directory entry.
7. Update modify time for directory entry.

To recover from a crash, `fsck` might take the following steps.

1. Scan inode table.
2. If any unlinked files (not in any directory), delete or put in lost and found directory.
3. Compare free block bitmap against inode trees.
4. Scan directories for missing update/access times.

It's important to note that this is not a foolproof method. While there are a few ways that failures can happen in spite of this method, the file system will be recoverable most of the times.

The other approach is to use **copy on write (COW)**. Instead of updating data in-place, new versions are written to a new location on disk, and the appropriate mappings and references to these data are subsequently updated. Since the mappings and references are updated last, this allows for easy recovery if the system crashes sometime in the middle of updating data since the old data and mapping will still be in tact. Furthermore, data is being only added, not modified, so batch updates and parallel writes can help improve performance.

Transactions

A more general way to handle reliability is through the use of **transactions**, indivisible units which execute independently from other transactions. Transactions typically follow the **ACID properties**.

Atomicity
A transaction must occur in its entirety or not at all.

Consistency
A transaction takes system from one consistent state (i.e. meets all integrity and correctness constraints) to another.

Isolation
Each transaction must *appear* to execute on its own.

Durability
A committed transaction's changes must persist through crashes.

The idea of a transaction is similar to that of a critical section with the newly added constraint of durability. Each entry in a transaction needs to be **idempotent**, which have the same effect when executed once or many times (i.e. $f(f(x)) = f(x)$).

Transactions are recorded on **logs/journals** which are stored on disk for persistence. Writes to a Log are assumed to be atomic. Logs will typically use a circular buffer as its data structure, which maintains a head and tail pointer.

Transactional file systems use the idea of transactions and logs to make the file system more reliable. There are two main types of transactional file systems: journaling and log structured.

Journaling file systems use **write-ahead logging (WAL)** where log entries are written to disk *before* the data gets modified. During the preparation phase, all planned updates are appended to the log. Each log entry is tagged with the transaction id. During the commit phase, a commit record is appended to the log, indicating the transaction must happen at some point. Next, the write back will take place *asynchronously* after the commit phase where the transaction's changes will be applied to persistent storage. In the background, garbage collection will take place to clean up fully written back transactions. When recovering from a crash, only the committed transactions are replayed to restore the state of the file system.

Log structured file systems (LFS) use the log as the storage. The log becomes one contiguous sequence of blocks that wrap around the whole disk.

1.1 Concept Check

1. What benefit with regards to read bandwidth might you see from using RAID 1?

   Read bandwidth can be doubled since there are two copies of the same data.

2. What is the minimum number of disks to use RAID 5?

   3. A disk is recovered by XOR-ing at least two other disks. It's important to note that the RAID level is not an indication of how many disks are needed.

3. RAID 4 had a dedicated disk with all the parity blocks. On the other hand, RAID 5 distributes the parity blocks across all disks in a round robin fashion. Why is this approach beneficial in terms of write bandwidth?

When updating data, the parity block always needs to be written to. If all the parity blocks are in one disk like in RAID 4, this becomes a bottleneck as multiple writes to disk ultimately contend on one disk. As a result, it's better to distribute the parity blocks evenly such that more writes can happen without contending for the same disk.

4. How can COW help with write speeds?

COW can transform random I/O into sequential I/O. For instance, take the example of appending a block to a file. In a traditional in-place system like FFS, the free space bitmap, file's inode, file's indirect block, and file's data block all need to be updated. On the other hand, COW could just find sequential unused blocks in disk and write the new bitmap, inode, indirect block, and data block.

5. Why does each entry in a transaction need to be idempotent?

On recovery, all the entries from a committed transaction are replayed. There is a possibility that the data corresponding to an entry was already modified prior to a crash. A non-idempotent operation would put us in an undesired state.

6. Consider a file system with a buffer cache. Your program creates a file called `pintos.bean`. While the changes have been logged with a commit entry, the tail of the log still points to before the start of the log entry corresponding to creating `pintos.bean`. Assuming no further disk reads or writes have happened, if your program wants to read `pintos.bean`, will it need to scan through the logs?

No. The buffer cache will still hold the blocks corresponding to `pintos.bean`.

7. What is the purpose of a commit entry in a log?

The commit entry makes each transaction atomic. These changes to the file system's on-disk structures are either completely applied or not applied at all. For instance, the creation of a file involves multiple steps (e.g. allocating data blocks, setting up the inode) that are not inherently atomic, nor is the action of recording these actions in the journal, but we want to treat these steps as a single logical transaction. Appending the final commit entry to the log (i.e. a single write to disk) *is* assumed to be an atomic operation and serves as the "tipping point" that guarantees the transaction is eventually applied.

1.2 Journaling

You create two new files, F_1 and F_2, right before your laptop's battery dies. You plug in and reboot your computer, and the operating system finds the following sequence of log entries in the file system's journal.

1. Find free blocks $x_1, x_2, \ldots, x_n$ to store the contents of F_1, and update the free map to mark these blocks as used.
2. Allocate a new inode for the file F_1, pointing to its data blocks.
3. Add a directory entry to F_1's parent directory referring to this inode.
4. *Commit*
5. Find free blocks $y_1, y_2, \ldots, y_n$ to store the contents of F_2, and update the free map to mark these blocks as used.

6. Allocate a new inode for the file F_2, pointing to its data blocks.

You may assume a single write to disk is an atomic operation.

1. What are the possible states of files F_1 and F_2 *on disk* at boot time?

   File F_1 may be fully intact on disk, with data blocks, an inode referring to them, and an entry in its parent directory referring to this inode. There may also be no trace of F_1 on disk outside of the journal if its creation was recorded in the journal but not yet applied. F_1 may also be in an intermediate state (e.g. data blocks may have been allocated in the free map, but there may be no inode for F_1, making the data blocks unreachable)

   F_2 is a simpler case. There is no *Commit* message in the log for F_2, so we know these operations have not yet been applied to the file system.

2. Say the following entries are also found at the end of the log.

   7. Add a directory entry to F_2's parent directory referring to F_2's inode.
   8. *Commit*

   How does this change the possible states of file F_2 on disk at boot time?

   The situation for F_2 is now the same as F_1: the file and its metadata could be fully intact, there could be no trace of F_2 on disk, or any intermediate between these two states.

3. Say the log contained only entries (5) through (8) shown above. What are the possible states of file F_1 on disk at the time of the reboot?

   We can now assume that F_1 is fully intact on disk. The log entries for its creation are only removed from the journal when the operation has been fully applied on disk.

4. When recovering from a system crash and applying the updates recorded in the journal, does the OS need to check if these updates were partially applied before the failure?

   No. The operation for each log entry is assumed to be idempotent. This greatly simplifies the recovery process, as it is safe to simply replay each committed transaction in the log, whether or not it was previously applied.