Section 11: File Systems, Journaling

CS 162

November 13, 2020

Contents

1 Vocabulary

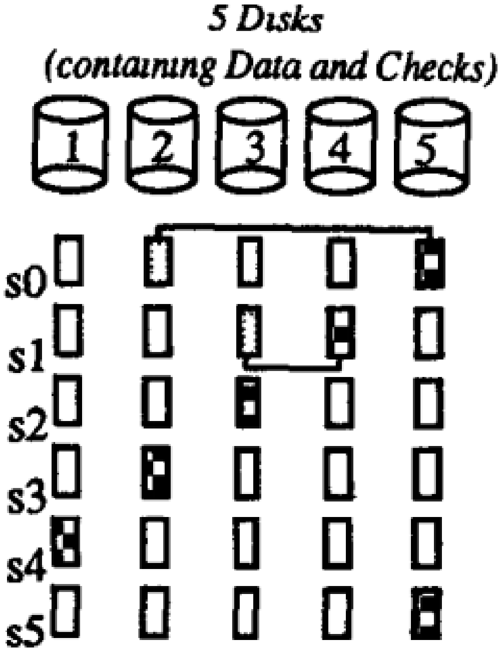
- **Fault Tolerance** The ability to preserve certain properties of a system in the face of failure of a component, machine, or data center. Typical properties include consistencies, availability, and persistence.
- **Transaction** - A transaction is a unit of work within a database management system. 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 can be repeated without an effect after the first iteration.
- **Log** - An append only, sequential data structure.
- **Checkpoint** - Aka a snapshot. An operation which involves marshaling the system's state. A checkpoint should encapsulate all information about the state of the system without looking at previous updates.
- **Write Ahead Logging (WAL)** - A common design pattern for fault tolerance involves writing updates to a system's state to a log, followed by a commit message. When the system is started it loads an initial state (or snapshot), then applies the updates in the log which are followed by a commit message.
- **Serializable** - A property of transactions which requires that there exists an order in which multiple transactions can be run sequentially to produce the same result. Serializability implies isolation.
- **ARIES** - A logging/recovery algorithm which stands for: Algorithms for Recovery and Isolation Exploiting Semantics. ARIES is characterized by a 3 step algorithm: Analysis, Redo, then Undo. Upon recovery from failure, ARIES guarantees a system will remain in a consistent state.
- **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 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.
- **Metadata Logging** - A technique in which only metadata is written to the log rather than writing the entire update to the log. Modern file systems use this technique to avoid duplicating all file system updates.
- **EXT4** - A modern file system primarily used with Linux. It features an FFS style inode structure and metadata journaling.
- **Log Structured File System** - A file system backed entirely by a log.

- **RAID** - A system consisting of a Redundant Array of Inexpensive Disks invented by Patterson, Gibson, and Katz.

  The fundamental thesis of RAID is that in most common use cases, it is cheaper and more effective to redundantly store data on cheap disks, than to use/engineer high performance/durable disks.

- **RAID I** - Full disk replication. With RAID I two identical copies of all data is stored. If disk heads are not fully synchronized, this can decrease write performance, but increase read performance.

- **RAID V+** - Striping with error correction. In RAID V, 4 sequential block writes are placed on separate disks, then a 5th parity block is written by XORing the data blocks on the same stripe. RAID VI uses the EVENODD scheme to encode error correction. In general, Reed Solomon coding can be used for an arbitrary number of error correcting disks.



  Note: Due to the large size of disks in practice, RAID V is no longer used in practice, because it is too likely that a second disk will fail while the first is recovering. RAID VI is usually combined with other error recovery techniques in practice.

- **Eventual Consistency** - A weaker form of a consistency guarantee. If a system is eventually consistent, it will converge to a consistent state over time.

