Operating Systems

File systems

By Paul Krzyzanowski
October 25, 2010 [updated March 21, 2011]

> *You can tune a file system, but you can't tune a fish.*
> *— tunefs(8) man page*

Introduction

A file system is a structure on a block device, typically a disk, that provides structured, organized access to data and metadata. We'll be using the term **metadata** quite a lot when discussing file systems. It refers to all the things that describe the data but are not the core file data. For example, given a jpeg image file, the contents that represent the image in jpeg format are the file's data. The length, creation/modification/access times, permissions, owner of the file, and the location of the actual data are all metadata. The file's name is generally not considered to be part of the metadata, as we'll discuss a bit later.

We have several design choices to contend with in implementing a file system:

- Is the **namespace** structured as a hierarchy, as a flat sequence of files, or some other structure? Most systems present a tree-structured hierarchy but other representations are possible. A file system **namespace** is the collection of the names and paths of files and directories.
- Can our operating system support **multiple formats** of file systems? This is highly desirable since we may plug in removable media that is formatted with a different file system (e.g., FAT-32 on a memory card from a digital camera or an ISO9660-formatted CD).
- Can we support **multiple file systems** at the same time? Do they all fit in one namespace? For example, the traditional DOS/Windows approach has been to identify a device and then a filename within the device while the UNIX approach was to allow one to add file systems anywhere onto the tree-structured hierarchy of the existing file system namespace. Having one common name space allows us to ignore the underlying physical composition of our file system.
- Do we need to be able to distinguish **types of files** or do we just treat them as byte streams? IBM mainframe systems, for instance, supported record-based indexed files (both fixed and variable-size) while the UNIX approach was that every file is just a byte stream.
- Metadata is the collection of attributes about a file. **What attributes should a file have?** Some are obvious, such as the owner, length, creation time, and permissions. Other attributes may include the name of the application that created the file, the place the file was downloaded from, or a desktop icon that is associated with the file.
- **How is the file metadata, data, and directory structure laid out on the disk?** The disk is just a collection of blocks and we need to throw some data structures on top of that to allow us to store and retrieve data.

Mounting

A file system must be **mounted** before it can be used by the operating system and made available to users. The mount process makes the set of files and directories within a disk partition (or disk or memory key/card) available within the file name space. The **mount** system call is given the **file system type**, **block device**, and **mount point**. The data on the

block device is interpreted in the format of the specified file system type and made available at the mount point. The mount point is any directory in the existing name space. The new mounted file system's root will be presented under that mount point. Any files and directories under that mount point directory in the original file system will no longer be visible.

A **pathname** identifies a file's location in a file system. For example, `/usr/src/linux-2.6/lib/errno.c` identifies the pathname of a file whose name is `errno.c`. When the operating system is looking up a pathname, it may have to traverse mount points and continue its search on other file systems.

Virtual File System Interface (VFS)

To make it easy to support different file systems, Sun introduced a layer of abstraction called the Virtual File System (VFS) Interface to its version of UNIX. It has since been adopted in minor variations to many other POSIX-like systems. The idea behind VFS is to provide an object-oriented approach to file systems. System calls related to file systems interact with the VFS layer. The VFS layer implements the high-level, generic parts related to managing files and directories. It interfaces with one or more file system modules that implement file system code that is unique to each file system. Each file system module must expose a common functional interface. The file system module interacts with the buffer cache and block device drivers to store and retrieve data from the actual devices. With this approach, we provide a set of well-defined responsibilities:

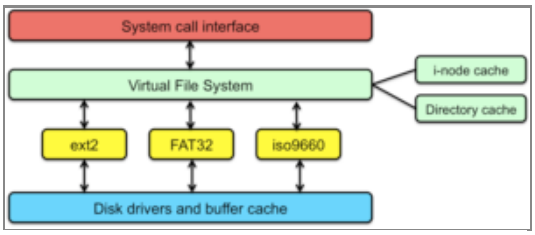

Figure 1. Virtual File System

- system call interface: APIs for user programs
- virtual file system: manages the namespace, keeps track of open files, reference counts, file system types, mount points, pathname traversal.
- file system module: understands how the file system is implemented on the disk. Can fetch and store metadata and data for a file, get directory contents, create and delete files and directories
- buffer cache: no understanding of the file system; takes read and write requests for blocks or parts of a block and caches frequently used blocks.
- device drivers: the components that actually know how to read and write data to the disk.

