Section 11: File Systems and Reliability, Two Phase Commit

CS162

November 6, 2018

Contents

1 Warmup **2**

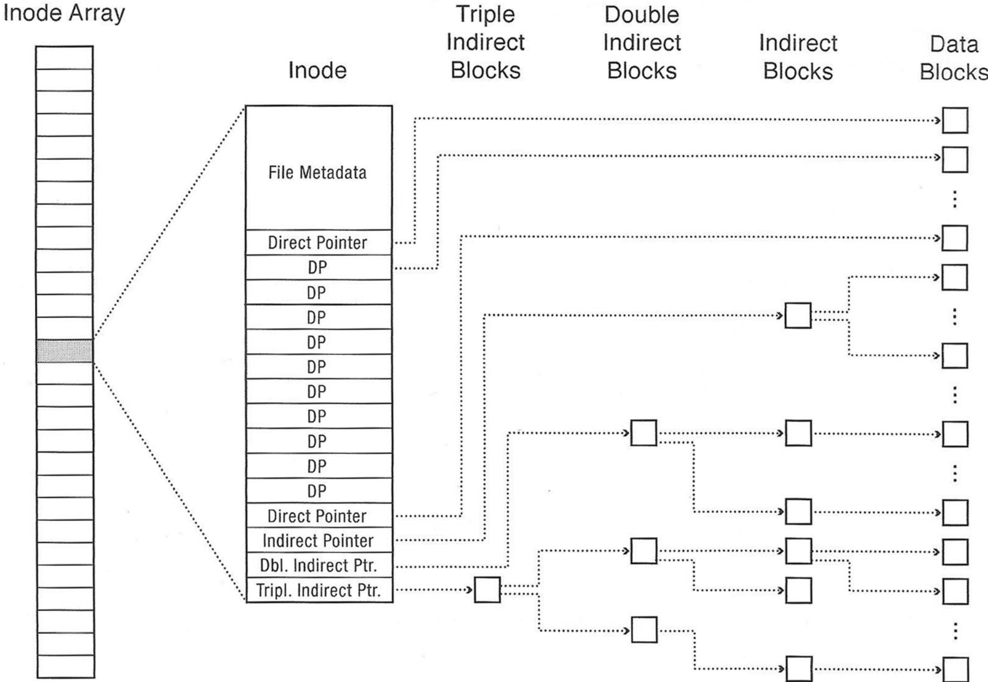
2 Vocabulary **4**

3 Problems **6**

3.1 Extending an inode 6

3.2 Network Layering and Fundamentals 11

3.3 A Simple 2PC 11



1 Warmup

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

Atomicity - data left in intermediate state
Consistency - data left in invalid state
Isolation - concurrent transactions may interfere with each other
Durability - crashes may destroy committed transactions

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

RAID 1 - The recovery program copies all the data from the good disk to a replacement
RAID 5 - The recovery program uses the data and parity bits from the N - 1 good disks to rebuild the data and parity that should belong on the Nth disk.

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

A hard link is just a directory entry. Multiple hard links can exist for a single inode. Modifying the name of a hard link will not modify the inode itself or any other hard links to that inode. Multiple hard links to the same file must all be on the same filesystem, since hard links point to an inode, and an inode can only be on a single filesystem. Hard links contribute to an inode's reference count.

A soft link (symbolic link) is a special file system object that contains a path to another file or directory. When programs access soft links, the standard library will resolve the soft link's target file. Soft links do not guarantee the existence of the target, like hard links do. Soft links can point to files and directories in different filesystems. Soft links do not contribute to the reference count.

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

In FAT, each directory entry contains the first block number for that file. This is because the blocks of a file is determined by its first block, i.e. the entry point into the FAT. Thus, a hard link might be implemented in a FAT system by having a directory entry contain the same first block number as another directory entry.

The problem is that a FAT entry, unlike an inode, doesnt keep track of how many directory entries refer to that entry. Thus, if a user deletes a file the block of that file will be put back on the list of free blocks, even if another hard link to that file exists.

