Chap 40: File System Implementation

**Very Simple File System** **(vsfs)** is a simplified version of a typical UNIX file system and thus serves to introduce some of the basic on-disk structures, access methods, and various policies that you will find in many file systems today.

**40.1 The Way To Think**

We will consider two aspects. The first is the **data structures** of the file system. In other words, what types of on-disk structures are utilized by the file system to organize its data and metadata? The second aspect of a file system is its **access methods**. How does it map the calls made by a process, such as open(), read(), write(), etc., onto its structures?

**40.2 Overall Organization**

The first thing we’ll need to do is divide the disk into **blocks**. Simple file systems use just one block size, and that’s exactly what we’ll do here. Let’s choose a commonly used size of 4 KB. Lets say that there are N blocks, from 0 to N-1, and there are only 64 blocks.

To store these blocks to build a file system, we first consider user data. Most of the space in any file system is user data. Lets call this region **data region** and let this region take 56 blocks:

Text

Description automatically generated

The file system has to track information about each file. This information is a key piece of **metadata**, and tracks things like which data blocks (in the data region) comprise a file, the size of the file, its owner and access rights, access and modify times, and other similar kinds of information. To store this information, the file systems usually have a structure called an **inode**.

To accommodate inodes, we’ll need to reserve some space on the disk for them as well. Let’s call this portion of the disk the **inode table**, which simply holds an array of on-disk inodes. We will use 5 blocks for inodes:

Text

Description automatically generated

The inodes are not big, about 128 or 256 bytes.

We also need a way to track whether inodes or data blocks are free or allocated, or **allocation structures**. We can use a **free list**, but in this case, we will use a structure called **bitmap**, one for the data region (the **data bitmap**), and one for the inode table (the **inode bitmap**). Bitmap is a simple structure to indicate if a corresponding block/object is free(0) or in-use(1). Therefore, with an inode bitmap (i) and data bitmap(d), we have:

Text

Description automatically generated

The one block left is served as **superblock**, which contains information about this particular file system, including, for example, how many inodes and data blocks are in the file system, where the inode table begins and magic number to identify the file system type.

Thus, when mounting a file system, the operating system will read the superblock first, to initialize various parameters, and then attach the volume to the file-system tree. When files within the volume are accessed, the system will thus know exactly where to look for the needed on-disk structures.

**40.3 File Organization: The Inode**

Inode is short for **index node**. Each inode is implicitly referred to by a number (called the **i-number**), which we’ve earlier called the **low-level name** of the file. In vsfs, we should be able to calculate where on the disk the corresponding inode is located based on an i-number. Assuming that each inode is 256 bytes and starting at 12KB. We will have the following inode table:

Table

Description automatically generated

The sector address of the inode block can be calculated as follows:



Inside each inode is virtually all of the information you need about a file, including its type, size, the number of blocks allocated to it, protection information (who owns the file and who can access it), some **time information** and where its data blocks reside in disk. We call all such information as **metadata**.

One of the most important decisions in the design of the inode is how it refers to where data blocks are. One simple approach would be to have one or more direct pointers (disk addresses) inside the inode; each pointer refers to one disk block that belongs to the file.

**The Multi-Level Index**

To support bigger files, file system designers have had to introduce different structures within inodes, which is **indirect pointer**. Instead of pointing to a block that contains user data, it point to a block contains more pointers, each of which point to user data. An inode may have some fixed number of direct pointers (e.g., 12), and a single indirect pointer. If a file is large enough, an indirect pointer is allocated and the inode’s slot for an indirect pointer is set to point to it.

To support larger files, we simply add another pointer to the inode (**double indirect pointer**). The larger the file, the more indirect pointer.

Overall, this imbalanced tree is referred to as the **multi-level index** approach to pointing to file blocks.

This imbalance makes sense since in reality, most files are small, so we optimize for this case.

**40.4 Directory Organization**

In vsfs, directories have a simple organization, consisting of a list of (entry name, inode number) pairs. For each file or directory in a given directory, there is a string and a number in the data block(s) of the directory. For each string, there may also be a length. For example, consider a directory having 3 files:

Table

Description automatically generated

Each file has its inode number, record length (number of bytes for the name plus any left-over space), string length (actual length of the name) and the name.