2 Extending an inode (from last week)

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. For this question, assume unlimited disk space.
- "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.
  for (int i = 0; i < 12; i++) { // Handle direct pointers
    if (size <= 512 * i && id->direct[i] != 0) { // Shrink inode
      block_free(id->direct[i]);
      id->direct[i] = 0;
    }
    if (size > 512 * i && id->direct[i] == 0) { // Grow inode
      id->direct[i] = allocate_block();
    }
  }
  if (id->indirect == 0 && size <= 12 * 512) { // Check to handle indirects
    id->length = size;
    return true;
  }
  block_sector_t buffer[128];
  memset(buffer, 0, 512);
  if (id->indirect == 0) { // Allocate indirect page if does not exist
    id->indirect = allocate_block();
  } else {
    block_read(id->indirect, buffer);
  }
  for (int i = 0; i < 128; i++) { // Handle indirect pointers
    if (size <= (12 + i) * 512 && buffer[i] != 0) { // Shrink inode
      block_free(buffer[i]);
      buffer[i] = 0;
    }
    if (size > (12 + i) * 512 && buffer[i] == 0) { // Grow inode
      buffer[i] = allocate_block();
    }
  }
  if (id->indirect != 0 && size <= 12 * 512) {
    block_free(id->indirect);
    id->indirect = 0;
  } else {
    block_write(id->indirect, buffer);
  }
  id->length = size;
  return true;
}
```

Solution that includes error handling (limited disk space)

```
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 = allocate_block();
      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 = allocate_block();
    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 = allocate_block();
      if (sector == 0) { // Handle failure
        inode_resize(id, id->length);
        return false;
      }
      buffer[i] = sector;
    }
  }
  if (id->indirect != 0 && size <= 12 * 512) {
    block_free(id->indirect);
    id->indirect = 0;
  } else {
    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 Comparison of File Allocation Strategies

In lecture three file allocation strategies were discussed: (a) Indexed files. (b) Linked files. (c) Contiguous (extent-based) allocation.

Each of these strategies has advantages and disadvantages, which depend on the goals of the file system and the expected file access patterns. For each of the following situations, rank the three designs in order of best to worst. Give a reason for your ranking.

1. You have a file system where the most important criteria is the performance of sequential access to very large files.

1. c (extent-based)
2. b (linked)
3. a (indexed)

It is easy to see that (c) is the best structure for sequential access to very large files, since in (c) files are contiguously allocated and the next block to read is physically the next on the disk. No seek time to find the next block, and each block will be read sequentially as the disk head moves.

Both (b) and (a) require some look up operation in order to know where the next block is. However, (a) may be slightly more expensive, since for ”very large files”, multiple disk accesses are required to read the indirect blocks.

2. You have a file system where the most important criteria is the performance of random access to very large files.

1. c (extent-based)
2. a (indexed)
3. b (linked)

(c) is still the best structure here: just need to use an offset.

(a) will probably need to look at some levels of indirect blocks in order to find the right block to access (we are dealing with very large files).

(b) is absolutely the worst structure. In fact, in order to find a random block, we will need to traverse a linked list of blocks, which will take a time linear in the offset size.

3. You have a file system where the most important criteria is the utilization of the disk capacity (i.e. getting the most file bytes on the disk).

1. b (linked)
2. a (indexed)
3. c (extent-based)

(c) can suffer heavily of external fragmentation, especially for large files. So it is not the best structure for getting the most bytes on the disk, since lots of space will be wasted. However, for small files and large block size, (c) might prove to be better than (a) and (b).

(a) and (b) stuctures are generally more suitable for this question. The metadata overhead for (b) is likely to be smaller than the one for (a), since it only needs pointers to the next allocated block rather than an entire block (or blocks) which may or may not be totally used.

4 Logs and 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.

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., its 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, so we know these operations have not yet been applied to the file system.

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.

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.

What is the purpose of the *Commit* entries in the log?

- The *Commit* entry makes the creation of each file *atomic*. These changes to the file system's on-disk structures are either completely applied or not applied at all.
- The creation of a file involves multiple steps (allocating data blocks, setting up the inode, etc.) 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 (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.

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 (e.g., updating an inode or a directory 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.