VFS keeps track of four basic types of objects in a file system:

- The **superblock** is a structure that defines the file system and its current state (clean or dirty; on a disk, dirty means that the file system was not cleanly unmounted and some data blocks may not have been written out)
- The **inode** number is a unique identifier for every object (file) in a specific file system. Its contents contain metadata for the file. File systems are able to translate a file name (pathname) to an inode. The VFS inode structure defines all the operations that are possible on inodes. The term "inode" originated with the Unix File System. Its use in VFS is abstract since it does not necessarily represent a storage structure on the disk.
- A **dentry**, or directory entry, is a collection of file name to inode mappings. A dentry contains the file name as a string, its corresponding inode, and a pointer to the parent dentry. A hierarchical pathname is resolved by going though it component by component, mapping each name to its corresponding inode to read the next directory file. The dentry lives only in memory within VFS and is not synced to the underlying file system.
- A **file** represents an open file. VFS keeps track of the state of the files, such as its access

mode (e.g., read-only) and reference counts (number of times the file is open).

We will take a closer look at these entries in the next section.

The superblock

The superblock is the structure that contains information about the overall file system. It includes the file system name, its type, its size, state (clean or dirty), and the block device on which it resides. It also contains a list of operations for managing inodes (metadata), the superblock, and the file system state. These operations are defined in `linux/fs.h` on Linux systems as:

```
struct super_operations {
        struct inode *(*alloc_inode) (struct super_block *sb);
        void (*destroy_inode) (struct inode *);
        void (*read_inode) (struct inode *);
        void (*dirty_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*sync_fs) (struct super_block *, int);
        void (*write_super_lockfs) (struct super_block *);
        void (*unlockfs) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
        int (*show_options) (struct seq_file *, struct vfsmount *);
};
```

Mounting file systems

Like device drivers, file systems can either be compiled into the kernel or built as separate dynamically loadable modules. Each file system type has to register itself with VFS, which maintains a linked list of known file systems.

Before mounting a file system, the VFS layer first checks if it knows the file system type. It looks through the `file_systems` list for a file system type that matches the type requested by the mount. If the file system type is not found, the kernel daemon will then load the file system module. Once the file system is known (i.e., a module for it exists), the kernel is ready to mount it onto the specified directory. It has to first check the pathname to make sure that the specified mount point is indeed a directory and check the list of mounted file systems to make sure that nothing is already mounted on that directory. If the checks pass, then the file system and corresponding mount point is added to the kernel's list of mounted file systems (on Linux, this is a linked list whose head is at `current->namespace->list`).

The inode

The inode uniquely identifies a file in a file system and contains metadata about the file *except* for the name of the file, which is present in directory entries. The separation of the name from the metadata is a useful one since it allows multiple names from multiple places in the file system to all point to the same inode and hence refer to the exact same file.

The inode information includes an inode number, access permissions, the file's user ID, the file's group ID, major/minor numbers if it's a file representing a device, size of the file, access/creation/modification timestamps, a pointer to the superblock to identify the file system that owns it, and a list of operations available for inodes.

The operations defined for inodes include functions to allow you to create and remove (unlink) files, create and remove directories, create special device files, set permissions, get/set attributes, and create links to files. Essentially, these are operations on files but not on

their data. The set of operations in Linux is:

```
struct inode_operations {
        int (*create) (struct inode *, struct dentry *, int);
        struct dentry * (*lookup) (struct inode *, struct dentry *);
        int (*link) (struct dentry *, struct inode *, struct dentry *);
        int (*unlink) (struct inode *, struct dentry *);
        int (*symlink) (struct inode *, struct dentry *, const char *);
        int (*mkdir) (struct inode *, struct dentry *, int);
        int (*rmdir) (struct inode *, struct dentry *);
        int (*mknod) (struct inode *, struct dentry *, int, dev_t);
        int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char *,int);
        int (*follow_link) (struct dentry *, struct nameidata *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *, const void *, size_t, int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
};
```

Files

The file object stores information about open files and the list of allowable operations on them. The operations are:

