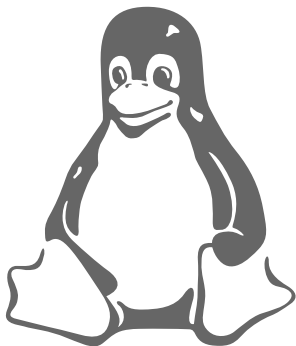


Chapter 9

The File system



This chapter describes how the Linux kernel maintains the files in the file systems that it supports. It describes the Virtual File System (VFS) and explains how the Linux kernel's real file systems are supported.

One of the most important features of Linux is its support for many different file systems. This makes it very flexible and well able to coexist with many other operating systems. At the time of writing, Linux supports 15 file systems; `ext`, `ext2`, `xia`, `minix`, `umsdos`, `msdos`, `vfat`, `proc`, `smb`, `ncp`, `iso9660`, `sysv`, `hpfs`, `affs` and `ufs`, and no doubt, over time more will be added.

In Linux, as it is for Unix™, the separate file systems the system may use are not accessed by device identifiers (such as a drive number or a drive name) but instead they are combined into a single hierarchical tree structure that represents the file system as one whole single entity. Linux adds each new file system into this single file system tree as it is mounted. All file systems, of whatever type, are mounted onto a directory and the files of the mounted file system cover up the existing contents of that directory. This directory is known as the mount directory or mount point. When the file system is unmounted, the mount directory's own files are once again revealed.

When disks are initialized (using `fdisk`, say) they have a partition structure imposed on them that divides the physical disk into a number of logical partitions. Each partition may hold a single file system, for example an `EXT2` file system. File systems organize files into logical hierarchical structures with directories, soft links and so on held in blocks on physical devices. Devices that can contain file systems are known as block devices. The IDE disk partition `/dev/hda1`, the first partition of the first IDE disk drive in the system, is a block device. The Linux file systems regard these

block devices as simply linear collections of blocks, they do not know or care about the underlying physical disk's geometry. It is the task of each block device driver to map a request to read a particular block of its device into terms meaningful to its device; the particular track, sector and cylinder of its hard disk where the block is kept. A file system has to look, feel and operate in the same way no matter what device is holding it. Moreover, using Linux's file systems, it does not matter (at least to the system user) that these different file systems are on different physical media controlled by different hardware controllers. The file system might not even be on the local system, it could just as well be a disk remotely mounted over a network link. Consider the following example where a Linux system has its root file system on a SCSI disk:

A	E	boot	etc	lib	opt	tmp	usr
C	F	cdrom	fd	proc	root	var	sbin
D	bin	dev	home	mnt	lost+found		

Neither the users nor the programs that operate on the files themselves need know that /C is in fact a mounted VFAT file system that is on the first IDE disk in the system. In the example (which is actually my home Linux system), /E is the master IDE disk on the second IDE controller. It does not matter either that the first IDE controller is a PCI controller and that the second is an ISA controller which also controls the IDE CDROM. I can dial into the network where I work using a modem and the PPP network protocol using a modem and in this case I can remotely mount my Alpha AXP Linux system's file systems on /mnt/remote.

The files in a file system are collections of data; the file holding the sources to this chapter is an ASCII file called `filesystems.tex`. A file system not only holds the data that is contained within the files of the file system but also the structure of the file system. It holds all of the information that Linux users and processes see as files, directories soft links, file protection information and so on. Moreover it must hold that information safely and securely, the basic integrity of the operating system depends on its file systems. Nobody would use an operating system that randomly lost data and files¹.

Minix, the first file system that Linux had is rather restrictive and lacking in performance. Its filenames cannot be longer than 14 characters (which is still better than 8.3 filenames) and the maximum file size is 64MBytes. 64Mbytes might at first glance seem large enough but large file sizes are necessary to hold even modest databases. The first file system designed specifically for Linux, the Extended File system, or EXT, was introduced in April 1992 and cured a lot of the problems but it was still felt to lack performance. So, in 1993, the Second Extended File system, or EXT2, was added. It is this file system that is described in detail later on in this chapter.

An important development took place when the EXT file system was added into Linux. The real file systems were separated from the operating system and system services by an interface layer known as the Virtual File system, or VFS. VFS allows Linux to support many, often very different, file systems, each presenting a common software interface to the VFS. All of the details of the Linux file systems are translated by software so that all file systems appear identical to the rest of the Linux kernel

¹Well, not knowingly, although I have been bitten by operating systems with more lawyers than Linux has developers

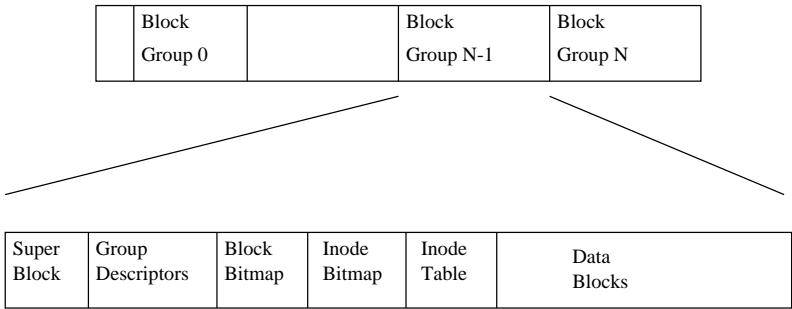


Figure 9.1: Physical Layout of the EXT2 File system

and to programs running in the system. Linux’s Virtual File system layer allows you to transparently mount the many different file systems at the same time.

The Linux Virtual File system is implemented so that access to its files is as fast and efficient as possible. It must also make sure that the files and their data are kept correctly. These two requirements can be at odds with each other. The Linux VFS caches information in memory from each file system as it is mounted and used. A lot of care must be taken to update the file system correctly as data within these caches is modified as files and directories are created, written to and deleted. If you could see the file system’s data structures within the running kernel, you would be able to see data blocks being read and written by the file system. Data structures, describing the files and directories being accessed would be created and destroyed and all the time the device drivers would be working away, fetching and saving data. The most important of these caches is the Buffer Cache, which is integrated into the way that the individual file systems access their underlying block devices. As blocks are accessed they are put into the Buffer Cache and kept on various queues depending on their states. The Buffer Cache not only caches data buffers, it also helps manage the asynchronous interface with the block device drivers.

