Chapter 12. The Virtual Filesystem

One of Linux's keys to success is its ability to coexist comfortably with other systems. You can transparently mount disks or partitions that host file formats used by Windows , other Unix systems, or even systems with tiny market shares like the Amiga. Linux manages to support multiple filesystem types in the same way other Unix variants do, through a concept called the Virtual Filesystem.

The idea behind the Virtual Filesystem is to put a wide range of information in the kernel to represent many different types of filesystems ; there is a field or function to support each operation provided by all real filesystems supported by Linux. For each read, write, or other function called, the kernel substitutes the actual function that supports a native Linux filesystem, the NTFS filesystem, or whatever other filesystem the file is on.

This chapter discusses the aims, structure, and implementation of Linux's Virtual Filesystem. It focuses on three of the five standard Unix file typesnamely, regular files, directories, and symbolic links. Device files are covered in Chapter 13, while pipes are discussed in Chapter 19. To show how a real filesystem works, Chapter 18 covers the Second Extended Filesystem that appears on nearly all Linux systems.

12.1. The Role of the Virtual Filesystem (VFS)

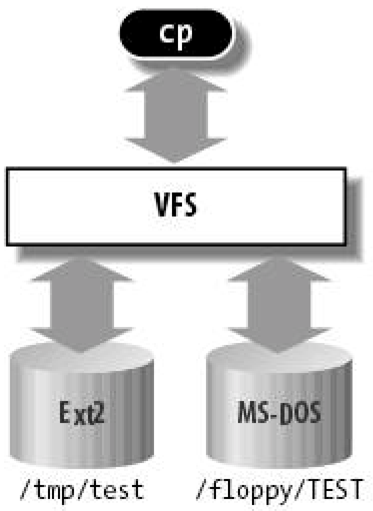
The *Virtual Filesystem* (also known as Virtual Filesystem Switch or *VFS*) is a kernel software layer that handles all system calls related to a standard Unix filesystem. Its main strength is providing a common interface to several kinds of filesystems.

For instance, let's assume that a user issues the shell command:

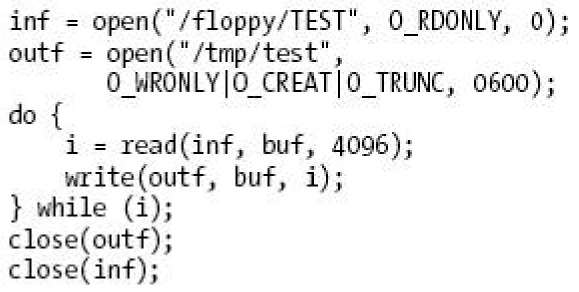
```
$ cp /floppy/TEST /tmp/test
```

where */floppy* is the mount point of an MS-DOS diskette and */tmp* is a normal Second Extended Filesystem (Ext2) directory. The VFS is an abstraction layer between the application program and the filesystem implementations (see Figure 12-1(a)). Therefore, the *cp* program is not required to know the filesystem types of */floppy/TEST* and */tmp/test*. Instead, *cp* interacts with the VFS by means of generic system calls known to anyone who has done Unix programming (see the section "File-Handling System Calls" in Chapter 1); the code executed by *cp* is shown in Figure 12-1(b).

Figure 12-1. VFS role in a simple file copy operation



(a)



```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

Filesystems supported by the VFS may be grouped into three main classes:

Disk-based filesystems

These manage memory space available in a local disk or in some other device that emulates a disk (such as a USB flash drive). Some of the well-known disk-based filesystems supported by the VFS are:

- Filesystems for Linux such as the widely used Second Extended Filesystem (Ext2), the recent Third Extended Filesystem (Ext3), and the Reiser Filesystems (ReiserFS)[*]

  [*] Although these filesystems owe their birth to Linux, they have been ported to several other operating systems.

- Filesystems for Unix variants such as sysv filesystem (System V , Coherent , Xenix), UFS (BSD , Solaris , NEXTSTEP), MINIX filesystem, and VERITAS VxFS (SCO UnixWare)
- Microsoft filesystems such as MS-DOS, VFAT (Windows 95 and later releases), and NTFS (Windows NT 4 and later releases)
- ISO9660 CD-ROM filesystem (formerly High Sierra Filesystem) and Universal Disk Format (UDF) DVD filesystem
- Other proprietary filesystems such as IBM's OS/2 (HPFS), Apple's Macintosh (HFS), Amiga's Fast Filesystem (AFFS), and Acorn Disk Filing System (ADFS)
- Additional journaling filesystems originating in systems other than Linux such as IBM's JFS and SGI's XFS

Network filesystems

These allow easy access to files included in filesystems belonging to other networked computers. Some well-known network filesystems supported by the VFS are NFS , Coda , AFS (Andrew filesystem), CIFS (Common Internet File System, used in Microsoft Windows), and NCP (Novell's NetWare Core Protocol).

Special filesystems

These do not manage disk space, either locally or remotely. The */proc* filesystem is a typical example of a special filesystem (see the later section "Special Filesystems").