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                unsigned long, unsigned long);
};
```

File system implementation

The VFS layer gives us an abstract interface to the file system. That is, it knows about directories, files, and metadata but it does not know or care about how they are stored on the disk. In this section, we turn our attention to the file system module, which has to implement those abstract interfaces for creating directories, reading bytes from a file, and everything else that the VFS layer expects from the file system.

Some terminology

There are a number of terms that we'll use (or at least touch upon) as we discuss file systems. Let's define them.

1. **Disk**: Technically, a disk refers to a spinning magnetic disk used for data storage but when we talk about file systems, we use it to refer to any block-addressable storage, which includes disks and flash memory. We generally expect disk storage to be **non-volatile** — persistent even after power goes off but this is not always the case. We can treat a block of system memory as a disk and create an in-memory file system as long as we provide an interface to read and write "blocks" of memory. We can even have a file on a disk act like a block device and hold a file system.

2. **Disk block**, or **sector**. This refers to the smallest chunk of I/O permissible on a disk. Most of today's disks have 512-byte blocks (e.g., a Western Digital 2 TB drive has 3,907,029,168 512-byte blocks). A disk drive is composed of one or more disk platters ("heads", in reference to the read/write head for that platter). Each platter contains a number of concentric **tracks**. The set of the same track number across all heads is called the **cylinder**. Each track is divided into fixed-size **sectors**. In the past (e.g., before the 1990s), we had an awareness of what cylinder-head-sector (CHS) was being requested and disk scheduling algorithms tried to optimize disk accesses. As disks became smarter, got substantial caches (and even adjunct flash memory), they became capable of reordering disk requests, remapping bad sectors, and servicing several requests concurrently. Trying to optimize for CHS became futile and we would think of a disk as simply storing a sequence of blocks, each with a unique number known as a **logical block address** (**LBA**). With flash memory, the concept of cylinders, tracks, and heads makes no sense.

3. **Partition**: Some disks may be divided into several chunks — smaller logical disks. A partition is a contiguous subset of all the blocks on a disk. A disk will have at least one partition.

4. **Volume**: A volume is either a partition, disk, or even a set of disks that contains a single file system.

5. **Superblock**: We saw *superblock* used at the VFS layer as an abstract concept. On a disk, a superblock is an area that contains key information about the file system (volume size, cluster size, type of file system, name, a flag indicating whether it was cleanly unmounted, etc.).

6. **Metadata**: The various attributes of a file, not the actual file data.

7. **inode**: A structure on the disk that stores a file's metadata and the location of the file's data.

Directories

Directories (sometimes known as folders to those whose view of a system is through a graphical user interface) tell us where files and other directories are located. They are a crucial element for finding a file's contents and for managing human-friendly names.

In most file systems, a directory is simply a file, just like every other file in the system. It just needs some flag that identifies it as a directory. Its contents will, of course, contain the file name. They need to tell us how to find the data and metadata that is associated with the file. There are two basic design options for this:

1. Option 1: The directory contains the file name and a pointer to something that contains the metadata and data for the file. This was the approach taken with the UNIX file system, where a file name has an inode number associated with it. The inode contains

the metadata about the file as well as the location of the file's data. This is the dominant approach for most of today's file systems.

2. Option 2: The directory contains the file name, metadata, and a pointer to the data for the file. This was the approach taken with Microsoft's FAT file systems (FAT-12, FAT-16, and FAT-32). The advantage of this approach is that directory listings that show file attributes can be done more efficiently since you don't need to look up the metadata. The disadvantage is that hard links cannot be implemented.

A common storage organization for directory entries was a **linear list**. For example, the Berkeley Fast File System (which we'll examine in greater detail) uses a simple list where every directory entry contains:

1. the size of the entry
2. the type of entry: file or directory (this is not really needed since the definitive information resides in the inode but is provided as a hint to optimize some directory listings)
3. the length of the filename (0-255)
4. the filename

Commonly-used directory entries will be cached by the VFS *dentry* cache, so pathname lookups where many elements of the pathname are the same won't be penalized by performance hits.

This structure isn't particularly efficient when we have to search for a file or directory in a directory with hundreds, thousands, or even tens of thousands of entries. A way of optimizing the list is to turn the simple list into a **hash table**. This would allow us to have linear time directory lookups at the expense of having a bigger directory file and managing collision chains.

