Discussion 8: Queueing Theory, File Systems

November 8, 2023

Contents

1 File Systems

A **file system** provides persistent, named data to the user. It is an abstraction layer maintained by the operating system to make it easy for users and programs to interact with files.

A **file** is a named collection of data in a file system. Files allow an arbitrary amount of data to be referred to by a single, meaningful name. A file is composed of two parts: metadata and data. The **metadata** contains information about a file needed by the operating system such as size, owner, and access control. The **data** is the actual content of the file that the user or program puts in. While this data can be interpreted as something meaningful using programs (e.g. text editors, PDF readers), the file system will view the data as simply a raw sequence of bytes.

A **directory** is a list of mappings from human readable file names to a specific underlying file or directory. When given a **path** to a file or directory, the process is a recursive procedure of reading directories and finding the correct mapping. Many file systems have support for two types of directory entries: hard and soft links. **Hard links** are directory entries that map different names to the same file number. This allows for one underlying file to have many different names. Hard links can't span across multiple file systems since file numbers will differ across different file systems. They can be created using `ln [target] [dest]` command. On the other hand, **soft links**, also known as **symbolic links**, are directory entries that map a name to another name. These are commonly known as shortcuts. Since path names can be the same across multiple file systems, soft links can move across different file systems. They can be created using `ln -s [target] [dest]` command (recall your project repo pre-commit hook).

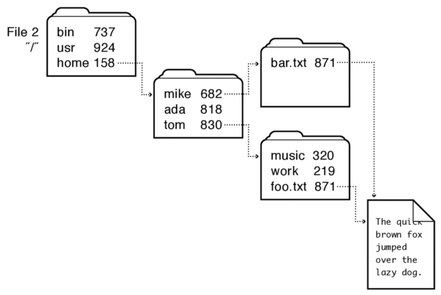

Figure 1: Hard link

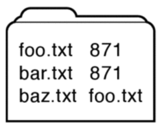

Figure 2: Soft link

Throughout the file system unit, we will examine several file system implementations. The main components to focus are the index structure, directory structure, and free space tracking.

Fast File System (FFS)

The **Berkeley Fast File System (FFS)**, also called the **BSD Fast File System**, emphasizes the importance of accessing a file's blocks quickly while also maintaining good disk performance. It maintains good performance for both small and large files.

Index Structure

FFS uses a fixed, asymmetric tree called a **multi-level index**. Each index is rooted at an **inode** that stores the file's metadata and *pointers* to data. It's important to note that an inode does not store the file's name or any actual data in the inode itself. Inodes are stored in a fixed block on disk in an **inode array**.

Pointers come in different forms, but they're just block numbers. **Direct pointers** point directly to the block, meaning the pointer is a block number of a block that contains the file's data. An **indirect pointer** points to an **indirect block** which is an array of direct pointers. An indirect block is just a block (i.e. raw bytes) which we just interpret as an array of unsigned integers; there is nothing special about an indirect block compared to any other block on disk. Similarly, a **double indirect pointer** points to a **double indirect block** which contains an array of indirect pointers. The levels of indirect pointer continue (i.e. triple, quadruple), but you will typically see up to triple indirect pointers.