In this book, we describe in detail the Ext2 and Ext3 filesystems only (see Chapter 18); the other filesystems are not covered for lack of space.

As mentioned in the section "An Overview of the Unix Filesystem" in Chapter 1, Unix directories build a tree whose root is the / directory. The root directory is contained in the *root filesystem*, which in Linux, is usually of type Ext2 or Ext3. All other filesystems can be "mounted" on subdirectories of the root filesystem.[*]

[*] When a filesystem is mounted on a directory, the contents of the directory in the parent filesystem are no longer accessible, because every pathname, including the mount point, will refer to the mounted filesystem. However, the original directory's content shows up again when the filesystem is unmounted. This somewhat surprising feature of Unix filesystems is used by system administrators to hide files; they simply mount a filesystem on the directory containing the files to be hidden.

A disk-based filesystem is usually stored in a hardware block device such as a hard disk, a floppy, or a CD-ROM. A useful feature of Linux's VFS allows it to handle *virtual block devices* such as */dev/loop0*, which may be used to mount filesystems stored in regular files. As a possible application, a user may protect her own private filesystem by storing an encrypted version of it in a regular file.

The first Virtual Filesystem was included in Sun Microsystems's SunOS in 1986. Since then, most Unix filesystems include a VFS. Linux's VFS, however, supports the widest range of filesystems.

12.1.1. The Common File Model

The key idea behind the VFS consists of introducing a *common file model* capable of representing all supported filesystems. This model strictly mirrors the file model provided by the traditional Unix filesystem. This is not surprising, because Linux wants to run its native filesystem with minimum overhead. However, each specific filesystem implementation must translate its physical organization into the VFS's common file model.

For instance, in the common file model, each directory is regarded as a file, which contains a list of files and other directories. However, several non-Unix disk-based filesystems use a File Allocation Table (FAT), which stores the position of each file in the directory tree. In these filesystems, directories are not files. To stick to the VFS's common file model, the Linux implementations of such FAT-based filesystems must be able to construct on the fly, when needed, the files corresponding to the directories. Such files exist only as objects in kernel memory.

More essentially, the Linux kernel cannot hardcode a particular function to handle an operation such as `read( )` or `ioctl( )` . Instead, it must use a pointer for each operation; the pointer is made to point to the proper function for the particular filesystem being accessed.

Let's illustrate this concept by showing how the `read( )` shown in Figure 12-1 would be translated by the kernel into a call specific to the MS-DOS filesystem. The application's call to `read( )` makes the kernel invoke the corresponding `sys_read( )` service routine, like every other system call. The file is represented by a `file` data structure in kernel memory, as we'll see later in this chapter. This data structure contains a field called `f_op` that contains pointers to functions specific to MS-DOS files, including a function that reads a file. `sys_read( )` finds the pointer to this function and invokes it. Thus, the application's `read( )` is turned into the rather indirect call:

```
file->f_op->read(...);
```

Similarly, the `write( )` operation triggers the execution of a proper Ext2 write function associated with the output file. In short, the kernel is responsible for assigning the right set of pointers to the `file` variable associated with each open file, and then for invoking the call specific to each filesystem that the `f_op` field points to.

One can think of the common file model as object-oriented, where an *object* is a software construct that defines both a data structure and the methods that operate on it. For reasons of efficiency, Linux is not coded in an object-oriented language such as C++. Objects are therefore implemented as plain C data structures with some fields pointing to functions that correspond to the object's methods.

The common file model consists of the following object types:

The superblock object

> Stores information concerning a mounted filesystem. For disk-based filesystems, this object usually corresponds to a *filesystem control block* stored on disk.

The inode object

> Stores general information about a specific file. For disk-based filesystems, this object usually corresponds to a *file control block* stored on disk. Each

inode object is associated with an *inode number*, which uniquely identifies the file within the filesystem.

