CS 333 :: Meeting Notes :: File System Implementation

Basic Organization

The disk (or other storage device) is divided into a logical array of blocks, and a *file system* takes ownership of some *partition* of the disk (Partitioning a disk is a fun activity that you might want to try some time. Partitioning essentially divides the LBA space of your drive into independent sections that each look like their own "disk". This is why you might see entries for `/dev/sda1`, `/dev/sda2`, ..., `/dev/sdaN`: each `/dev/sda_i_` is a separate partition of the same physical disk). A file system uses those blocks to store both *metadata* (data structures that store information about data, such as **super blocks** and **inodes**) and *data* (actual file contents). Efficiently placing data/metadata, enforcing data/metadata permissions, and preserving data/metadata consistency in the presence of system crashes are three key file system responsibilities.

Data Structures

Any file system will rely on a handful of key data structures. However the Linux Virtual File System (VFS) has standardized several of the most common data structures (in fact, your file system *must* utilize some form of these data structures in any reasonable implementation; at minimum, it must implement fake data structure values to satisfy the VFS functions that define FS behavior. Mapping a new FS design onto the well-structured VFS framework can be a challenge for innovative designs; we will revisit this later when we talk about full-path-indexed file systems).

- **inode**: persistent metadata structure that holds information about a single on-disk file, including the file's permissions, size, and the location of the file's contents (spread amongst potentially many disk blocks)
  - inode numbers are unique (e.g., an index into a table, a monotonically increasing counter, etc.)
  - inodes are *reference counted*; when the link count reaches zero the file is freed (recall the `link` / `unlink` system calls)
  - As stated in the book, the **i** could stand for index (i.e., index node), but I assert that the **i** would be more descriptive if it stood for *indirection*: names are mapped to inodes by directory files (recall that directories are often implemented as special files that store a set of {name, inum} pairs), and inodes contain a mapping to the file's contents (the LBAs of its data blocks).
  - common inode designs efficiently support small files, but grow to accommodate large files as well.
    - In FFS-inspired file systems, inodes use a combination of *direct*, *indirect*, *double indirect*, *triple indirect*, etc. blocks to grow as needed.
    - Other file systems use *extents* to track allocations. An extent describes an allocation using two values: the starting block and the extent size (in blocks). For files whose blocks have good locality, an extent is a very efficient representation.
- **super block**: contains overall information about the file system. You should look at the superblock fields and have a high level understanding of their role:
  - magic number: a special byte that is unique to this file system and serves as a "fingerprint". You can compare a known magic number against the value stored in a superblock to verify that the superblock does in fact describe the file system type you expect (e.g., ext4, zfs, btrfs). This magic number check happens during `mount`.
- **allocation structures** (e.g., free list, bitmap, extent list). Allocation structures may be used to track free data blocks, metadata structures, or any other resource needed by the file system. In the simple file system described in the text, there is one bitmap that keeps track of which blocks in the data section of the disk are allocated/free.

VFS Data Structures Questions

1. Why do some file systems use a combination of direct & indirect blocks, instead of just using all direct or all indirect blocks?
2. What is the worst case lookup time for a file system that uses a linked-list to represent its data blocks (the FAT file system does this)?
    - This performance sounds (and is) bad, but it might not be as noticeable to end users as you'd think. What are common file access patterns that work tolerably well with this behavior?
    - What types of operations might be made easier or harder when data blocks are represented by a linked list?
3. What is one advantage and one disadvantage of a "block pre-allocation" heuristic for creating new files (i.e., when allocating a new file, `X`, *n* contiguous blocks are reserved for `X`'s data)?
4. We often assume that our hardware works as desired and our data is always read/written reliably, but crashes and corruptions are a reality.
    - Which data structures, if lost, would have catastrophic ramifications on our FS?
    - How might we try to protect ourselves against the inadvertent loss of our most important metadata structures?

Caching

Caching is important for performance. The file system layer's shared cache, often called the *page cache* or *buffer cache*, stores in-memory copies of file contents. Data may be cached before being written to disk. Metadata may also be stored in a cache, often separately from the data cache.

1. Why is a static partitioning of RAM for the file system's page cache a not-so-great idea (e.g., allocating exactly 2GiB for your page cache)? What do most systems do instead?
2. What is meant by the term **direct I/O**, and in what situations might it be desirable or undesirable to perform direct I/O?
3. What order should you write objects in the cache: data first or metadata first? Why?

Opening, reading, and writing files

Let's assume that we start with a "cold cache". This means that no data structures (other than the superblock) are resident in memory; thus, the first access to any block requires an I/O.

1. What data structures (and fields) must be accessed to satisfy the command: `fd = open("/home/bill/file.txt", O_RDWR);` ?

2. In the Linux kernel, path lookup is a very complicated piece of code! What challenges might the presence of symbolic links add to the pathname resolution process?

3. Once the file is open (suppose `fd=7`), what data structures (and fields) must be accessed in order to satisfy the command: `n = pread(fd, buf, 32, 1024);` ?

4. What data structures (and fields) must be accessed to satisfy the command: `n = pwrite(fd, buf, 32, 1024);` ?

5. What happens when you execute: `fsync(fd)` ?

Simple File System Design

The simple file system design presented in chapter 40 makes some design choices that serve as a useful starting point when exploring the file system design space, but the choices add some limitations that a scalable file system would want to avoid. Let's examine the implications of some of those assumptions.

The disk is divided into two regions: a metadata region followed by a data region. This is not unreasonable, but let's think about the implications for the system.

1. What about this design choice, if anything, imposes a maximum file size?
2. What about this design choice, if anything, imposes a maximum number of files?
3. What is the maximum distance that a file's metadata could be from the data that it describes?

The metadata region has a fixed-size inode table filled with fixed-sized inodes that are each stored at a formulaic location (i.e., given an inode number, there is exactly one place the inode could be stored and that location can be calculated using a well-defined formula).

1. What operations does this speed up (*hint*: are there common FS tasks that we can solve with simple math)?
2. Does a static inode add any scalability limitations (think about data AND metadata scalability)?

For performance reasons, we often care a lot about locality.

1. How does the concept of "the inode" (in general) preserve/harm the locality of metadata?
    - Hint: think about locality in a broad sense. We can think of inter-file locality, intra-file locality, and free-space locality.
2. How does the "inode design" in the simple file system (a static inode table size/location) preserve/harm the locality of metadata?
3. How does the design of the simple file system preserve/harm the locality of file data?
4. Is there any relationship between the location of metadata and the location of data described by that metadata in this design?
    - What does this mean for the access patterns of different operations? Think about how many seek operations are required (in the worst case) when:
        - opening a file
        - reading data from an open file
        - writing data to an open file
        - syncing dirty data in a file