9.1 The Second Extended File system (EXT2)

The Second Extended File system was devised (by Rémy Card) as an extensible and powerful file system for Linux. It is also the most successful file system so far in the Linux community and is the basis for all of the currently shipping Linux distributions. The EXT2 file system, like a lot of the file systems, is built on the premise that the data held in files is kept in data blocks. These data blocks are all of the same length and, although that length can vary between different EXT2 file systems the block size of a particular EXT2 file system is set when it is created (using `mke2fs`). Every file’s size is rounded up to an integral number of blocks. If the block size is 1024 bytes, then a file of 1025 bytes will occupy two 1024 byte blocks. Unfortunately this means that on average you waste half a block per file. Usually in computing you trade off CPU usage for memory and disk space utilisation. In this case Linux, along with most operating systems, trades off a relatively inefficient disk usage in order to reduce the workload on the CPU. Not all of the blocks in the file system hold data, some must be used to contain the information that describes the structure of the file system. EXT2 defines the file system topology by describing each file in the system with an inode data structure. An inode describes which blocks the data within a

See
fs/ext2/*

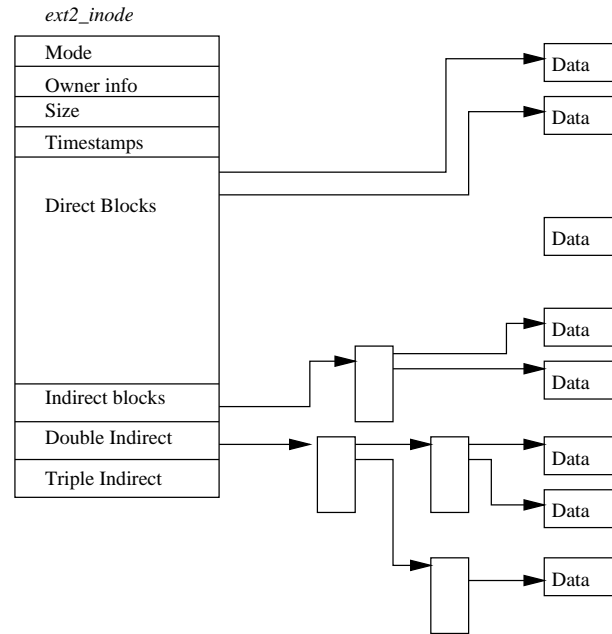


Figure 9.2: EXT2 Inode

file occupies as well as the access rights of the file, the file's modification times and the type of the file. Every file in the EXT2 file system is described by a single inode and each inode has a single unique number identifying it. The inodes for the file system are all kept together in inode tables. EXT2 directories are simply special files (themselves described by inodes) which contain pointers to the inodes of their directory entries.

Figure 9.1 shows the layout of the EXT2 file system as occupying a series of blocks in a block structured device. So far as each file system is concerned, block devices are just a series of blocks that can be read and written. A file system does not need to concern itself with where on the physical media a block should be put, that is the job of the device's driver. Whenever a file system needs to read information or data from the block device containing it, it requests that its supporting device driver reads an integral number of blocks. The EXT2 file system divides the logical partition that it occupies into Block Groups. Each group duplicates information critical to the integrity of the file system as well as holding real files and directories as blocks of information and data. This duplication is necessary should a disaster occur and the file system need recovering. The subsections describe in more detail the contents of each Block Group.

9.1.1 The EXT2 Inode

In the EXT2 file system, the inode is the basic building block; every file and directory in the file system is described by one and only one inode. The EXT2 inodes for each Block Group are kept in the inode table together with a bitmap that allows the system to keep track of allocated and unallocated inodes. Figure 9.2 shows the format of an EXT2 inode, amongst other information, it contains the following fields:

See
`include/linux/-`
`ext2_fs_i.h`

mode This holds two pieces of information; what this inode describes and the permissions that users have to it. For EXT2, an inode can describe one of file, directory, symbolic link, block device, character device or FIFO.

Owner Information The user and group identifiers of the owners of this file or directory. This allows the file system to correctly allow the right sort of accesses,

Size The size of the file in bytes,

Timestamps The time that the inode was created and the last time that it was modified,

Datablocks Pointers to the blocks that contain the data that this inode is describing. The first twelve are pointers to the physical blocks containing the data described by this inode and the last three pointers contain more and more levels of indirection. For example, the double indirect blocks pointer points at a block of pointers to blocks of pointers to data blocks. This means that files less than or equal to twelve data blocks in length are more quickly accessed than larger files.

You should note that EXT2 inodes can describe special device files. These are not real files but handles that programs can use to access devices. All of the device files in `/dev` are there to allow programs to access Linux's devices. For example the `mount` program takes as an argument the device file that it wishes to mount.

9.1.2 The EXT2 Superblock

The Superblock contains a description of the basic size and shape of this file system. The information within it allows the file system manager to use and maintain the file system. Usually only the Superblock in Block Group 0 is read when the file system is mounted but each Block Group contains a duplicate copy in case of file system corruption. Amongst other information it holds the:

See
`include/linux/-
 ext2_fs_sb.h`

Magic Number This allows the mounting software to check that this is indeed the Superblock for an EXT2 file system. For the current version of EXT2 this is `0xEF53`.