The file object

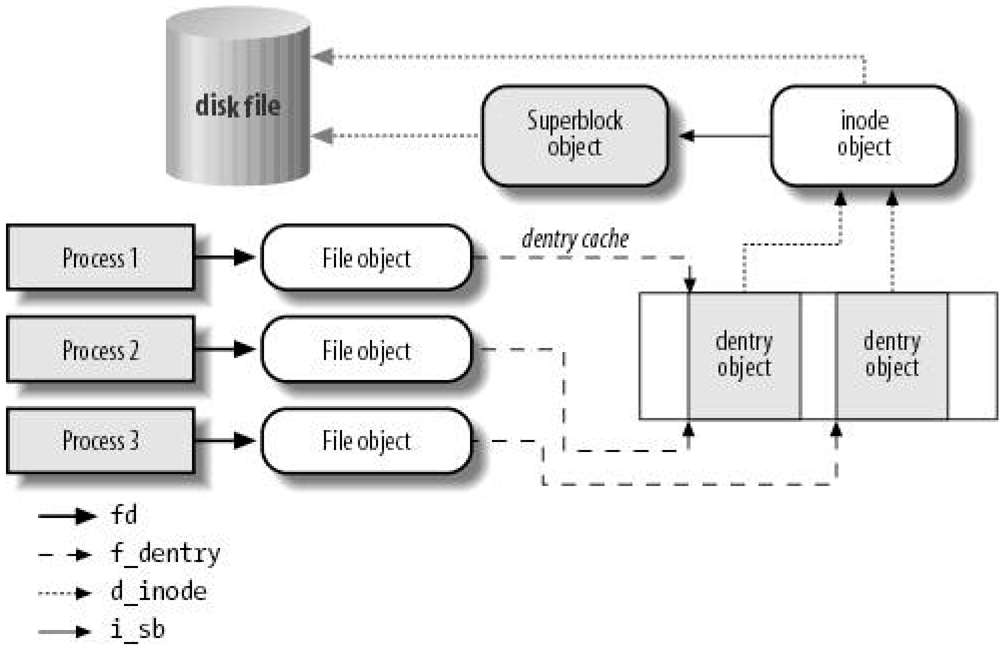
Stores information about the interaction between an open file and a process. This information exists only in kernel memory during the period when a process has the file open.

The dentry object

Stores information about the linking of a directory entry (that is, a particular name of the file) with the corresponding file. Each disk-based filesystem stores this information in its own particular way on disk.

Figure 12-2 illustrates with a simple example how processes interact with files. Three different processes have opened the same file, two of them using the same hard link. In this case, each of the three processes uses its own file object, while only two dentry objects are requiredone for each hard link. Both dentry objects refer to the same inode object, which identifies the superblock object and, together with the latter, the common disk file.

Figure 12-2. Interaction between processes and VFS objects



Besides providing a common interface to all filesystem implementations, the VFS has another important role related to system performance. The most recently used dentry objects are contained in a disk cache named the *dentry cache* , which speeds up the translation from a file pathname to the inode of the last pathname component.

Generally speaking, a *disk cache* is a software mechanism that allows the kernel to keep in RAM some information that is normally stored on a disk, so that further accesses to that data can be quickly satisfied without a slow access to the disk itself.

Notice how a disk cache differs from a hardware cache or a memory cache, neither of which has anything to do with disks or other devices. A hardware cache is a fast static RAM that speeds up requests directed to the slower dynamic RAM (see the section "Hardware Cache" in Chapter 2). A memory cache is a software mechanism introduced to bypass the Kernel Memory Allocator (see the section "The Slab Allocator" in Chapter 8).

Beside the dentry cache and the inode cache, Linux uses other disk caches. The most important one, called the page cache, is described in detail in Chapter 15.

12.1.2. System Calls Handled by the VFS

Table 12-1 illustrates the VFS system calls that refer to filesystems, regular files, directories, and symbolic links. A few other system calls handled by the VFS, such as `ioperm( )` , `ioctl( )` , `pipe( )` , and `mknod( )` , refer to device files and pipes. These are discussed in later chapters. A last group of system calls handled by the VFS, such as `socket( )` , `connect( )` , and `bind( )` , refer to sockets and are used to implement networking. Some of the kernel service routines that correspond to the system calls listed in Table 12-1 are discussed either in this chapter or in Chapter 18.

Table 12-1. Some system calls handled by the VFS

System call name	Description
`mount( )` `umount( )` `umount2( )`	Mount/unmount filesystems
`sysfs( )`	Get filesystem information
`statfs( )` `fstatfs( )` `statfs64( )` `fstatfs64( )` `ustat( )`	Get filesystem statistics
`chroot( )` `pivot_root( )`	Change root directory
`chdir( )` `fchdir( )` `getcwd( )`	Manipulate current directory
`mkdir( )` `rmdir( )`	Create and destroy directories
`getdents( )` `getdents64( )` `readdir( )` `link( )` `unlink( )` `rename( )` `lookup_dcookie( )`	Manipulate directory entries
`readlink( )` `symlink( )`	Manipulate soft links
`chown( )` `fchown( )` `lchown( )` `chown16( )` `fchown16( )` `lchown16( )`	Modify file owner
`chmod( )` `fchmod( )` `utime( )`	Modify file attributes

Table 12-1. Some system calls handled by the VFS

System call name	**Description**
`stat( )` `fstat( )` `lstat( )` `access( )` `oldstat( )` `oldfstat( )` `oldlstat( )` `stat64( )` `lstat64( )` `fstat64( )`	Read file status
`open( )` `close( )` `creat( )` `umask( )`	Open, close, and create files
`dup( )` `dup2( )` `fcntl( )` `fcntl64( )`	Manipulate file descriptors
`select( )` `poll( )`	Wait for events on a set of file descriptors
`truncate( )` `ftruncate( )` `truncate64( )` `ftruncate64( )`	Change file size
`lseek( )` `_llseek( )`	Change file pointer
`read( )` `write( )` `readv( )` `writev( )` `sendfile( )` `sendfile64( )` `readahead( )`	Carry out file I/O operations
io_setup( ) io_submit( ) io_getevents( ) io_cancel( ) io_destroy( )	Asynchronous I/O (allows multiple outstanding read and write requests)
`pread64( )` pwrite64( )	Seek file and access it
`mmap( )` mmap2( ) munmap( ) madvise( ) mincore( ) `remap_file_pages( )`	Handle file memory mapping
`fdatasync( )` fsync( ) sync( ) msync( )	Synchronize file data
`flock( )`	Manipulate file lock
setxattr( ) lsetxattr( ) fsetxattr( ) getxattr( ) lgetxattr( ) fgetxattr( ) listxattr( ) llistxattr( ) flistxattr( ) removexattr( ) lremovexattr( ) fremovexattr( )	Manipulate file extended attributes

We said earlier that the VFS is a layer between application programs and specific filesystems. However, in some cases, a file operation can be performed by the VFS itself, without invoking a lower-level procedure. For instance, when a process closes an open file, the file on disk doesn't usually need to be touched, and hence the VFS simply releases the corresponding file object. Similarly, when the `lseek( )` system call modifies a file pointer, which is an attribute related to the interaction between an opened file and a process, the VFS needs to modify only the corresponding file object without accessing the file on disk, and therefore it does not have to invoke a specific filesystem procedure. In some sense, the VFS could be considered a "generic" filesystem that relies, when necessary, on specific ones.

