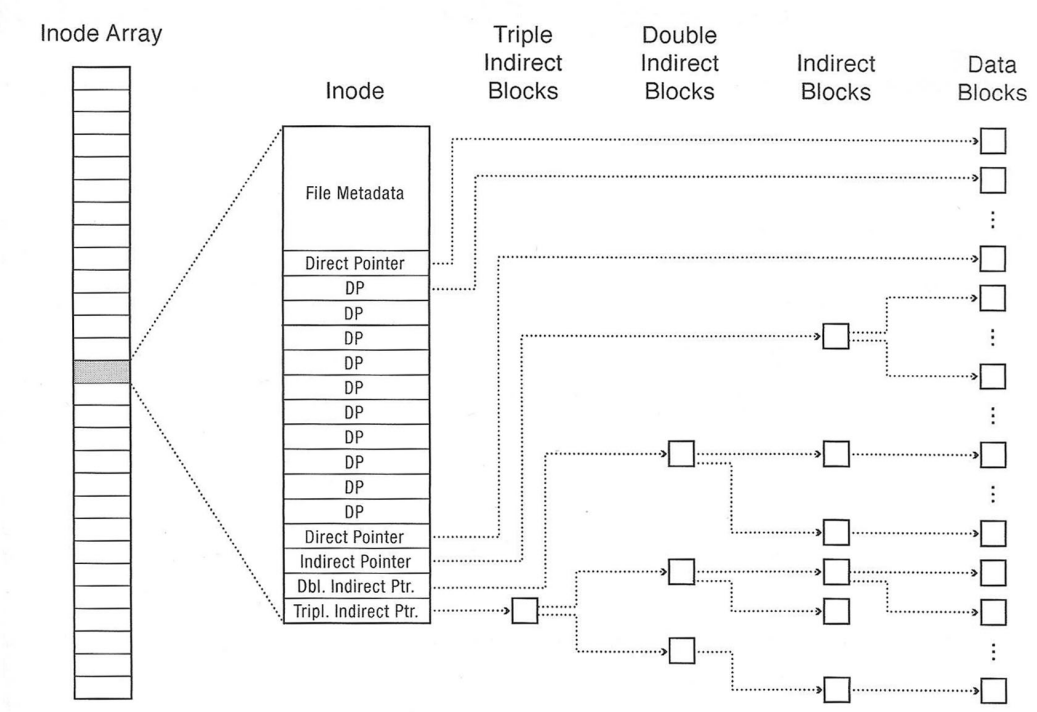
Section 12: File Systems and Reliability, Two Phase Commit

CS162

April 13, 2018

Contents



1 Warmup

What are the ACID properties? Explain each one and discuss the implications of a system without that property.

Name 2 different RAID levels that offer redundancy. For each level, explain how a recovery program could recover data from a degraded array.

Explain the difference between a hard link and a soft link (symbolic link).

How could you implement hard links for the FAT file system? What problem would you encounter?

What is a journaled file system? Explain the purpose of the file system's "journal".

Discuss the advantages and drawbacks of memory mapped file accesses compared to traditional disk accesses for small random file reads and writes to many files of varying size.

2 Vocabulary

- **Unix File System (Fast File System)** - The Unix File System is a file system used by many Unix and Unix-like operating systems. Many modern operating systems use file systems that are based off of the Unix File System.
- **inode** - An inode is the data structure that describes the metadata of a file or directory. Each inode contains several metadata fields, including the owner, file size, modification time, file mode, and reference count. Each inode also contains several data block pointers, which help the file system locate the file's data blocks.

  Each inode typically has 12 direct block pointers, 1 singly indirect block pointer, 1 doubly indirect block pointer, and 1 triply indirect block pointer. Every direct block pointer directly points to a data block. The singly indirect block pointer points to a block of pointers, each of which points to a data block. The doubly indirect block pointer contains another level of indirection, and the triply indirect block pointer contains yet another level of indirection.
- **Transaction** - A transaction is a unit of work within a database management system. Each transaction is treated as an indivisible unit which executes independently from other transactions. The ACID properties are usually used to describe reliable transactions.
- **ACID** - An acronym standing for the four key properties of a reliable transaction.

  Atomicity - the transaction must either occur in its entirety, or not at all.

  Consistency - transactions must take data from one consistent state to another, and cannot compromise data integrity or leave data in an intermediate state.

  Isolation - concurrent transactions should not interfere with each other; it should appear as if all transactions are serialized.