Revision Level The major and minor revision levels allow the mounting code to determine whether or not this file system supports features that are only available in particular revisions of the file system. There are also feature compatibility fields which help the mounting code to determine which new features can safely be used on this file system,

Mount Count and Maximum Mount Count Together these allow the system to determine if the file system should be fully checked. The mount count is incremented each time the file system is mounted and when it equals the maximum mount count the warning message “maximal mount count reached, running `e2fsck` is recommended” is displayed,

Block Group Number The Block Group number that holds this copy of the Superblock,

Block Size The size of the block for this file system in bytes, for example 1024 bytes,

Blocks per Group The number of blocks in a group. Like the block size this is fixed when the file system is created,

Free Blocks The number of free blocks in the file system,

Free Inodes The number of free Inodes in the file system,

First Inode This is the inode number of the first inode in the file system. The first inode in an EXT2 root file system would be the directory entry for the '/' directory.

9.1.3 The EXT2 Group Descriptor

Each Block Group has a data structure describing it. Like the Superblock, all the group descriptors for all of the Block Groups are duplicated in each Block Group in case of file system corruption. Each Group Descriptor contains the following information:

See
`ext2_group_desc`
 in `include/linux/ext2_fs.h`

Blocks Bitmap The block number of the block allocation bitmap for this Block Group. This is used during block allocation and deallocation,

Inode Bitmap The block number of the inode allocation bitmap for this Block Group. This is used during inode allocation and deallocation,

Inode Table The block number of the starting block for the inode table for this Block Group. Each inode is represented by the EXT2 inode data structure described below.

Free blocks count, Free Inodes count, Used directory count

The group descriptors are placed on after another and together they make the group descriptor table. Each Blocks Group contains the entire table of group descriptors after its copy of the Superblock. Only the first copy (in Block Group 0) is actually used by the EXT2 file system. The other copies are there, like the copies of the Superblock, in case the main copy is corrupted.

9.1.4 EXT2 Directories

In the EXT2 file system, directories are special files that are used to create and hold access paths to the files in the file system. Figure 9.3 shows the layout of a directory entry in memory. A directory file is a list of directory entries, each one containing the following information:

See
`ext2_dir_entry`
 in `include/linux/ext2_fs.h`

inode The inode for this directory entry. This is an index into the array of inodes held in the Inode Table of the Block Group. In figure 9.3, the directory entry for the file called `file` has a reference to inode number `i1`,

name length The length of this directory entry in bytes,

name The name of this directory entry.

The first two entries for every directory are always the standard “.” and “..” entries meaning “this directory” and “the parent directory” respectively.

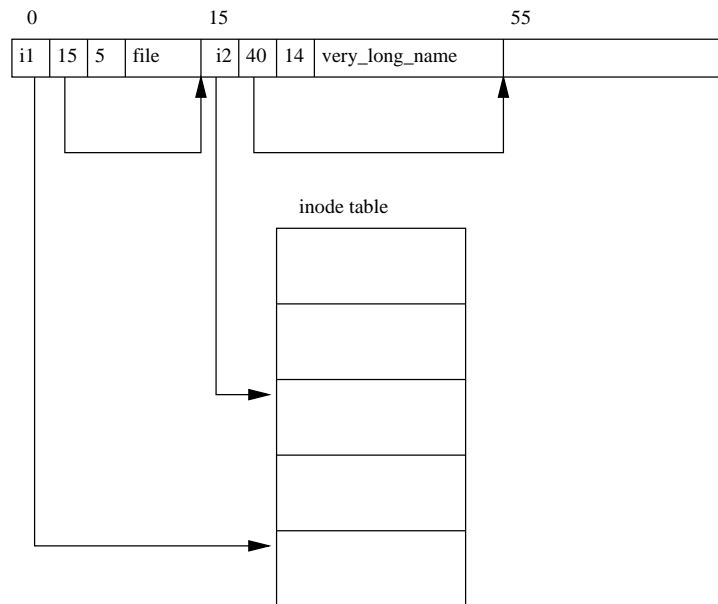


Figure 9.3: EXT2 Directory

9.1.5 Finding a File in an EXT2 File System

A Linux filename has the same format as all Unix™ filenames have. It is a series of directory names separated by forward slashes (“/”) and ending in the file’s name. One example filename would be `/home/rusling/.cshrc` where `/home` and `/rusling` are directory names and the file’s name is `.cshrc`. Like all other Unix™ systems, Linux does not care about the format of the filename itself; it can be any length and consist of any of the printable characters. To find the inode representing this file within an EXT2 file system the system must parse the filename a directory at a time until we get to the file itself.

The first inode we need is the inode for the root of the file system and we find its number in the file system’s superblock. To read an EXT2 inode we must look for it in the inode table of the appropriate Block Group. If, for example, the root inode number is 42, then we need the 42nd inode from the inode table of Block Group 0. The root inode is for an EXT2 directory, in other words the mode of the root inode describes it as a directory and its data blocks contain EXT2 directory entries.

`home` is just one of the many directory entries and this directory entry gives us the number of the inode describing the `/home` directory. We have to read this directory (by first reading its inode and then reading the directory entries from the data blocks described by its inode) to find the `rusling` entry which gives us the number of the inode describing the `/home/rusling` directory. Finally we read the directory entries pointed at by the inode describing the `/home/rusling` directory to find the inode number of the `.cshrc` file and from this we get the data blocks containing the information in the file.

9.1.6 Changing the Size of a File in an EXT2 File System

One common problem with a file system is its tendency to fragment. The blocks that hold the file’s data get spread all over the file system and this makes sequentially

accessing the data blocks of a file more and more inefficient the further apart the data blocks are. The EXT2 file system tries to overcome this by allocating the new blocks for a file physically close to its current data blocks or at least in the same Block Group as its current data blocks. Only when this fails does it allocate data blocks in another Block Group.

Whenever a process attempts to write data into a file the Linux file system checks to see if the data has gone off the end of the file's last allocated block. If it has, then it must allocate a new data block for this file. Until the allocation is complete, the process cannot run; it must wait for the file system to allocate a new data block and write the rest of the data to it before it can continue. The first thing that the EXT2 block allocation routines do is to lock the EXT2 Superblock for this file system. Allocating and deallocating changes fields within the superblock, and the Linux file system cannot allow more than one process to do this at the same time. If another process needs to allocate more data blocks, it will have to wait until this process has finished. Processes waiting for the superblock are suspended, unable to run, until control of the superblock is relinquished by its current user. Access to the superblock is granted on a first come, first served basis and once a process has control of the superblock, it keeps control until it has finished. Having locked the superblock, the process checks that there are enough free blocks left in this file system. If there are not enough free blocks, then this attempt to allocate more will fail and the process will relinquish control of this file system's superblock.