12.2. VFS Data Structures

Each VFS object is stored in a suitable data structure, which includes both the object attributes and a pointer to a table of object methods. The kernel may dynamically modify the methods of the object and, hence, it may install specialized behavior for the object. The following sections explain the VFS objects and their interrelationships in detail.

12.2.1. Superblock Objects

A superblock object consists of a `super_block` structure whose fields are described in Table 12-2.

Table 12-2. The fields of the superblock object

Type	Field	Description
`struct list_head`	`s_list`	Pointers for superblock list
`dev_t`	`s_dev`	Device identifier
`unsigned long`	`s_blocksize`	Block size in bytes
`unsigned long`	`s_old_blocksize`	Block size in bytes as reported by the underlying block device driver
`unsigned char`	`s_blocksize_bits`	Block size in number of bits
`unsigned char`	`s_dirt`	Modified (dirty) flag
`unsigned long long`	`s_maxbytes`	Maximum size of the files
`struct file_system_type *`	`s_type`	Filesystem type
`struct super_operations *`	`s_op`	Superblock methods
`struct dquot_operations *`	`dq_op`	Disk quota handling methods
struct quotactl_ops *	s_qcop	Disk quota administration methods
struct export_operations *	s_export_op	Export operations used by network filesystems
`unsigned long`	`s_flags`	Mount flags
`unsigned long`	`s_magic`	Filesystem magic number
`struct dentry *`	`s_root`	Dentry object of the filesystem's root directory
`struct rw_semaphore`	`s_umount`	Semaphore used for unmounting
`struct semaphore`	`s_lock`	Superblock semaphore
`int`	`s_count`	Reference counter

Table 12-2. The fields of the superblock object

Type	Field	Description
int	s_syncing	Flag indicating that inodes of the superblock are being synchronized
int	s_need_sync_fs	Flag used when synchronizing the superblock's mounted filesystem
atomic_t	s_active	Secondary reference counter
void *	s_security	Pointer to superblock security structure
struct xattr_handler **	s_xattr	Pointer to superblock extended attribute structure
struct list_head	s_inodes	List of all inodes
struct list_head	s_dirty	List of modified inodes
struct list_head	s_io	List of inodes waiting to be written to disk
struct hlist_head	s_anon	List of anonymous dentries for handling remote network filesystems
struct list_head	s_files	List of file objects
struct block_device *	s_bdev	Pointer to the block device driver descriptor
struct list_head	s_instances	Pointers for a list of superblock objects of a given filesystem type (see the later section "Filesystem Type Registration")
struct quota_info	s_dquot	Descriptor for disk quota
int	s_frozen	Flag used when freezing the filesystem (forcing it to a consistent state)
wait_queue_head_t	s_wait_unfrozen	Wait queue where processes sleep until the filesystem is unfrozen
char[]	s_id	Name of the block device containing the superblock
void *	s_fs_info	Pointer to superblock information of a specific filesystem
struct semaphore	s_vfs_rename_sem	Semaphore used by VFS when renaming files across directories
u32	s_time_gran	Timestamp's granularity (in nanoseconds)

All superblock objects are linked in a circular doubly linked list. The first element of this list is represented by the `super_blocks` variable, while the `s_list` field of the superblock object stores the pointers to the adjacent elements in the list. The `sb_lock` spin lock protects the list against concurrent accesses in multiprocessor systems.

The `s_fs_info` field points to superblock information that belongs to a specific filesystem; for instance, as we'll see later in Chapter 18, if the superblock object refers to an Ext2 filesystem, the field points to an `ext2_sb_info` structure, which includes the disk allocation bit masks and other data of no concern to the VFS common file model.