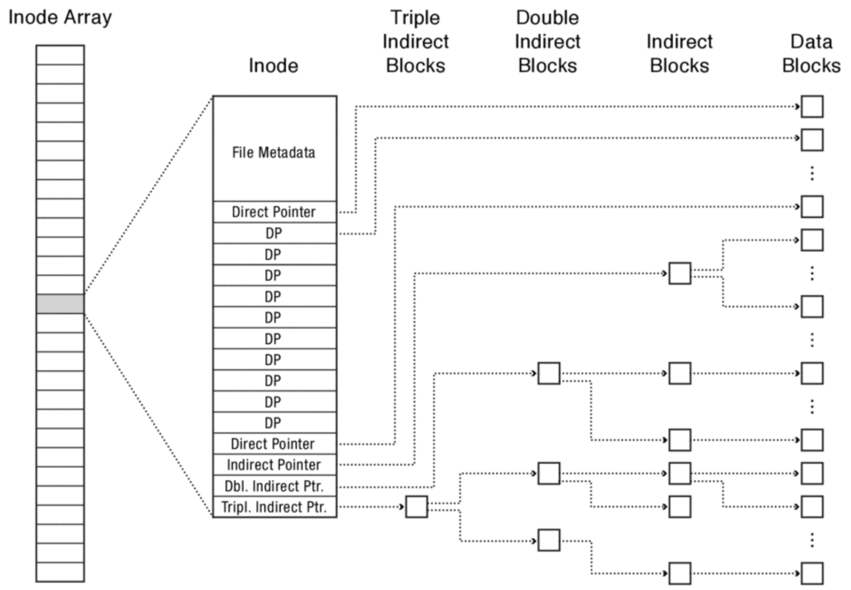

Figure 3: Inode

Directory Structure

The index of an inode in this inode array is called the **inumber**, which FFS uses as the file number as well. Directories will map a file name to an inumber. The metadata of a file is now stored in the inode itself, not the directory entry. As a result, FFS supports hard links.

Free Space Management

To track free space, FFS allocates a bitmap, where each bit indicates whether the block corresponding to that bit is free. This bitmap is stored on a fixed place in disk. Unlike FAT, FFS will optimize disk performance using **block group placement** which places a file's inode block and data block near each other. Moreover, it will reserve a fraction of the disk's space (e.g. 10%), showing a reduced capacity to the user. This is important because the block group placement relies on there being enough free space on the disk.

1.1 Concept Check

1. Consider an inode-based file system with 2 KiB blocks and 32-bit disk and file block pointers. Each inode has 12 direct pointers, an indirect pointer, a double indirect pointer, and a triple indirect pointer. How large of a disk can this file system support? What is the maximum file size?

A 32-bit disk pointer means there can be at most 2^{32} blocks, so the maximum disk size this file system can support is

$$2^{32} \times 2^{11} = 2^{43} = 8 \text{ TiB}$$

The maximum file size is the block size multiplied by the number of all direct pointers (i.e. including the ones from the indirect pointers). We see that there are $2048/4 = 512$ pointers per block. Therefore, the maximum file size is

$$\begin{aligned} 2048 \times (12 + 512 + 512^2 + 512^3) &= 2^{11} \times (2^2 \times 3 + 2^9 + 2^{9\times 2} + 2^{9\times 3}) \\ &= 2^{13} \times 3 + 2^{20} + 2^{29} + 2^{38} \\ &= 24 \text{ KiB} + 513 \text{ MiB} + 256 \text{ GiB} \end{aligned}$$

The last term dominates here, so the maximum file size is essentially 256 GiB.

2. Compare bitmap-based allocation of blocks on disk with a free block list.

Bitmap based block allocation is a fixed size proportional to the size of the disk. This means wasted space when the disk is full. A free block list shrinks as space is used up, so when the disk is full, the size of the free block list is tiny. However, contiguous allocation is easier to perform with a bitmap. Most modern file systems use a free block bitmap, not a free block list.

3. For inode-based file systems, what is the point of having direct pointers? Why not just have indirect pointers since they can point to much more data than a single direct pointer?

As seen from the empirical study of file sizes, most files are small enough to be contained within the given direct pointers of an inode-based file system. Direct pointers are faster compared to indirect pointers since they need to read in less blocks to get to the underlying data block.

4. List the set of disk blocks that must be read into memory in order to read the file `/home/cs162/pintos.bean` in its entirety from a UNIX BSD 4.2 file system (10 direct pointers, an indirect pointer, a double indirect pointer, and a triple indirect pointer) on a magnetic disk with 1 KiB block size. The `/home/cs162/pintos.bean` file is 15,234 bytes. Assume that the directories in question all fit into a single disk block each and the inode array is resident in memory (i.e. don't count inode array disk accesses).

1. Read in inode for root directory.
2. Read in block pointed to by the the first direct pointer for root inode.
3. Read in inode for `home` directory.
4. Read in block pointed to by the the first direct pointer for `home` inode.
5. Read in inode for `cs162` directory.
6. Read in block pointed to by the first direct pointer for `cs162` inode.
7. Read in inode for `pintos.bean` file.

8-17. Read in each block pointed to by the each direct pointer for `pintos.bean` inode. This will read through $1024 \times 10 = 10,240$ bytes of the file.

18. Since we still have $15,234 - 10,240 = 4,994$ bytes remaining, we need to read in the indirect pointer for `pintos.bean` inode.

19-23. Read in direct blocks corresponding to the first five direct pointers in the indirect block read in from the previous step. Five direct pointers points to $1024 \times 5 = 5120$ bytes, so the fifth block read in will only be partially full.