Reading list:

File system chapter form xv6 book.
Disk driver from chapter-3 of xv6 book.
Disk scheduling algorithm from Chapter 12 (12.4) of Silberschatz and Glavin book.

Every file/directory metadata is stored in an inode. Before reading/writing to a file, xv6 acquires a lock in the in-memory inode for synchronized access.

As we already know that reading/writing to the disk is very expensive as compared to memory read/write. An in-memory buffer cache (managed by OS) holds a copy of recently accessed disk blocks to eliminate the need to read the data multiple times from the disk.

There are two ways to implement buffer cache: write through and write back

A write-through cache always writes data to the disk, whenever a cached buffer is modified. xv6 provides bwrite interface to persist data on the disk. A write-back cache asynchronously syncs the data with the disk (faster than write-through).

A buffer cache is a circular list of buffers. Each node contains a lock (using busy flag), disk block number, and the data of the disk block. The busy flag ensures that only one thread can access the buffer at a given time.

You might wonder why locking at buffer cache granularity is needed when we already have synchronization at the file granularity using inode locks. The reason behind this is multiple indoes can live on a single disk block. If multiple threads update different inodes, on the same block, concurrently and one thread persist the block to the disk, then it is possible that a partially updated inode is written to the disk. If a crash happens before a fully updated inode is written to the disk, the file system may see an inconsistent inode after reboot. For this reason, only one thread is allowed to update a buffer cache block at a given time.

Another important job of an OS is recovery after a crash. A crash (say due to power failure) should not leave the file system in an inconsistent state. For example: let us see the steps of file creation:

1. allocate inode for file
2. add an entry to the directory for that inode
3. initialize an inode

Notice, that a power failure after step-2 will leave a dangling pointer (pointer to an uninitialized inode) in the directory. This is very bad because the filesystem may overwrite some critical files in your system due to garbage value in the uninitialized inode.

The consequences of a crash are not so severe if you reorder the operations during file creation. Let us say the order followed during creation is:

1. allocate inode for file
2. initialize an inode
3. add an entry to the directory for that inode

Notice that in this case, a power failure after step 1 or 2 is not so bad, because nobody is pointing to the newly created inode. It is a space leak.

Similarly, a dangling reference problem may occur during file deletion. Let us say the steps during deletion are:

1. free data blocks corresponding to file which is going to be deleted
2. free inode of the file
3. remove reference from the parent directory

In this case, a power failure after step 1 will leave the dangling references in the inode. A power failure after step 2 will leave the dangling references in the directory.

We can avoid dangling references by reordering the steps.

1. remove reference from the parent directory
2. free inode of file
3. free data blocks corresponding to file which is going to be deleted

In general, there can be two types of dangling references.

1. inode points to free blocks
2. directory points to (free/uninitialized) inode

We can always avoid dangling pointers by following the rules as described below.

1. Always initialize the block before creating references to it
2. Always erase the references before freeing a block

The above rules will not solve the problem of space leak. A separate program (e.g., “fsck” on Linux) can be used to free the leaked space. fsck walks the entire file system and free unreferenced blocks.

However, fsck cannot fix all inconsistencies. For example: mv a/b c/d can be implemented as follows:

1. remove b from a
2. add d to c

A power failure after step 1 will delete the file a. fsck cannot fix this. Alternatively, the same command can be implemented using:

1. add d to c
2. remove b from a

A power failure after step 1 will leave two files b and d, which are aliases of each other. Certainly, fsck won’t fix this because this is also a valid file system state.

xv6 and Linux implement logging to fix all type of inconsistencies due to crash.

The high-level idea is to provide interfaces to users for declaring the sequence of writes that need to be atomic, save all the writes in a temporary space on disk, and then finally write them to the original disk location. If a power failure happens during the partial update to the original disk locations, then the data can be stored from the temporary copy after reboot.

In xv6, the programmers can declare atomic operation using begin_op and end_op interface. For example:

```
mv a/b c/d
begin_op()
    1. remove b from a
    2. add d to c
end_op()
```

The above code will ensure that after recovering from a crash, the file system will either see a/b or c/d.

Multiple atomic operations can execute in parallel. xv6 does a group commit. During group commit the data is written first to the log space and then to the original locations.

Notice that an operation cannot execute concurrently with group commit, because the commit may persist the partial changes made by a concurrent operation to the disk. xv6 does not allow concurrent operations during group commit. However, Linux ext3 file system allows concurrent operations. To allow concurrent operations, the committer (commit code) must,

1. disallow new operations
2. wait for existing operations to finish
3. make a copy of dirty blocks
4. allow new operations
5. write copied data (in step 3) to the log file
6. write log header
7. copy data from the log file to the original disk locations

Concurrent group commit is only doing memory to memory copy when the concurrent operations are disallowed. So, the concurrent operations need not wait for a very long period. The cost of copying can also be reduced using copy on write optimization.

ext3 periodically (say 5 secs) commits. The problem with periodic commit is the application is notified about the disk write even though the write is still pending. There are two ways of thinking about this:

1. The application knows that data has been written
2. The user knows that the data has been written

ext3 has this nice observation that we need not need to worry about persistence unless the user has been notified about it. The common ways of communicating with user are: through console or through network. ext3 system never externalizes data (writing to the console or network), if it depends on some uncommitted data. In this case, ext3 waits for commit before notifying the user to ensure consistency. In this approach, the user may notice arbitrary latency, but the overall throughput of the system is good.