In general, data pointed to by the `s_fs_info` field is information from the disk duplicated in memory for reasons of efficiency. Each disk-based filesystem needs to access and update its allocation bitmaps in order to allocate or release disk blocks. The VFS allows these filesystems to act directly on the `s_fs_info` field of the superblock in memory without accessing the disk.

This approach leads to a new problem, however: the VFS superblock might end up no longer synchronized with the corresponding superblock on disk. It is thus necessary to introduce an `s_dirt` flag, which specifies whether the superblock is dirtythat is, whether the data on the disk must be updated. The lack of synchronization leads to the familiar problem of a corrupted filesystem when a site's power goes down without giving the user the chance to shut down a system cleanly. As we'll see in the section "Writing Dirty Pages to Disk" in Chapter 15, Linux minimizes this problem by periodically copying all dirty superblocks to disk.

The methods associated with a superblock are called *superblock operations* . They are described by the `super_operations` structure whose address is included in the `s_op` field.

Each specific filesystem can define its own superblock operations. When the VFS needs to invoke one of them, say `read_inode( )`, it executes the following:

```
    sb->s_op->read_inode(inode);
```

where `sb` stores the address of the superblock object involved. The `read_inode` field of the `super_operations` table contains the address of the suitable function, which is therefore directly invoked.

Let's briefly describe the superblock operations, which implement higher-level operations like deleting files or mounting disks. They are listed in the order they appear in the `super_operations` table:

`alloc_inode(sb)`

Allocates space for an inode object, including the space required for filesystem-specific data.

`destroy_inode(inode)`

Destroys an inode object, including the filesystem-specific data.

`read_inode(inode)`

Fills the fields of the inode object passed as the parameter with the data on disk; the `i_ino` field of the inode object identifies the specific filesystem inode on the disk to be read.

`dirty_inode(inode)`

Invoked when the inode is marked as modified (dirty). Used by filesystems such as ReiserFS and Ext3 to update the filesystem journal on disk.

`write_inode(inode, flag)`

Updates a filesystem inode with the contents of the inode object passed as the parameter; the `i_ino` field of the inode object identifies the filesystem inode on disk that is concerned. The `flag` parameter indicates whether the I/O operation should be synchronous.

`put_inode(inode)`

Invoked when the inode is releasedits reference counter is decreasedto perform filesystem-specific operations.

`drop_inode(inode)`

Invoked when the inode is about to be destroyedthat is, when the last user releases the inode; filesystems that implement this method usually make use of `generic_drop_inode( )`. This function removes every reference to the inode from the VFS data structures and, if the inode no longer appears in any directory, invokes the `delete_inode` superblock method to delete the inode from the filesystem.

`delete_inode(inode)`

Invoked when the inode must be destroyed. Deletes the VFS inode in memory and the file data and metadata on disk.

`put_super(super)`

Releases the superblock object passed as the parameter (because the corresponding filesystem is unmounted).

`write_super(super)`

Updates a filesystem superblock with the contents of the object indicated.

`sync_fs(sb, wait)`

Invoked when flushing the filesystem to update filesystem-specific data structures on disk (used by journaling filesystems).

`write_super_lockfs(super)`

Blocks changes to the filesystem and updates the superblock with the contents of the object indicated. This method is invoked when the filesystem is frozen, for instance by the Logical Volume Manager (LVM) driver.

`unlockfs(super)`

Undoes the block of filesystem updates achieved by the `write_super_lockfs` superblock method.

`statfs(super, buf)`

Returns statistics on a filesystem by filling the `buf` buffer.

`remount_fs(super, flags, data)`

Remounts the filesystem with new options (invoked when a mount option must be changed).

`clear_inode(inode)`

Invoked when a disk inode is being destroyed to perform filesystem-specific operations.

`umount_begin(super)`

Aborts a mount operation because the corresponding unmount operation has been started (used only by network filesystems).

`show_options(seq_file, vfsmount)`

Used to display the filesystem-specific options

`quota_read(super, type, data, size, offset)`

Used by the quota system to read data from the file that specifies the limits for this filesystem.[*]

[*] The *quota system* defines for each user and/or group limits on the amount of space that can be used on a given filesystem (see the `quotactl()` system call.)

`quota_write(super, type, data, size, offset)`

Used by the quota system to write data into the file that specifies the limits for this filesystem.

The preceding methods are available to all possible filesystem types. However, only a subset of them applies to each specific filesystem; the fields corresponding to unimplemented methods are set to `NULL`. Notice that no `get_super` method to read a superblock is definedhow could the kernel invoke a method of an object yet to be read from disk? We'll find an equivalent `get_sb` method in another object describing the filesystem type (see the later section "Filesystem Type Registration").

12.2.2. Inode Objects

All information needed by the filesystem to handle a file is included in a data structure called an inode. A filename is a casually assigned label that can be changed, but the inode is unique to the file and remains the same as long as the file exists. An inode object in memory consists of an `inode` structure whose fields are described in Table 12-3.