See
`ext2_new_block()`
in `fs/ext2/-`
`ballo.c`

If there are enough free blocks in the file system, the process tries to allocate one. If the EXT2 file system has been built to preallocate data blocks then we may be able to take one of those. The preallocated blocks do not actually exist, they are just reserved within the allocated block bitmap. The VFS inode representing the file that we are trying to allocate a new data block for has two EXT2 specific fields, `prealloc_block` and `prealloc_count`, which are the block number of the first preallocated data block and how many of them there are, respectively. If there were no preallocated blocks or block preallocation is not enabled, the EXT2 file system must allocate a new block. The EXT2 file system first looks to see if the data block after the last data block in the file is free. Logically, this is the most efficient block to allocate as it makes sequential accesses much quicker. If this block is not free, then the search widens and it looks for a data block within 64 blocks of the of the ideal block. This block, although not ideal is at least fairly close and within the same Block Group as the other data blocks belonging to this file.

If even that block is not free, the process starts looking in all of the other Block Groups in turn until it finds some free blocks. The block allocation code looks for a cluster of eight free data blocks somewhere in one of the Block Groups. If it cannot find eight together, it will settle for less. If block preallocation is wanted and enabled it will update `prealloc_block` and `prealloc_count` accordingly.

Wherever it finds the free block, the block allocation code updates the Block Group's block bitmap and allocates a data buffer in the buffer cache. That data buffer is uniquely identified by the file system's supporting device identifier and the block number of the allocated block. The data in the buffer is zero'd and the buffer is marked as "dirty" to show that it's contents have not been written to the physical disk. Finally, the superblock itself is marked as "dirty" to show that it has been changed and it is unlocked. If there were any processes waiting for the superblock, the first one in the queue is allowed to run again and will gain exclusive control of

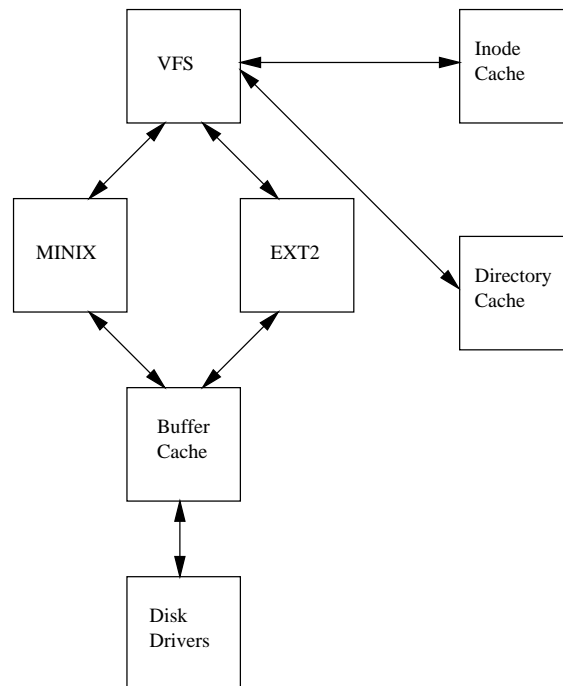


Figure 9.4: A Logical Diagram of the Virtual File System

the superblock for its file operations. The process's data is written to the new data block and, if that data block is filled, the entire process is repeated and another data block allocated.

9.2 The Virtual File System (VFS)

Figure 9.4 shows the relationship between the Linux kernel's Virtual File System and its real file systems. The virtual file system must manage all of the different file systems that are mounted at any given time. To do this it maintains data structures that describe the whole (virtual) file system and the real, mounted, file systems. Rather confusingly, the VFS describes the system's files in terms of superblocks and inodes in much the same way as the EXT2 file system uses superblocks and inodes. Like the EXT2 inodes, the VFS inodes describe files and directories within the system; the contents and topology of the Virtual File System. From now on, to avoid confusion, I will write about VFS inodes and VFS superblocks to distinguish them from EXT2 inodes and superblocks.

See	<code>fs/*</code>
-----	-------------------

As each file system is initialised, it registers itself with the VFS. This happens as the operating system initialises itself at system boot time. The real file systems are either built into the kernel itself or are built as loadable modules. File System modules are loaded as the system needs them, so, for example, if the VFAT file system is implemented as a kernel module, then it is only loaded when a VFAT file system is mounted. When a block device based file system is mounted, and this includes the root file system, the VFS must read its superblock. Each file system type's superblock read routine must work out the file system's topology and map that information onto a VFS superblock data structure. The VFS keeps a list of the

mounted file systems in the system together with their VFS superblocks. Each VFS superblock contains information and pointers to routines that perform particular functions. So, for example, the superblock representing a mounted EXT2 file system contains a pointer to the EXT2 specific inode reading routine. This EXT2 inode read routine, like all of the file system specific inode read routines, fills out the fields in a VFS inode. Each VFS superblock contains a pointer to the first VFS inode on the file system. For the root file system, this is the inode that represents the ‘/’ directory. This mapping of information is very efficient for the EXT2 file system but moderately less so for other file systems.

See
`fs/inode.c`

As the system’s processes access directories and files, system routines are called that traverse the VFS inodes in the system. For example, typing `ls` for a directory or `cat` for a file cause the the Virtual File System to search through the VFS inodes that represent the file system. As every file and directory on the system is represented by a VFS inode, then a number of inodes will be being repeatedly accessed. These inodes are kept in the inode cache which makes access to them quicker. If an inode is not in the inode cache, then a file system specific routine must be called in order to read the appropriate inode. The action of reading the inode causes it to be put into the inode cache and further accesses to the inode keep it in the cache. The less used VFS inodes get removed from the cache.