Deleting a file (call unlink()) can leave an empty space in the middle of directory, so there will be some way of marking it. Such a delete is one reason the record length is used: a new entry may reuse an old, bigger entry and thus have extra space within.

File systems often treat directory as a special type of file. Thus, a directory has an inode.

Note that a linear list of directory entries is not the only way. We can implement in B-tree form, for example.

**40.5 Free Space Management**

A file system must track which inodes and data blocks are free, and which are not, so that when a new file or directory is allocated, it can find space for it. Thus free space management is important for all file systems. In vsfs, we have two simple bitmaps for this task.

When we create a file, we will have to allocate an inode for that file. The file system will look through the bitmap for an inode that is free and allocate it for the file. The file system will mark it as used and eventually update the on-disk bitmap with the correct information.

When we allocate data blocks for a new file, some file systems will look for a sequence of blocks that are free when a new file is created and needs data block. By doing so, the file system guarantees that a portion of the file will be contiguous on the disk, thus improving performance. We call this **pre-allocation policy**.

**40.6 Access Paths: Reading and Writing**

**Reading A File From Disk**

Assuming that we want to open the file bar (/foo/bar) to read it and close. Let’s assume that the file is just 12KB in size.

Table

Description automatically generated

When we issue an open, the file system will find the inode for the file. To do so, the system will traverse the pathname of the file and locate the inode. All traversals begin at the root of the file system, in the root directory which is simply called /. To do so, the inode of the root directory must be declared (most of the time it is 2).

As it locates to bar, the final step is to check the permission, allocate a file descriptor for this process in the per-process open-file table, and return it to the user.

As the file is opened, the system can issue a read() system call to read. The first read will read the first block of the file and consult the inode to find the location of such a block; it may also update the inode with a new last accessed time. The read will further update the in-memory open file table for this file descriptor, updating the file offset such that the next read will read the second file block, etc.

To close the file, the file descriptor is simply deallocated.

Notice that the amount of I/O generated by the open is proportional to the length of the pathname as for each additional directory in the path, we have to read its inode as well as its data.

**Writing A File To Disk**

To write to a file, the file must first be opened. It will write and then close it.

Unlike reading, writing to the file may also allocate a block. When writing out a new file, each write not only has to write data to disk but has to first decide which block to allocate to the file and thus update other structures of the disk accordingly (e.g., the data bitmap and inode). Each write to a file generally takes 5 I/Os: one to read the data bitmap, one to write the bitmap, two more to read and then write the inode and one to write the actual block itself.

The amount of write traffic is even worse when one considers a simple and common operation such as file creation. To create a file, the file system must not only allocate an inode, but also allocate space within the directory containing the new file. The total amount of I/O traffic to do so is quite high: one read to the inode bitmap (to find a free inode), one write to the inode bitmap (to mark it allocated), one write to the new inode itself (to initialize it), one to the data of the directory (to link the high-level name of the file to its inode number), and one read and write to the directory inode to update it. If the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too.

Table

Description automatically generated

**40.7 Caching and Buffering**

To make the I/O faster, file systems introduced a **fixed-size cache** to hold popular blocks. This fixed-size cache would usually be allocated at boot time to be roughly 10% of total memory. However, unused pages in the file cache cannot be re-purposed for some other use, and thus go to waste if the file system does not need 10% of memory at a given time.

Modern systems, in contrast, employ a **dynamic partitioning approach**. Specifically, many modern operating systems integrate virtual memory pages and file system pages into a **unified page cache**.

To cache on write, **write buffering** will have benefits. First, by delaying writes, the file system can **batch** some updates into a smaller set of I/Os. Second, by buffering a number of writes in memory, the system can then schedule the subsequent I/Os and thus increase performance. Finally, some writes are avoided altogether by delaying them (e.g. create a file and then delete it immediately). The trade-offs is that if the system crashes before the updates have been propagated, the updates are lost.

Some applications do not like this trade-off. Thus, to avoid unexpected data loss, they force writes to disk (using fsync()) by using direct I/O interfaces that work around the cache, or by using the raw disk interface and avoiding the file system altogether.