Table 12-3. The fields of the inode object

Type	Field	Description
struct hlist_node	i_hash	Pointers for the hash list
struct list_head	i_list	Pointers for the list that describes the inode's current state
struct list_head	i_sb_list	Pointers for the list of inodes of the superblock
struct list_head	i_dentry	The head of the list of dentry objects referencing this inode
unsigned long	i_ino	inode number
atomic_t	i_count	Usage counter
umode_t	i_mode	File type and access rights
unsigned int	i_nlink	Number of hard links
uid_t	i_uid	Owner identifier
gid_t	i_gid	Group identifier
dev_t	i_rdev	Real device identifier
loff_t	i_size	File length in bytes
struct timespec	i_atime	Time of last file access
struct timespec	i_mtime	Time of last file write
struct timespec	i_ctime	Time of last inode change
unsigned int	i_blkbits	Block size in number of bits
unsigned long	i_blksize	Block size in bytes
unsigned long	i_version	Version number, automatically increased after each use
unsigned long	i_blocks	Number of blocks of the file
unsigned short	i_bytes	Number of bytes in the last block of the file
unsigned char	i_sock	Nonzero if file is a socket
spinlock_t	i_lock	Spin lock protecting some fields of the inode
struct semaphore	i_sem	inode semaphore
struct rw_semaphore	i_alloc_sem	Read/write semaphore protecting against race conditions in direct I/O file operations
struct inode_operations *	i_op	inode operations
struct file_operations *	i_fop	Default file operations
struct super_block *	i_sb	Pointer to superblock object
struct file_lock *	i_flock	Pointer to file lock list

Table 12-3. The fields of the inode object

Type	Field	Description
struct address_space *	i_mapping	Pointer to an address_space object (see Chapter 15)
struct address_space	i_data	address_space object of the file
struct dquot * []	i_dquot	inode disk quotas
struct list_head	i_devices	Pointers for a list of inodes relative to a specific character or block device (see Chapter 13)
struct pipe_inode_info *	i_pipe	Used if the file is a pipe (see Chapter 19)
struct block_device *	i_bdev	Pointer to the block device driver
struct cdev *	i_cdev	Pointer to the character device driver
int	i_cindex	Index of the device file within a group of minor numbers
_ _u32	i_generation	inode version number (used by some filesystems)
unsigned long	i_dnotify_mask	Bit mask of directory notify events
struct dnotify_struct *	i_dnotify	Used for directory notifications
unsigned long	i_state	inode state flags
unsigned long	dirtied_when	Dirtying time (in ticks) of the inode
unsigned int	i_flags	Filesystem mount flags
atomic_t	i_writecount	Usage counter for writing processes
void *	i_security	Pointer to inode's security structure
void *	u.generic_ip	Pointer to private data
seqcount_t	i_size_seqcount	Sequence counter used in SMP systems to get consistent values for i_size

Each inode object duplicates some of the data included in the disk inodefor instance, the number of blocks allocated to the file. When the value of the i_state field is equal to I_DIRTY_SYNC, I_DIRTY_DATASYNC, or I_DIRTY_PAGES, the inode is dirtythat is, the corresponding disk inode must be updated. The I_DIRTY macro can be used to check the value of these three flags at once (see later for details). Other values of the i_state field are I_LOCK (the inode object is involved in an I/O transfer), I_FREEING (the inode object is being freed), I_CLEAR (the inode object contents are no longer meaningful), and I_NEW (the inode object has been allocated but not yet filled with data read from the disk inode).

Each inode object always appears in one of the following circular doubly linked lists (in all cases, the pointers to the adjacent elements are stored in the `i_list` field):

- The list of valid unused inodes, typically those mirroring valid disk inodes and not currently used by any process. These inodes are not dirty and their `i_count` field is set to 0. The first and last elements of this list are referenced by the `next` and `prev` fields, respectively, of the `inode_unused` variable. This list acts as a disk cache.
- The list of in-use inodes, that is, those mirroring valid disk inodes and used by some process. These inodes are not dirty and their `i_count` field is positive. The first and last elements are referenced by the `inode_in_use` variable.
- The list of dirty inodes. The first and last elements are referenced by the `s_dirty` field of the corresponding superblock object.

Each of the lists just mentioned links the `i_list` fields of the proper inode objects.

Moreover, each inode object is also included in a per-filesystem doubly linked circular list headed at the `s_inodes` field of the superblock object; the `i_sb_list` field of the inode object stores the pointers for the adjacent elements in this list.

Finally, the inode objects are also included in a hash table named `inode_hashtable`. The hash table speeds up the search of the inode object when the kernel knows both the inode number and the address of the superblock object corresponding to the filesystem that includes the file. Because hashing may induce collisions, the inode object includes an `i_hash` field that contains a backward and a forward pointer to other inodes that hash to the same position; this field creates a doubly linked list of those inodes.