See
`fs/buffer.c`

All of the Linux file systems use a common buffer cache to cache data buffers from the underlying devices to help speed up access by all of the file systems to the physical devices holding the file systems. This buffer cache is independent of the file systems and is integrated into the mechanisms that the Linux kernel uses to allocate and read and write data buffers. It has the distinct advantage of making the Linux file systems independent from the underlying media and from the device drivers that support them. All block structured devices register themselves with the Linux kernel and present a uniform, block based, usually asynchronous interface. Even relatively complex block devices such as SCSI devices do this. As the real file systems read data from the underlying physical disks, this results in requests to the block device drivers to read physical blocks from the device that they control. Integrated into this block device interface is the buffer cache. As blocks are read by the file systems they are saved in the global buffer cache shared by all of the file systems and the Linux kernel. Buffers within it are identified by their block number and a unique identifier for the device that read it. So, if the same data is needed often, it will be retrieved from the buffer cache rather than read from the disk, which would take somewhat longer. Some devices support read ahead where data blocks are speculatively read just in case they are needed.

See
`fs/dcache.c`

The VFS also keeps a cache of directory lookups so that the inodes for frequently used directories can be quickly found. As an experiment, try listing a directory that you have not listed recently. The first time you list it, you may notice a slight pause but the second time you list its contents the result is immediate. The directory cache does not store the inodes for the directories itself; these should be in the inode cache, the directory cache simply stores the mapping between the full directory names and their inode numbers.

9.2.1 The VFS Superblock

See
`include/linux/fs.h`

Every mounted file system is represented by a VFS superblock; amongst other information, the VFS superblock contains the:

Device This is the device identifier for the block device that this file system is contained in. For example, `/dev/hda1`, the first IDE hard disk in the system has a device identifier of `0x301`,

Inode pointers The `mounted` inode pointer points at the first inode in this file system. The `covered` inode pointer points at the inode representing the directory that this file system is mounted on. The root file system's VFS superblock does not have a `covered` pointer,

Blocksize The block size in bytes of this file system, for example 1024 bytes,

Superblock operations A pointer to a set of superblock routines for this file system. Amongst other things, these routines are used by the VFS to read and write inodes and superblocks.

File System type A pointer to the mounted file system's `file_system_type` data structure,

File System specific A pointer to information needed by this file system,

9.2.2 The VFS Inode

Like the EXT2 file system, every file, directory and so on in the VFS is represented by one and only one VFS inode. The information in each VFS inode is built from information in the underlying file system by file system specific routines. VFS inodes exist only in the kernel's memory and are kept in the VFS inode cache as long as they are useful to the system. Amongst other information, VFS inodes contain the following fields:

See
`include/linux/-
fs.h`

device This is the device identifier of the device holding the file or whatever that this VFS inode represents,

inode number This is the number of the inode and is unique within this file system. The combination of `device` and `inode number` is unique within the Virtual File System,

mode Like EXT2 this field describes what this VFS inode represents as well as access rights to it,

user ids The owner identifiers,

times The creation, modification and write times,

block size The size of a block for this file in bytes, for example 1024 bytes,

inode operations A pointer to a block of routine addresses. These routines are specific to the file system and they perform operations for this inode, for example, truncate the file that is represented by this inode.

count The number of system components currently using this VFS inode. A count of zero means that the inode is free to be discarded or reused,

lock This field is used to lock the VFS inode, for example, when it is being read from the file system,

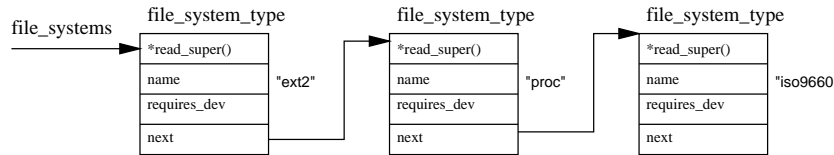


Figure 9.5: Registered File Systems

dirty Indicates whether this VFS inode has been written to, if so the underlying file system will need modifying,

file system specific information

9.2.3 Registering the File Systems

When you build the Linux kernel you are asked if you want each of the supported file systems. When the kernel is built, the file system startup code contains calls to the initialisation routines of all of the built in file systems. Linux file systems may also be built as modules and, in this case, they may be demand loaded as they are needed or loaded by hand using `insmod`. Whenever a file system module is loaded it registers itself with the kernel and unregisters itself when it is unloaded. Each file system's initialisation routine registers itself with the Virtual File System and is represented by a `file_system_type` data structure which contains the name of the file system and a pointer to its VFS superblock read routine. Figure 9.5 shows that the `file_system_type` data structures are put into a list pointed at by the `file_systems` pointer. Each `file_system_type` data structure contains the following information:

See `sys_setup()`
in `fs/-`
`filesystems.c`

See
`file_system_type`
in `include/-`
`linux/fs.h`

Superblock read routine This routine is called by the VFS when an instance of the file system is mounted,

File System name The name of this file system, for example `ext2`,

Device needed Does this file system need a device to support? Not all file system need a device to hold them. The `/proc` file system, for example, does not require a block device,

You can see which file systems are registered by looking in at `/proc/filesystems`. For example:

```

ext2
nodev proc
iso9660
  
```

9.2.4 Mounting a File System

When the superuser attempts to mount a file system, the Linux kernel must first validate the arguments passed in the system call. Although `mount` does some basic checking, it does not know which file systems this kernel has been built to support or that the proposed mount point actually exists. Consider the following `mount` command:

```
$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom
```

This `mount` command will pass the kernel three pieces of information; the name of the file system, the physical block device that contains the file system and, thirdly, where in the existing file system topology the new file system is to be mounted.