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

A journaled file system maintains a journal, which contains the most recent updates to file system metadata. The journal is used to recover the file system's state, in case of a power failure or a system crash.

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.

Memory mapped files allow for in-place updates without the need to seek into the file to the appropriate location.

However, since this access pattern accesses data in many different files, it will incur overhead for paging the files in and out. This may or may not be worse than the overhead of traditional I/O, which involves overhead in the syscall framework and copying memory.

2 Vocabulary

- **Memory-Mapped File** A memory-mapped file is a segment of virtual memory which has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource. This resource is typically a file that is physically present on-disk, but can also be a device, shared memory object, or other resource that the operating system can reference through a file descriptor. Once present, this correlation between the file and the memory space permits applications to treat the mapped portion as if it were primary memory.

- **Memory-Mapped I/O** Memory-mapped I/O (not to be confused with memory-mapped file I/O) uses the same address bus to address both memory and I/O devices the memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices.

- **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.

- **Unix File System (Fast File System)** - Most files are small in size, but disk storage is mostly used up by a few large files. The Unix File System was designed with this in mind and uses an inode based structure to allow for files very small in size that can scale up to larger if necessary. To allow for easier allocation of contiguous blocks within a file, the UFS uses bitmap allocation. Lastly, the FFS introduced various allocation and placement policies built on top of UFS.

- **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?

> The file name belongs in the directory entry.

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

> It is $2^{16} + 12 \times 2^9$ bytes

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)

> You could use a `block_sector_t[128]`, which is exactly 512 bytes.

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;
  for (int i = 0; i < 12; i++) {
    if (size <= 512 * i && id->direct[i] != 0) {
      block_free(id->direct[i]);
      id->direct[i] = 0;
    }
    if (size > 512 * i && id->direct[i] == 0) {
      sector = block_allocate();
      if (sector == 0) {
        inode_resize(id, id->length);
        return false;
      }
      id->direct[i] = sector;
    }
  }
  if (id->indirect == 0 && size <= 12 * 512) {
    id->length = size;
    return true;
  }
  block_sector_t buffer[128];
  if (id->indirect == 0) {
    memset(buffer, 0, 512);
    sector = block_allocate();
    if (sector == 0) {
      inode_resize(id, id->length);
      return false;
    }
    id->indirect = sector;
  } else {
    block_read(id->indirect, buffer);
  }
  for (int i = 0; i < 128; i++) {
    if (size <= (12 + i) * 512 && buffer[i] != 0) {
      block_free(buffer[i]);
      buffer[i] = 0;
    }
    if (size > (12 + i) * 512) && buffer[i] == 0) {
      sector = block_allocate();
      if (sector == 0) {
        inode_resize(id, id->length);
        return false;
      }
      buffer[i] = sector;
    }
  }
  block_write(id->indirect, buffer);
  id->length = size;
  return true;
}
```

Another solution that you may find useful

```
#include <stdio.h>
#include <unistd.h>

typedef uint32_t block_sector_t;

struct inode_disk
{
    off_t length; /* File size in bytes. */
    block_sector_t pointers[13]; /* 12 direct pointers and 1 indirect pointer*/
    uint32_t unused[114]; /* Not used. */
};

struct indirect_disk
{
    block_sector_t pointers[128];
}

bool calculate_indices(int blocknumber, int *offsets, int *offset_cnt)
{
    if (sector_idx < 12)
    {
        offsets[0] = sector_idx;
        *offset_cnt = 1;
        return true;
    }
    sector_idx -= 12;
    if (sector_idx < PTRS_PER_SECTOR)
    {
        offsets[0] = 12;
        offsets[1] = sector_idx % PTRS_PER_SECTOR;
        *offset_cnt = 2;
        return true;
    }
    return false;
}

bool inode_change_block(struct inode_disk *id, block_sector_t block, bool add)
{
    int offsets[2];
    int offset_cnt;
    int i = 0;
    uint8_t zeros[512];
    struct indirect_disk cur;
    block_sector_t cur_indirect;

    memset(zeros, 0, sizeof(zeros));
    calculate_indices(block, offsets, &offset_cnt);
```