More advanced structures, such as **B-trees** can also be used to provide an efficient structure that lends itself to adding and deleting elements easily while keeping the tree balanced to ensure rapid searches. NTFS, for example, uses a B-tree for its directories. Linux's ext3 and ext4 file systems use a structure called an **Htree,** which is similar to a B-tree for managing large directories.

Implementing file operations

This section is not a thorough treatment of file operations but, rather, is intended to provide an idea of the higher-level operations that take place in implementing certain common operations on files and file systems.

Initialization

There are two concepts of formatting: low-level and high-level. In **low-level formatting**, we define the sectors on a track, create spare sectors, and identify and remap bad blocks. These functions are done by the disk controller logic in response to a request from the driver software. In **high-level formatting**, we are concerned with laying out a file system on a barren sequence of blocks. This requires initializing a free/used block map (usually a bitmap), initializing size of the inode and journal areas and positioning them in the volume, initializing the superblock, and creatiing a top-level (root) directory.

Mounting & unmounting a file system

All scheduled output as well as any modified blocks for the device in the the buffer cache have to be flushed out to the disk. Once all the data is written, the file system is marked as "clean" in the superblock. This means that it is in a consistent state. On a mount request, we first check this "clean" field. If the file system is clean, we proceed with the mount (see earlier discussion). If it's not clean, then we have to run a file system consistency checking program (e.g., *fsck*).

Opening files

Opening a file is a two-step process:

1. look up the pathname (*namei* function in VFS as well as every file system module): traverse the directory structure based on the pathname to find the file. Return the associated inode. VFS will cache frequently-used directory entries.
2. Now that we have an inode, verify access permissions (policy) and allocate in-memory structure to maintain state about the access (e.g., open in read-only or append mode)

Writing files

Depending on where our seek pointer is, writing means that we either overwrite data in a file or, if writing at the end, we need to grow the file. If we're just modifying data in the file then we read the associated blocks, modify the bytes, and write them out again. If we need to grow the file, we need to:

1. allocate disk blocks to hold the data.
2. add the blocks to the list of blocks owned by the file.
3. modify the free/used block bitmap.
4. modify the block map in the inode or in indirect blocks (depending on the file system implementation).
5. modify the file length in the inode.
6. change the current file offset in the kernel.

Deleting files

Deleting files takes a few steps:

1. Remove the name from the directory. This prevents others from opening the file.
2. If there are no more links to the inode then mark the file for deletion. Note that neither the data nor metadata is released until there are no more programs referencing it.
3. If there are no more programs with open handles to the file then release the resources used by the file: return data blocks to the free block map, and return inode to the free inode list.

An example of how reference counts help you is using temporary files in a program. A common practice is to create and open a temporary file, delete it, but continue to access it. As long as the process still has the file open, all the storage structures still exist on the disk. There is just no directory that points to it. As soon as the program exits, the reference to the file is decremented, goes to zero, and the VFS will request that the file be deleted. This practice of deleting the file first is done to ensure that no stray temporary files are left even if the process exits abnormally.

Some additional file system operations

Symbolic links

The file's data contains a path name. When opening a file or traversing a directory, once a symbolic link is encountered, File pathname resolution continues by reading the pathname in that link. A problem with symbolic links is that the file it references can disappear or change but the links still remain. See the POSIX *ln -s* command

Hard links (also known as aliases)

Hard links (see the POSIX *ln* command) are multiple directory entries that all refer to the same inode. They are not pointers. One name is just as valid as any other that points to the

same inode. The inode contains reference count. The file itself is deleted only when the reference count goes to 0.

Extended attributes (used by NTFS, HFS+, XFS, etc.)

VFS expects certain "standard" attributes, such as file permissions, owner, file creation/modification/access timestamps, length, and a few others as part of the file's metadata. A file system may, however, support many other attributes. For example, a file system may store the URL from downloaded web or ftp content or store the name of the application that created the file.

indexing

Some file systems may support the creation of a database of names and attributes for fast searching of files.

journaling

We will discuss journaling in more detail. Basically, every change to a file system is written to a transaction log. If the file system crashes, the system can replay the log to get it to a known, coherent, state.

access control lists

The classic UNIX approach to access control is to allow the setting of read, write, and execute permissions for the owner, a specific group, and everyone else. While restrictive, this used a small, fixed-size of data. Access control lists allow an enumerated list of users and their access rights.

Block allocation

File data needs disk blocks. A **block allocation** algorithm is responsible for figuring out how to keep track of free space on the disk and which free blocks to allocate for a file.