The first thing that the Virtual File System must do is to find the file system. To do this it searches through the list of known file systems by looking at each `file_system_type` data structure in the list pointed at by `file_systems`. If it finds a matching name it now knows that this file system type is supported by this kernel and it has the address of the file system specific routine for reading this file system's superblock. If it cannot find a matching file system name then all is not lost if the kernel is built to demand load kernel modules (see Chapter 12). In this case the kernel will request that the kernel daemon loads the appropriate file system module before continuing as before.

See `do_mount()`
in
`fs/super.c`

See
`get_fs_type()` in
`fs/super.c`

Next if the physical device passed by `mount` is not already mounted, it must find the VFS inode of the directory that is to be the new file system's mount point. This VFS inode may be in the inode cache or it might have to be read from the block device supporting the file system of the mount point. Once the inode has been found it is checked to see that it is a directory and that there is not already some other file system mounted there. The same directory cannot be used as a mount point for more than one file system.

At this point the VFS mount code must allocate a VFS superblock and pass it the mount information to the superblock read routine for this file system. All of the system's VFS superblocks are kept in the `super_blocks` vector of `super_block` data structures and one must be allocated for this mount. The superblock read routine must fill out the VFS superblock fields based on information that it reads from the physical device. For the EXT2 file system this mapping or translation of information is quite easy, it simply reads the EXT2 superblock and fills out the VFS superblock from there. For other file systems, such as the MS DOS file system, it is not quite such an easy task. Whatever the file system, filling out the VFS superblock means that the file system must read whatever describes it from the block device that supports it. If the block device cannot be read from or if it does not contain this type of file system then the `mount` command will fail.

Each mounted file system is described by a `vfsmount` data structure; see figure 9.6. These are queued on a list pointed at by `vfsmntlist`. Another pointer, `vfsmnttail` points at the last entry in the list and the `mr_u_vfsmnt` pointer points at the most recently used file system. Each `vfsmount` structure contains the device number of the block device holding the file system, the directory where this file system is mounted and a pointer to the VFS superblock allocated when this file system was mounted. In turn the VFS superblock points at the `file_system_type` data structure for this sort of file system and to the root inode for this file system. This inode is kept resident in the VFS inode cache all of the time that this file system is loaded.

See
`add_vfsmnt()` in
`fs/super.c`

9.2.5 Finding a File in the Virtual File System

To find the VFS inode of a file in the Virtual File System, VFS must resolve the name a directory at a time, looking up the VFS inode representing each of the intermediate directories in the name. Each directory lookup involves calling the file system specific lookup whose address is held in the VFS inode representing the parent directory. This

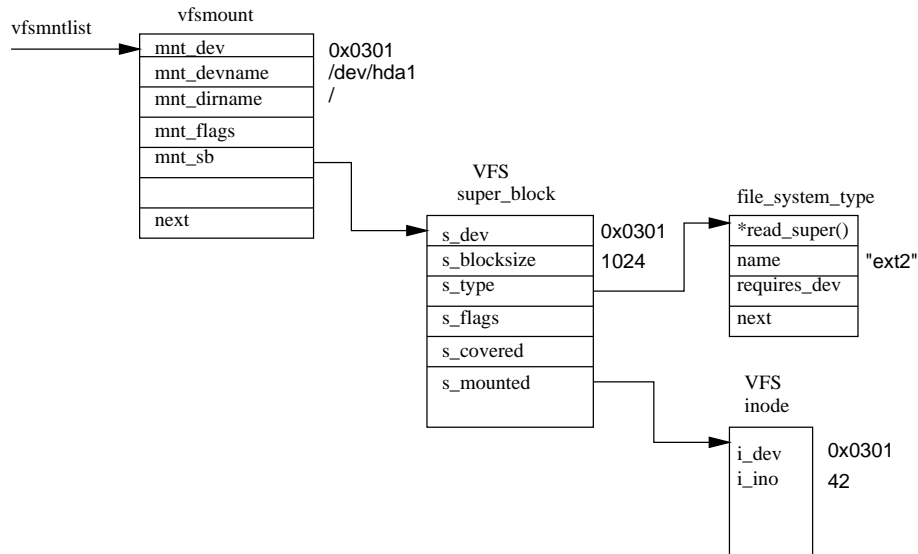


Figure 9.6: A Mounted File System

works because we always have the VFS inode of the root of each file system available and pointed at by the VFS superblock for that system. Each time an inode is looked up by the real file system it checks the directory cache for the directory. If there is no entry in the directory cache, the real file system gets the VFS inode either from the underlying file system or from the inode cache.

9.2.6 Creating a File in the Virtual File System

9.2.7 Unmounting a File System

See `do_umount()`
in
`fs/super.c`

The workshop manual for my MG usually describes assembly as the reverse of disassembly and the reverse is more or less true for unmounting a file system. A file system cannot be unmounted if something in the system is using one of its files. So, for example, you cannot unmount `/mnt/cdrom` if a process is using that directory or any of its children. If anything is using the file system to be unmounted there may be VFS inodes from it in the VFS inode cache, and the code checks for this by looking through the list of inodes looking for inodes owned by the device that this file system occupies. If the VFS superblock for the mounted file system is dirty, that is it has been modified, then it must be written back to the file system on disk. Once it has been written to disk, the memory occupied by the VFS superblock is returned to the kernel's free pool of memory. Finally the `vsmount` data structure for this mount is unlinked from `vsmntlist` and freed.

See
`remove_vfsmnt()`
in
`fs/super.c`

9.2.8 The VFS Inode Cache

As the mounted file systems are navigated, their VFS inodes are being continually read and, in some cases, written. The Virtual File System maintains an inode cache to speed up accesses to all of the mounted file systems. Every time a VFS inode is read from the inode cache the system saves an access to a physical device.

See
`fs/inode.c`

The VFS inode cache is implemented as a hash table whose entries are pointers to

lists of VFS inodes that have the same hash value. The hash value of an inode is calculated from its inode number and from the device identifier for the underlying physical device containing the file system. Whenever the Virtual File System needs to access an inode, it first looks in the VFS inode cache. To find an inode in the cache, the system first calculates its hash value and then uses it as an index into the inode hash table. This gives it a pointer to a list of inodes with the same hash value. It then reads each inode in turn until it finds one with both the same inode number and the same device identifier as the one that it is searching for.