The methods associated with an inode object are also called *inode operations* . They are described by an `inode_operations` structure, whose address is included in the `i_op` field. Here are the inode operations in the order they appear in the `inode_operations` table:

`create(dir, dentry, mode, nameidata)`

> Creates a new disk inode for a regular file associated with a dentry object in some directory.

`lookup(dir, dentry, nameidata)`

> Searches a directory for an inode corresponding to the filename included in a dentry object.

`link(old_dentry, dir, new_dentry)`

Creates a new hard link that refers to the file specified by `old_dentry` in the directory `dir`; the new hard link has the name specified by `new_dentry`.

`unlink(dir, dentry)`

Removes the hard link of the file specified by a dentry object from a directory.

`symlink(dir, dentry, symname)`

Creates a new inode for a symbolic link associated with a dentry object in some directory.

`mkdir(dir, dentry, mode)`

Creates a new inode for a directory associated with a dentry object in some directory.

`rmdir(dir, dentry)`

Removes from a directory the subdirectory whose name is included in a dentry object.

`mknod(dir, dentry, mode, rdev)`

Creates a new disk inode for a special file associated with a dentry object in some directory. The `mode` and `rdev` parameters specify, respectively, the file type and the device's major and minor numbers.

`rename(old_dir, old_dentry, new_dir, new_dentry)`

Moves the file identified by `old_entry` from the `old_dir` directory to the `new_dir` one. The new filename is included in the dentry object that `new_dentry` points to.

`readlink(dentry, buffer, buflen)`

Copies into a User Mode memory area specified by `buffer` the file pathname corresponding to the symbolic link specified by the dentry.

`follow_link(inode, nameidata)`

Translates a symbolic link specified by an inode object; if the symbolic link is a relative pathname, the lookup operation starts from the directory specified in the second parameter.

`put_link(dentry, nameidata)`

Releases all temporary data structures allocated by the `follow_link` method to translate a symbolic link.

`truncate(inode)`

Modifies the size of the file associated with an inode. Before invoking this method, it is necessary to set the `i_size` field of the inode object to the required new size.

`permission(inode, mask, nameidata)`

Checks whether the specified access mode is allowed for the file associated with `inode`.

`setattr(dentry, iattr)`

Notifies a "change event" after touching the inode attributes.

`getattr(mnt, dentry, kstat)`

Used by some filesystems to read inode attributes.

`setxattr(dentry, name, value, size, flags)`

Sets an "extended attribute" of an inode (extended attributes are stored on disk blocks outside of any inode).

`getxattr(dentry, name, buffer, size)`

Gets an extended attribute of an inode.

`listxattr(dentry, buffer, size)`

Gets the whole list of extended attribute names.

`removexattr(dentry, name)`

Removes an extended attribute of an inode.

The methods just listed are available to all possible inodes and filesystem types. However, only a subset of them applies to a specific inode and filesystem; the fields corresponding to unimplemented methods are set to `NULL`.

12.2.3. File Objects

A file object describes how a process interacts with a file it has opened. The object is created when the file is opened and consists of a `file` structure, whose fields are described in Table 12-4. Notice that file objects have no corresponding image on disk, and hence no "dirty" field is included in the `file` structure to specify that the file object has been modified.

Table 12-4. The fields of the file object

Type	Field	Description
`struct list_head`	`f_list`	Pointers for generic file object list
`struct dentry *`	`f_dentry`	dentry object associated with the file
`struct vfsmount *`	`f_vfsmnt`	Mounted filesystem containing the file
`struct file_operations *`	`f_op`	Pointer to file operation table
`atomic_t`	`f_count`	File object's reference counter
`unsigned int`	`f_flags`	Flags specified when opening the file
`mode_t`	`f_mode`	Process access mode
int	f_error	Error code for network write operation

Table 12-4. The fields of the file object

Type	Field	Description
`loff_t`	`f_pos`	Current file offset (file pointer)
`struct fown_struct`	`f_owner`	Data for I/O event notification via signals
`unsigned int`	`f_uid`	User's UID
`unsigned int`	`f_gid`	User group ID
struct file_ra_state	f_ra	File read-ahead state (see Chapter 16)
size_t	f_maxcount	Maximum number of bytes that can be read or written with a single operation (currently set to 231-1)
`unsigned long`	`f_version`	Version number, automatically increased after each use
void *	f_security	Pointer to file object's security structure
`void *`	`private_data`	Pointer to data specific for a filesystem or a device driver
struct list_head	f_ep_links	Head of the list of event poll waiters for this file
spinlock_t	f_ep_lock	Spin lock protecting the `f_ep_links` list
`struct address_space *`	`f_mapping`	Pointer to file's address space object (see Chapter 15)

The main information stored in a file object is the *file pointer*the current position in the file from which the next operation will take place. Because several processes may access the same file concurrently, the file pointer must be kept in the file object rather than the inode object.