Contiguous allocation

A desirable layout of files is to have each file occupy a set of adjacent blocks in storage. All you need to know to retrieve the file's data is the starting block and the length of the file. Seeking to a random point is easy and performance for reading the file sequentially is great since all the blocks are in physical proximity to each other and the disk's head does not have to seek. It sounds great until you remember the pains of dealing with allocating memory to processors in a variable partition multiprogramming environment without pages. The problem is **external fragmentation**: having free blocks (holes) left when a file is deleted. Will you be able to find another file that's just the right size? What happens if a file grows and there's another file with higher block numbers? Moreover, if your system isn't creating and fully populating its files one at a time, how do you know how much space to allocate for a file?

Extents

One compromise approach to contiguous allocation is to use **extents**. An extent is a contiguous set of blocks but it may not be enough to hold the entire file. As we need more space, we allocate additional extents (contiguous chunks of blocks). With extents, we strive to get contiguous blocks but can handle the case where we cannot. Because a file may comprise more than one extent, we will need to have a way of keeping track of all the extents for a file.

When extents are used, they are typically used as alternatives to block numbers. If the same data structures are used as for block number based file systems, then the extent will be stored in place of a block number but contain the starting block as well as the range of blocks in that extent. For example, a 64-bit extent number may contain a 48-bit block address along with a two-byte length, allowing an extent to span up to 65,535 consecutive blocks.

A problem with extents is handling file seek operations. If you want to get to a certain

position in a file, it's easy enough to figure out what block number it is but you will need to traverse the list of extents from the beginning of the file to find which extent encompasses that block (or use a more suitable storage structure, such as a B-tree.

Linked allocation

With linked allocation, a file's data is stored on an arbitrary set of disk blocks (we can use a bitmap to keep track of which blocks on a disk are free and which are not). The directory entry for the file contains the block number of the first block of the file's contents. Each block, in addition to the file's data, contains the block number of the next allocated block in the file.

It may be irritating to have a block pointer taking up a bit of space in the block. This means that instead of a 512-byte block, the file system driver has to treat it as a 508-byte block with a 4-byte block pointer. The real problem with this form of allocation, however, is that seeking to an arbitrary point in a file (random access) may be extremely time-consuming. Let's suppose we have a 16 MB file and need to seek to the 15 MB position: byte offset 15,728,640. We have to read every block from the very first block of the file to follow the list of blocks to get to this offset. With 512-byte blocks, we'll be reading 30,720 blocks just to read the block that we really want. The same problem is encountered if we want to append to this file. We will need to seek and read 32,768 blocks just to get to the end of the file so we can set a pointer to the next new block.

A way to reduce both the overhead of block pointers as well as the number of seek+read operations that we need to do is to use a **logical block size** in our file system and group multiple physical blocks into these fixed-size logical chunks. For example, instead of using a disk's native 512-byte block, we can use a block size of 4096 bytes (4 KB). That will reduce the number of seek+reads that we have to do in the above example by a factor of eight: 3,840 versus 30,720. It's still a ridiculously high number, making a linked allocation approach impractical for anything but sequentially accessed file systems with no append capability. The approach of grouping multiple physical blocks together, however, is often used and is called **clustering**. A **cluster** is a logical rather than physical block size that is a multiple of physical blocks. It is sometimes referred to as an *allocation unit*. It improves throughput because the block allocations are adjacent. On a disk, reading multiple adjacent blocks is significantly more efficient compared to reading random blocks. When we look at the Berkeley Fast File System, we'll see that just using a larger logical block size yielded huge performance gains. The downside of clusters is increased **internal fragmentation**: there's a higher chance that there will be more unused space within an allocated cluster. For instance, with 4096-byte clusters, a 16-byte file will cause 4080 bytes to go unallocated.

File Allocation Table (FAT)

A variation on linked allocation is to move the block pointers out of the blocks and into a dedicated array of block numbers that is stored in a special place in the disk at the start of the volume. This table is called the **file allocation table**. The directory entry for a file contains the starting block number of the file. Each element of the file allocation table contains the number of the *next* block of the file. There is one of these per file system since the elements represent blocks on the disk. To traverse a file sequentially, you

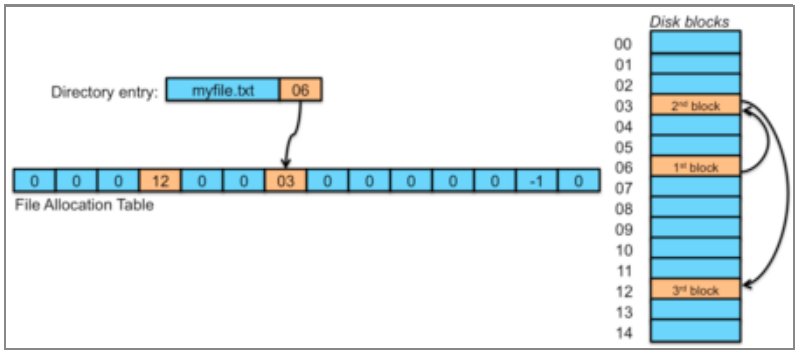

Figure 2. FAT allocation

simply follow the chain within the table (figure 2). Here's an example in pseudocode of printing the blocks that a file has allocated to it in the sequence that it uses them:

```
b = dir[i].block; /* assume directory entry i contains the file we want */
for (; b != -1; b = fat[b]);
        printf("block: %d\n", b);
```

The File allocation table contains every allocation list within it. Seeking requires traversing the list from the start of a file but it no longer requires reading every block of the file. If the entire FAT cannot be cached in memory then performance will lag for any file accesses since parts of the FAT will have to be read in. One approach to making the table smaller is to use larger clusters.

Microsoft's FAT file systems

The FAT approach is used by Microsoft's FAT12, FAT16, and FAT32 file systems. These were used since the first MSDOS floppy-disk based file systems (FAT12) and still supported through today's systems. All of these file systems rely on clusters (groups of blocks) and the number after the FAT refers to the size of the cluster pointers. The FAT12 file system, for instance, was designed for floppy disks and used 12-bit cluster addresses, limiting the cluster count to 4,084 (12 clusters were reserved) and supporting a maximum volume (partition) size of 32 MB.

FAT16 used 16-bit cluster pointers and supported up to 64 sectors per cluster, offering a maximum file system size (with 512 byte sectors) of 2 GB. FAT32, introduced with Windows 95, increased the cluster pointers to 32-bit values. With up to 64 sectors per cluster, it now supported a maximum file system size of 8 TB and a maximum file size of 4 GB (due to restrictions in the directory entry fields). In all of these systems, all file metadata was stored in the directory file.

Indexed allocation

Linked allocation, or even a FAT, isn't very efficient for random access and any efficient performance with a FAT requires that the whole table be cached in memory. This can be a considerable amount of memory. For example, a 250 GB disk with 4 KB clusters would require a file allocation table of approximately 15 MB. This is generally not practical. Even then, seeking to some random spot in a file requires following a path of links. If the entire FAT is not sitting in memory, then additional blocks containing pieces of the FAT will have to be read from the disk. A different approach is **indexed allocation**, where we store a list of block pointers for each file in one place: the **index block**, or **inode**. The inode may also contain the file's metadata (length, various timestamps, owner, permissions, etc.). A directory entry for a file will contain the file's inode number. When a file is opened, the file system driver will read the file's inode and get the list of block pointers for the file's contents. Now it has a list of the blocks it needs — just those for the file — rather than a list of all the blocks in the file system.

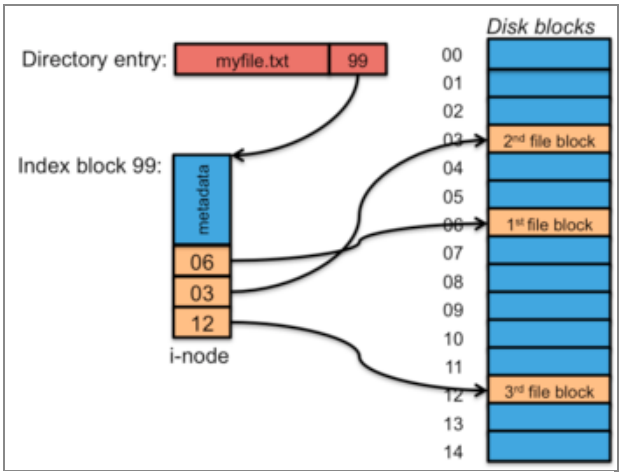

Figure 3. Indexed allocation

Combined indexing

The problem with the indexed allocation approach is that files are of varying sizes. Hence, the

list of blocks used by a file will vary in size, resulting in variable sized inodes. That's not good for three reasons. First, a fixed-size inode makes it easy to allocate and re-use inodes as files get created and destroyed. You have no problems with external fragmentation when your objects are all the same size. Secondly, a fixed-size inode makes it easy to find locate on the disk. If we devote a portion of the disk to a table of inodes, we know that the byte offset of the inode *i* is:

```
offset = sizeof(inode)*i;
```

and the corresponding disk block holding the inode will be at:

```
block = inode_base + offset/blocksize;
```

The third issue is in-kernel caching. We'd like it to be easy to manage cached inodes and not have the inode for a huge file take up a disproportionate part of the cache.

For all these reasons, a variation of indexed allocation, called **combined indexing** is more practical. Combined indexing uses a fixed-size inode. This inode contains a number of **direct block pointers**. This is an array of the first few blocks allocated to the file. If a file is large and the fixed-size table (typically 10-12 entries) is not big enough to hold all the blocks that make up the file's data, the next entry in the inode is an **indirect block pointer**. This is also a disk block number but the contents of this disk block are just a sequence of the block numbers that are allocated to the file.

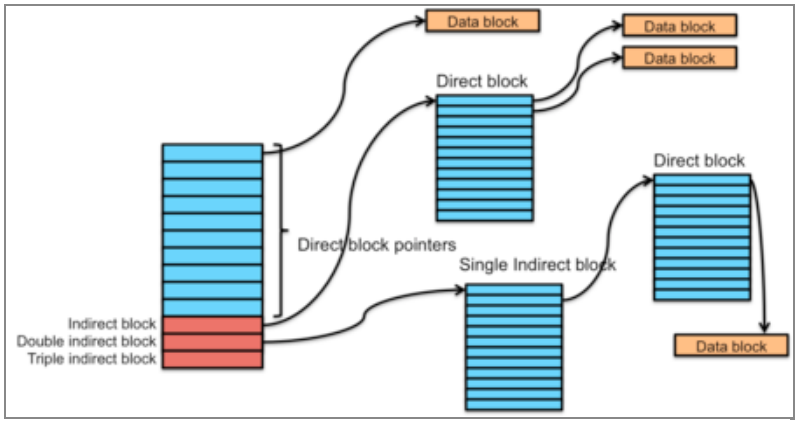

Figure 4. Combined indexing

Unix File System

The combined indexing approach is the exact method used in the Unix File System (UFS): the file system that was created in the early Bell Labs versions of Unix and used in AT&T's versions of Unix through the 1980s. The Unix File System's inode comprises 10 direct blocks, one indirect block, one double indirect block, and one triple indirect block. It typically used 1024-byte blocks and 32-bit block pointers.

With 1024-byte blocks and 10 direct blocks, we can store a list of blocks for a 10 KB file. If the file gets any bigger, we will need to use an indirect block. Assuming 32-bit (4 byte) block numbers, the indirect block can store a list of 256 (1024/4) block pointers, mapping another 256 KB (256 × 1 KB) of the file. If the file gets larger than 256 KB, then we use the next entry in the inode, which is a **double indirect block pointer**. This is the number of a block whose entire set of 256 entries are block numbers of direct blocks. Now we can address an additional 64 MB (256 × 256 KB) of storage. If that's not enough, the entry after that in the inode is a **triple indirect block pointer**, which points to a double indirect block and allows us to address an additional 16 GB (256 × 64 MB). Hence, without using a larger cluster, the biggest file that the file system could support was 64 MB + 256 KB + 10 KB, or 66.5 MB.

References

- The File System (http://tldp.org/LDP/tlk/fs/filesystem.html), Linux Documentation Project

- The Linux Virtual File System (http://www.win.tue.nl/~aeb/linux/lk/lk-8.html), The Linux Kernel, Andries Brouwer
- How NTFS Works (http://technet.microsoft.com/en-us/library/cc781134(WS.10).aspx), Microsoft TechNet
- What Is NTFS? (http://technet.microsoft.com/en-us/library/cc778410(v=WS.10).aspx), Microsoft TechNet
- Files and Filessytems (http://www.linux-tutorial.info/modules.php?name=MContent&pageid=95), The Linux Tutorial, James Mohr and David Rusling.

© 2003-2013 Paul Krzyzanowski. All rights reserved.

For questions or comments about this site, contact Paul Krzyzanowski, webinfo@pk.org

The entire contents of this site are protected by copyright under national and international law. No part of this site may be copied, reproduced, stored in a retrieval system, or transmitted, in any form, or by any means whether electronic, mechanical or otherwise without the prior written consent of the copyright holder. If there is something on this page that you want to use, please let me know.

Any opinions expressed on this page do not necessarily reflect the opinions of my employers and may not even reflect my own.

Last updated: February 2, 2013