If it can find the inode in the cache, its count is incremented to show that it has another user and the file system access continues. Otherwise a free VFS inode must be found so that the file system can read the inode from memory. VFS has a number of choices about how to get a free inode. If the system may allocate more VFS inodes then this is what it does; it allocates kernel pages and breaks them up into new, free, inodes and puts them into the inode list. All of the system's VFS inodes are in a list pointed at by `first_inode` as well as in the inode hash table. If the system already has all of the inodes that it is allowed to have, it must find an inode that is a good candidate to be reused. Good candidates are inodes with a usage count of zero; this indicates that the system is not currently using them. Really important VFS inodes, for example the root inodes of file systems always have a usage count greater than zero and so are never candidates for reuse. Once a candidate for reuse has been located it is cleaned up. The VFS inode might be dirty and in this case it needs to be written back to the file system or it might be locked and in this case the system must wait for it to be unlocked before continuing. The candidate VFS inode must be cleaned up before it can be reused.

However the new VFS inode is found, a file system specific routine must be called to fill it out from information read from the underlying real file system. Whilst it is being filled out, the new VFS inode has a usage count of one and is locked so that nothing else accesses it until it contains valid information.

To get the VFS inode that is actually needed, the file system may need to access several other inodes. This happens when you read a directory; only the inode for the final directory is needed but the inodes for the intermediate directories must also be read. As the VFS inode cache is used and filled up, the less used inodes will be discarded and the more used inodes will remain in the cache.

9.2.9 The Directory Cache

To speed up accesses to commonly used directories, the VFS maintains a cache of directory entries. As directories are looked up by the real file systems their details are added into the directory cache. The next time the same directory is looked up, for example to list it or open a file within it, then it will be found in the directory cache. Only short directory entries (up to 15 characters long) are cached but this is reasonable as the shorter directory names are the most commonly used ones. For example, `/usr/X11R6/bin` is very commonly accessed when the X server is running.

The directory cache consists of a hash table, each entry of which points at a list of directory cache entries that have the same hash value. The hash function uses the device number of the device holding the file system and the directory's name to calculate the offset, or index, into the hash table. It allows cached directory entries to be quickly found. It is no use having a cache when lookups within the cache take

See <code>fs/dcache.c</code>

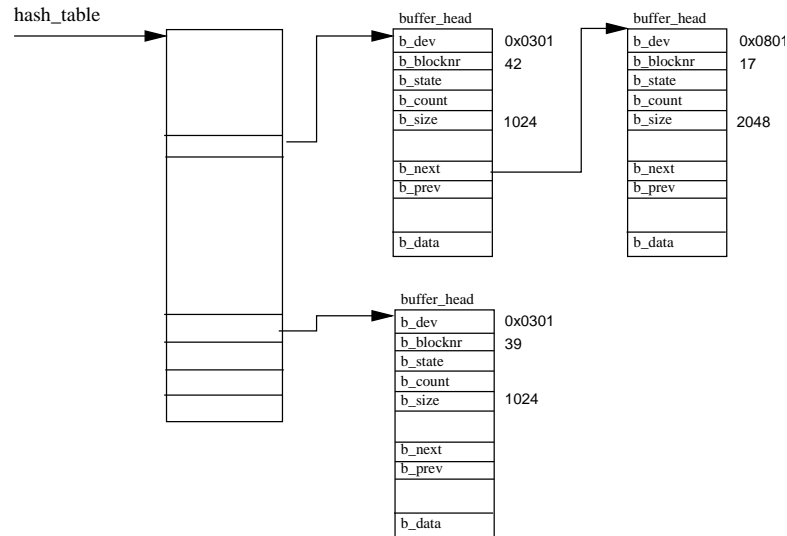


Figure 9.7: The Buffer Cache

too long to find entries, or even not to find them.

In an effort to keep the caches valid and up to date the VFS keeps lists of Least Recently Used (LRU) directory cache entries. When a directory entry is first put into the cache, which is when it is first looked up, it is added onto the end of the first level LRU list. In a full cache this will displace an existing entry from the front of the first LRU list. As the directory entry is accessed again it is promoted to the back of the second LRU cache list. Again, this may displace a cached level two directory entry at the front of the level two LRU cache list. This displacing of entries at the front of the level one and level two LRU lists is fine. The only reason that entries are at the front of the lists is that they have not been recently accessed. If they had, they would be nearer the back of the lists. The entries in the second level LRU cache list are safer than entries in the level one LRU cache list. This is the intention as these entries have not only been looked up but also they have been repeatedly referenced.

REVIEW NOTE: *Do we need a diagram for this?*

9.3 The Buffer Cache

As the mounted file systems are used they generate a lot of requests to the block devices to read and write data blocks. All block data read and write requests are given to the device drivers in the form of `buffer_head` data structures via standard kernel routine calls. These give all of the information that the block device drivers need; the device identifier uniquely identifies the device and the block number tells the driver which block to read. All block devices are viewed as linear collections of blocks of the same size. To speed up access to the physical block devices, Linux maintains a cache of block buffers. All of the block buffers in the system are kept somewhere in this buffer cache, even the new, unused buffers. This cache is shared between all of the physical block devices; at any one time there are many block buffers in the cache, belonging to any one of the system's block devices and often in many different states. If valid data is available from the buffer cache this saves the

system an access to a physical device. Any block buffer that has been used to read data from a block device or to write data to it goes into the buffer cache. Over time it may be removed from the cache to make way for a more deserving buffer or it may remain in the cache as it is frequently accessed.