```
    for (i = 0; i < offset_cnt; i++)
    {
        if (i == 0)
        {
            if (add && id->pointers[offsets[0]] == 0)
            {
                block_sector_t next_indirect;
                if ((next_indirect = block_allocate()) == 0)
                    return false;
                id->pointers[offsets[0]] = next_indirect;
                block_write(next_indirect, zeros);
                cur_indirect = next_indirect;
            }
            if (!add && ((offset_cnt == 1) || (offset_cnt == 2 && offsets[1] == 0)))
            {
                block_free(id->pointers[offsets[0]]);
                id->pointers[offsets[0]] = 0;
            }
        }
        else
        {
            block_read(cur_indirect, &cur);
            if (add && cur.pointers[offsets[1]] == 0)
            {
                block_sector_t next_indirect;
                if ((next_indirect = block_allocate()) == 0)
                    return false;
                cur.pointers[offsets[1]] = next_indirect;
                block_write(next_indirect, zeros);
                block_write(cur_indirect, &cur);
                cur_indirect = next_indirect;
            }
            if (!add)
            {
                block_free(cur.pointers[offsets[1]]);
                cur.pointers[offsets[1]] = 0;
                block_write(cur_indirect, &cur);
            }
        }
    }

}

bool inode_resize(struct inode_disk *id, size_t size)
{
    size_t cur_blocks;
    size_t new_blocks;
    off_t cur;
    off_t d_cur;
```

```
    cur_blocks = id->length/BLOCK_SIZE;
    new_blocks = size/BLOCK_SIZE;
    if (new_blocks > cur_blocks)
    {
        //allocate blocks from [cur_blocks+1, new_blocks];
        for (cur = cur_blocks + 1; cur <= new_blocks; cur++)
        {
            if (!inode_change_block(id, cur, true))
            {
                // must deallocate if failed to allocate
                for (d_cur = cur_blocks + 1; d_cur < cur; d_cur++)
                {
                    inode_change_block(id, d_cur, false);
                }
                return false;
            }
        }
        id->length = size;
        return true;
    }
    else if (cur_blocks > new_blocks)
    {
    //deallocate blocks from [new_blocks+1, cur_blocks];
        for (cur = new_blocks + 1; cur <= cur_blocks; cur++)
            inode_change_block(id, d_cur, false);
        id->length = size;
        return true
    }
    else
    {
        id->length = size;
    }
}
```

3.2 Network Layering and Fundamentals

What is the purpose of layering?

> Breaks the problem of network communication into a stack of abstracted modules. Layers can communicate with adjacent layers by understanding a shared protocol/interface.

What are the 5 basic network layers?

> Application, Transport, Network, Data Link, Physical

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

> Transport layer. A reliable protocol is TCP.

What is the end-to-end principle?

> The principle states that some parts of network functionality (such as reliability and security) should be kept in the end-hosts, and not within the network itself.

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)

...

```
SEND: PUT(I LOVE)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
SEND: PUT(OPERATING SYSTEMS)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
SEND: DEL(I LOVE)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
SEND: DEL(I LOVE)
RECEIVE: VOTE-ABORT
SEND: GLOBAL-ABORT
RECEIVE: ACK
SEND: PUT(GOBEARS)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
```

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

```
PUT(I LOVE)
COMMIT
PUT(OPERATING SYSTEMS)
COMMIT
DEL(I LOVE)
COMMIT
DEL(I LOVE)
ABORT
PUT(GOBEARS)
COMMIT
* notice that there are no read commands (i.e. GET) in this toy example,
but there would be no need to commit reads anyways as they do not modify
the state of the server.
```