  Durability - the effect of a committed transaction should persist despite crashes.
- **Idempotent** - An idempotent operation is an operation that can be repeated without effect after the first iteration.
- **Logging file system** - A logging file system (or journaling file system) is a file system in which all updates are performed via a transaction log ("journal") to ensure consistency, in case the system crashes or loses power. Each file system transaction is first written to an append-only redo log. Then, the transaction can be committed to disk. In the event of a crash, a file system recovery program can scan the journal and re-apply any transactions that may not have completed successfully. Each transaction must be idempotent, so the recovery program can safely re-apply them.
- **TPC/2PC** - Two Phase Commit is an algorithm that coordinates transactions between one coordinator (Master) and many slaves. Transactions that change the state of the slave are considered TPC transactions and must be logged and tracked according to the TPC algorithm. TPC ensures atomicity and durability by ensuring that a write happens across ALL replicas or NONE of them. The replication factor indicates how many different slaves a particular entry is copied among. The sequence of message passing is as follows:

```
for every slave replica and an ACTION from the master,
origin [MESSAGE] -> dest :
---
MASTER [VOTE-REQUEST(ACTION)] -> SLAVE
SLAVE [VOTE-ABORT/COMMIT] -> MASTER
MASTER [GLOBAL-COMMIT/ABORT] -> SLAVE
```

```
SLAVE [ACK] -> MASTER
```

If at least one slave votes to abort, the master sends a GLOBAL-ABORT. If all slaves vote to commit, the master sends GLOBAL-COMMIT. Whenever a master receives a response from a slave, it may assume that the previous request has been recognized and committed to log and is therefore fault tolerant. (If the master receives a VOTE, the master can assume that the slave has logged the action it is voting on. If the master receives an ACK for a GLOBAL-COMMIT, it can assume that action has been executed, saved, and logged such that it will remain consistent even if the slave dies and rebuilds.)

3 Problems

3.1 Extending an inode

Consider the following `inode_disk` struct, which is used on a disk with a 512 byte block size.

```
/* Definition of block_sector_t */
typedef uint32_t block_sector_t;

/* Contents of on-disk inode. Must be exactly 512 bytes long. */
struct inode_disk
  {
    off_t length;                       /* File size in bytes. */
    block_sector_t direct[12];          /* 12 direct pointers */
    block_sector_t indirect;            /* a singly indirect pointer */
    uint32_t unused[114];               /* Not used. */
  };
```

Why isn't the file name stored inside the `inode_disk` struct?

What is the maximum file size supported by this inode design?

How would you design the in-memory representation of the indirect block? (e.g. the disk sector that corresponds to an inode's `indirect` member)

Implement the following function, which changes the size of an inode. If the resize operation fails, the inode should be unchanged and the function should return `false`. Use the value `0` for unallocated block pointers. You do not need to write the inode itself back to disk. You can use these functions:

- "`block_sector_t block_allocate()`" – Allocates a disk block and returns the sector number. If the disk is full, then returns 0.
- "`void block_free(block_sector_t n)`" – Free a disk block.
- "`void block_read(block_sector_t n, uint8_t buffer[512])`" – Reads the contents of a disk sector into a buffer.
- "`void block_write(block_sector_t n, uint8_t buffer[512])`" – Writes the contents of a buffer into a disk sector.

```
bool inode_resize(struct inode_disk *id, off_t size) {
    block_sector_t sector;  // A variable that may be useful.









}
```

3.2 Network Layering and Fundamentals

What is the purpose of layering?

What are the 5 basic network layers?

Which layer is responsible for maintaining reliability? Give an example of a protocol at this level that is reliable.

What is the end-to-end principle?

3.3 A Simple 2PC

Suppose you had a remote storage system composed of a client (you), a single master server, and a single slave server. All units are separated from each other and communicate using RPC. There is no caching or local memory; all requests are eventually serviced using the backing store (disk) of the slave. The slave guards itself against failure by committing entries to a non-volatile log that never gets deleted in the event of a crash. The system only understands PUT(VALUE) and DEL(VALUE) commands, where VALUE is an arbitrary string. Calls to DEL on values that dont exist cause the slave to VOTE-ABORT.

Suppose you issue the following sequence of commands. Recall that the correct sequence of message passing is CLIENT - MASTER - SLAVE - MASTER - CLIENT. Calls to PUT on values that already exist cause VOTE-ABORT.

- PUT(I LOVE)

- PUT(OPERATING SYSTEMS)

- DEL(I LOVE)

- DEL(I LOVE)

- PUT(GOBEARS)

What is the sequence of messages sent and received by the MASTER server? List communications with the slave only. Your answer should be a list of the form:

SEND: PUT(XXX)

RECIEVE: VOTE-XXX

SEND: DEL(XXX)

...

What is the sequence of messages committed to the log of the slave?