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*` 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 3 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, you *have* to implement some form of these in any reasonable file system, or at least fake them in order to be able to implement the VFS hooks that define FS behavior. We will go into this in more detail when we talk about the file system *BetrFS*).

- **inode**: metadata structure that holds information about a single on-disk file, including 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, and 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).
  - inode designs must 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)
- **super block**: contains overall information about the file system. You should look at the superblock fields and have a high level understanding of what they are/do.
  - 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 super block to verify that it is in fact the file system you expect it to be (e.g., ext4 not zfs). 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 a bitmap that keeps track of which blocks in the data section of the disk are allocated/free.

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 of data blocks (like FAT)? Why might this not be quite as bad as you'd think (hint: what are common file access patterns)?
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)?

Caching

Caching is important for performance. The file system's 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 (somewhat) bad idea? What do most systems do instead?
2. What is meant by the term direct I/O, and why might it be desirable or undesirable?
3. What order should you write objects in the cache: data first or metadata first? Why?

Opening, reading, and writing files

Let's assume a cold cache. This means that no data structures (other than the superblock) are resident in memory; 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 pathname resolution?
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 it does add some limits to the system.

1. Maximum file size?
2. Maximum number of files?

The metadata region has a fixed-size inode table filled with fixed-sized inodes stored at a fixed location.

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 inodes (in general) preserve/harm the locality of metadata?
2. How does the design of inodes 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 in this design? * What does this mean for the access patterns when opening/reading/writing/syncing files?

Hand-in question

A file system designer might decide to exclusively use direct pointers to implement its inode data structure (recall that an inode stores information about the location of the blocks in a file. When using direct pointers, the inode tracks a file's blocks as an array of block offsets.)

1. Briefly describe the main limitation of this approach (a sentence or two should be more than enough).
2. Describe one alternative to using direct pointers and describe how that alternative addresses the above limitation.