Block buffers within the cache are uniquely identified by the owning device identifier and the block number of the buffer. The buffer cache is composed of two functional parts. The first part is the lists of free block buffers. There is one list per supported buffer size and the system's free block buffers are queued onto these lists when they are first created or when they have been discarded. The currently supported buffer sizes are 512, 1024, 2048, 4096 and 8192 bytes. The second functional part is the cache itself. This is a hash table which is a vector of pointers to chains of buffers that have the same hash index. The hash index is generated from the owning device identifier and the block number of the data block. Figure 9.7 shows the hash table together with a few entries. Block buffers are either in one of the free lists or they are in the buffer cache. When they are in the buffer cache they are also queued onto Least Recently Used (LRU) lists. There is an LRU list for each buffer type and these are used by the system to perform work on buffers of a type, for example, writing buffers with new data in them out to disk. The buffer's type reflects its state and Linux currently supports the following types:

clean Unused, new buffers,

locked Buffers that are locked, waiting to be written,

dirty Dirty buffers. These contain new, valid data, and will be written but so far have not been scheduled to write,

shared Shared buffers,

unshared Buffers that were once shared but which are now not shared,

Whenever a file system needs to read a buffer from its underlying physical device, it tries to get a block from the buffer cache. If it cannot get a buffer from the buffer cache, then it will get a clean one from the appropriate sized free list and this new buffer will go into the buffer cache. If the buffer that it needed is in the buffer cache, then it may or may not be up to date. If it is not up to date or if it is a new block buffer, the file system must request that the device driver read the appropriate block of data from the disk.

Like all caches, the buffer cache must be maintained so that it runs efficiently and fairly allocates cache entries between the block devices using the buffer cache. Linux uses the `bdflush` kernel daemon to perform a lot of housekeeping duties on the cache but some happen automatically as a result of the cache being used.

9.3.1 The `bdflush` Kernel Daemon

The `bdflush` kernel daemon is a simple kernel daemon that provides a dynamic response to the system having too many dirty buffers; buffers that contain data that must be written out to disk at some time. It is started as a kernel thread at system startup time and, rather confusingly, it calls itself “`kflushd`” and that is the name that you will see if you use the `ps` command to show the processes in the system. Mostly this daemon sleeps waiting for the number of dirty buffers in the system to

See `bdflush()`
in
`fs/buffer.c`

grow too large. As buffers are allocated and discarded the number of dirty buffers in the system is checked. If there are too many as a percentage of the total number of buffers in the system then `bdflush` is woken up. The default threshold is 60% but, if the system is desperate for buffers, `bdflush` will be woken up anyway. This value can be seen and changed using the `update` command:

```
# update -d
```

```
bdflush version 1.4
```

```
0:    60 Max fraction of LRU list to examine for dirty blocks
1:    500 Max number of dirty blocks to write each time bdflush activated
2:    64 Num of clean buffers to be loaded onto free list by refill_freelist
3:    256 Dirty block threshold for activating bdflush in refill_freelist
4:    15 Percentage of cache to scan for free clusters
5:   3000 Time for data buffers to age before flushing
6:    500 Time for non-data (dir, bitmap, etc) buffers to age before flushing
7:   1884 Time buffer cache load average constant
8:     2 LAV ratio (used to determine threshold for buffer fratricide).
```

All of the dirty buffers are linked into the `BUF_DIRTY` LRU list whenever they are made dirty by having data written to them and `bdflush` tries to write a reasonable number of them out to their owning disks. Again this number can be seen and controlled by the `update` command and the default is 500 (see above).

9.3.2 The update Process

The `update` command is more than just a command; it is also a daemon. When run as superuser (during system initialisation) it will periodically flush all of the older dirty buffers out to disk. It does this by calling a system service routine that does more or less the same thing as `bdflush`. Whenever a dirty buffer is finished with, it is tagged with the system time that it should be written out to its owning disk. Every time that `update` runs it looks at all of the dirty buffers in the system looking for ones with an expired flush time. Every expired buffer is written out to disk.

See
`sys_bdflush()` in
`fs/buffer.c`

9.4 The /proc File System

The `/proc` file system really shows the power of the Linux Virtual File System. It does not really exist (yet another of Linux's conjuring tricks), neither the `/proc` directory nor its subdirectories and its files actually exist. So how can you `cat /proc/devices`? The `/proc` file system, like a real file system, registers itself with the Virtual File System. However, when the VFS makes calls to it requesting inodes as its files and directories are opened, the `/proc` file system creates those files and directories from information within the kernel. For example, the kernel's `/proc/devices` file is generated from the kernel's data structures describing its devices.

The `/proc` file system presents a user readable window into the kernel's inner workings. Several Linux subsystems, such as Linux kernel modules described in chapter 12, create entries in the the `/proc` file system.

9.5 Device Special Files

Linux, like all versions of Unix™ presents its hardware devices as special files. So, for example, `/dev/null` is the null device. A device file does not use any data space in the file system, it is only an access point to the device driver. The EXT2 file system and the Linux VFS both implement device files as special types of inode. There are two types of device file; character and block special files. Within the kernel itself, the device drivers implement file semantics: you can open them, close them and so on. Character devices allow I/O operations in character mode and block devices require that all I/O is via the buffer cache. When an I/O request is made to a device file, it is forwarded to the appropriate device driver within the system. Often this is not a real device driver but a pseudo-device driver for some subsystem such as the SCSI device driver layer. Device files are referenced by a major number, which identifies the device type, and a minor type, which identifies the unit, or instance of that major type. For example, the IDE disks on the first IDE controller in the system have a major number of 3 and the first partition of an IDE disk would have a minor number of 1. So, `ls -l` of `/dev/hda1` gives:

```
$ brw-rw---- 1 root disk 3, 1 Nov 24 15:09 /dev/hda1
```

see
`/include/linux/
major.h` for all of
Linux's major
device numbers.

Within the kernel, every device is uniquely described by a `kdev_t` data type, this is two bytes long, the first byte containing the minor device number and the second byte holding the major device number. The IDE device above is held within the kernel as `0x0301`. An EXT2 inode that represents a block or character device keeps the device's major and minor numbers in its first direct block pointer. When it is read by the VFS, the VFS inode data structure representing it has its `i_rdev` field set to the correct device identifier.

See
`include/linux/-
kdev_t.h`

