Filesystem layout

Draw diagram of disk:

- Boot sector (sector 0)
- Superblock (sector 1)
- Inodes (sector 2 to ninodes/IPB+3)
- Free-block bitmap (one bit per data block indicating free/used)
- Blocks

File Creation

```
$ echo > a

1. allocate an inode (write to inode region).
2. update the size/status of the inode (write to inode region).
3. add a directory-entry record to the parent directory's data blocks. (write
to data block region)
4. update parent directory's inode (size)
```

File Write/Append

```
$ echo x > a

1. allocate a block (write to block bitmap region)
2. zero out block contents (write to data block region)
3. write "x" to block (write to data block region)
4. update "a"'s inode (size, addrs)
```

File Unlink

```
$ rm a

1. write zero to "a"'s directory-entry record in parent directory's
  data blocks (write to data block region)
2. update parent directory's inode (size)
3. update "a"'s inode to mark it free
4. update free block bitmap to mark data blocks free
```

Notice that each operation involves 4-8 disk writes and these filesystem accesses are to different regions of disk, i.e., they involve multiple seeks/rotations. Notice that we are assuming a write-through cache. Also, the number of outstanding I/O's (in-flight I/O's for the disk controller) is relatively small (one?) so this is not very efficient usage of the disk.

Synchronization for FS Accesses

What happens if concurrent accesses to the filesystem are performed by multiple threads? For example, bad things can happen if two threads try to create files within the same directory concurrently.

Solution 1: Use a coarse-grained lock for the entire filesystem.
This is bad because it serializes concurrent accesses to different files. Given that file accesses are anyways very slow, this can be a huge performance problem. Moreover, it restricts the number of outstanding I/Os to a very small number (typically, one), as only one thread could be executing within the filesystem at any time.

Question: does the coarse-grained global filesystem lock need to be stored on-disk or does an in-memory lock suffice? Answer: In-memory lock suffices, as all threads need to access the disk through the OS interfaces which can synchronize using the in-memory lock.

Solution 2: Use fine-grained locks --- in particular, use a lock per block. Use the buffer cache to synchronize. Mandate that a block can only be accessed by first bringing it into the buffer cache. Then use a lock per buffer in the buffer cache. (recall that a buffer corresponds to a disk block). The locks need to be blocking locks, as critical sections are likely to be very large (disk accesses).

xv6 implements per-buffer locks by using a BUSY bit per buffer. Accesses to the BUSY bit are protected by a global buffer cache spinlock, `bcache.lock`. Thus, to access a block:

1. First the global `bcache.lock` is acquired.
2. The block being accessed is searched in the buffer cache.
3. If found and !BUSY,
    - Mark it busy
    - Release `bcache.lock`
    - Return buffer contents
4. If found and BUSY,

    - `sleep(buffer address, &bcache.lock)`. This will release `bcache.lock`. A thread that is currently working on this buffer (and has thus marked it BUSY) should call wakeup after clearing the BUSY bit.
    - On waking up, the thread should rescan/re-search the buffer cache for the desired block at step 2 (as it may have been replaced after the `bcache.lock` was released).
5. If not found
    - Find a replacement buffer that is not currently BUSY (this ensures that a busy buffer cannot be evicted).
    - Mark it busy
    - Release `bcache.lock`. (notice that I now hold a fine-grained lock on the desired buffer, and so do not need `bcache.lock`)
    - Write replacement buffer to disk if dirty
    - Read desired block into this buffer from disk
    - Return buffer contents

   Notice that all disk operations are done without holding `bcache.lock`. Instead disk operations are protected per-buffer locks (BUSY bits).

Notice that the global buffer cache lock is used only to manipulate the busy bits of the individual buffers. The busy bits act as fine-grained locks.

But doesn't the busy bit protect only against concurrent accesses to a single block. What about operations that require accesses to multiple blocks. How are they made atomic?

Simple: mark all the blocks (on which an atomic operation needs to be performed) busy. (Just like taking multiple locks). But . . . we know that fine-grained locks have deadlocks. So, need an order on the setting of busy bits for buffers. Here are some global invariants followed by xv6 (check on your own):

- Always mark the superblock busy first (if it needs to be involved in the atomic operation).
- Always mark inode blocks before data blocks
- Always mark parent (dir)'s inode before child (dir)'s inode
- Never mark any other block if one of the bitmap blocks is marked

xv6 implements LRU for buffer cache replacement.

- Maintain the buffers in a doubly-linked list.
- Each time you are done with accessing a buffer (at the time of clearing the busy bit), move the buffer to the front of the buffer cache list (see `brelse`).
- Start replacement at the last entry of the list.

Notice that because xv6 uses a write-through buffer cache, many consistency problems become easy. We will next look at the consistency issues that can arise due to a write-back buffer cache.

Recall that concurrent accesses to the disk are synchronized by `idelock`. Also, recall that the xv6 IDE device driver allows only one outstanding disk request at any time (the front of the idequeue). Clearly, this can be improved so that there are multiple outstanding disk requests and the disk device can schedule them efficiently (recall elevator algorithm).

How fast can an xv6 application read big files? If the file has contiguous blocks? xv6 has no prefetching, it will wait for the first block to return before it issues command for second block. In doing so, we waste a full rotation of the disk! A better approach would be to prefetch, or to allow multiple outstanding requests to the disk device.

Q: Why does it make sense to have a double copy of I/O? First we read data from disk to buffer cache. Then, from buffer cache to user space. Can we do better? e.g., pass user-space buffer to the disk device driver? Few issues:

- Need to lock that page in memory (can't be swapped out)
- What if the user process accesses it in the middle. May be okay, actually.
- But, the buffer will not be available to other processes -- no caching! may be okay for large scans, but not okay for workloads exhibiting locality across processes.

Q: how much RAM should we dedicate to disk buffers? Tradeoff: if too big buffer cache, less space for virtual memory pages; and vice-versa. Can be adapted at runtime based on usage patterns.

Crash Recovery

A filesystem operation requires multiple disk accesses. What happens if power fails in the middle of a filesystem operation. It can leave the filesystem in an inconsistent state.

Example: file creation involves two main steps (in terms of disk writes)

1. Initialization/allocation of inode
2. Creating entry for it in the parent directory

If at power failure time, the entry for the new inode has been created in the parent directory, but the inode has not been allocated, then we have a

case of a *dangling pointer* in our filesystem tree. This can be a major problem, as the uninitialized inode may contain junk data (or worse private data of another user) which may confuse future accesses to this part of the filesystem.

On the other hand, if at power failure time, the opposite is true, i.e., the inode has been allocated/initialized but its entry has not been created in the parent directory, then there is no problem of a dangling pointer in the filesystem tree. In this case, we have a *space leak* because an inode has been marked allocated but it is not pointed-to by the filesystem tree, and so it will never be used.

A space-leak is more acceptable than a dangling pointer. Using this observation, a filesystem can enforce an order on the disk writes. The order should ensure that there will never be any dangling pointers in the filesystem tree. In this file creation example, this means that an inode should be initialized before an entry is created for it in its parent directory. Similarly at file unlink time, the entry must be removed from the parent directory before deallocating the inode (if done in the opposite order, dangling pointers can result).