File objects are allocated through a slab cache named *filp*, whose descriptor address is stored in the `filp_cachep` variable. Because there is a limit on the number of file objects that can be allocated, the `files_stat` variable specifies in the `max_files` field the maximum number of allocatable file objectsi.e., the maximum number of files that can be accessed at the same time in the system.[*]

[*] The `files_init( )` function, executed during kernel initialization, sets the `max_files` field to one-tenth of the available RAM in kilobytes, but the system administrator can tune this parameter by writing into the */proc/sys/fs/file-max* file. Moreover, the superuser can always get a file object, even if `max_files` file objects have already been allocated.

"In use" file objects are collected in several lists rooted at the superblocks of the owning filesystems. Each superblock object stores in the `s_files` field the head of a list of file objects; thus, file objects of files belonging to different filesystems are included in different lists. The pointers to the previous and next element in the list are stored in the `f_list` field of the file object. The `files_lock` spin lock protects the superblock `s_files` lists against concurrent accesses in multiprocessor systems.

The `f_count` field of the file object is a reference counter: it counts the number of processes that are using the file object (remember however that lightweight processes created with the `CLONE_FILES` flag share the table that identifies the open files, thus they use the same file objects). The counter is also increased when the file object is used by the kernel itselffor instance, when the object is inserted in a list, or when a `dup( )` system call has been issued.

When the VFS must open a file on behalf of a process, it invokes the `get_empty_filp( )` function to allocate a new file object. The function invokes `kmem_cache_alloc( )` to get a free file object from the *filp* cache, then it initializes the fields of the object as follows:

```
    memset(f, 0, sizeof(*f));
    INIT_LIST_HEAD(&f->f_ep_links);
    spin_lock_init(&f->f_ep_lock);
    atomic_set(&f->f_count, 1);
    f->f_uid = current->fsuid;
    f->f_gid = current->fsgid;
    f->f_owner.lock = RW_LOCK_UNLOCKED;
    INIT_LIST_HEAD(&f->f_list);
    f->f_maxcount = INT_MAX;
```

As we explained earlier in the section "The Common File Model," each filesystem includes its own set of *file operations* that perform such activities as reading and writing a file. When the kernel loads an inode into memory from disk, it stores a pointer to these file operations in a `file_operations` structure whose address is contained in the `i_fop` field of the inode object. When a process opens the file, the VFS initializes the `f_op` field of the new file object with the address stored in the inode so that further calls to file operations can use these functions. If necessary, the VFS may later modify the set of file operations by storing a new value in `f_op`.

The following list describes the file operations in the order in which they appear in the `file_operations` table:

`llseek(file, offset, origin)`

Updates the file pointer.

`read(file, buf, count, offset)`

Reads `count` bytes from a file starting at position `*offset`; the value `*offset` (which usually corresponds to the file pointer) is then increased.

`aio_read(req, buf, len, pos)`

Starts an asynchronous I/O operation to read `len` bytes into `buf` from file position `pos` (introduced to support the `io_submit( )` system call).

`write(file, buf, count, offset)`

Writes `count` bytes into a file starting at position `*offset`; the value `*offset` (which usually corresponds to the file pointer) is then increased.

`aio_write(req, buf, len, pos)`

Starts an asynchronous I/O operation to write `len` bytes from `buf` to file position `pos`.

`readdir(dir, dirent, filldir)`

Returns the next directory entry of a directory in `dirent`; the `filldir` parameter contains the address of an auxiliary function that extracts the fields in a directory entry.

`poll(file, poll_table)`

Checks whether there is activity on a file and goes to sleep until something happens on it.

`ioctl(inode, file, cmd, arg)`

Sends a command to an underlying hardware device. This method applies only to device files.

`unlocked_ioctl(file, cmd, arg)`

Similar to the `ioctl` method, but it does not take the big kernel lock (see the section "The Big Kernel Lock" in Chapter 5). It is expected that all device drivers and all filesystems will implement this new method instead of the `ioctl` method.

`compat_ioctl(file, cmd, arg)`

Method used to implement the `ioctl()` 32-bit system call by 64-bit kernels.

`mmap(file, vma)`

Performs a memory mapping of the file into a process address space (see the section "Memory Mapping" in Chapter 16).

`open(inode, file)`

Opens a file by creating a new file object and linking it to the corresponding inode object (see the section "The open( ) System Call" later in this chapter).

`flush(file)`

Called when a reference to an open file is closed. The actual purpose of this method is filesystem-dependent.

`release(inode, file)`

Releases the file object. Called when the last reference to an open file is closedthat is, when the `f_count` field of the file object becomes 0.

`fsync(file, dentry, flag)`

Flushes the file by writing all cached data to disk.

`aio_fsync(req, flag)`

Starts an asynchronous I/O flush operation.

`fasync(fd, file, on)`

Enables or disables I/O event notification by means of signals.

`lock(file, cmd, file_lock)`

Applies a lock to the file (see the section "File Locking" later in this chapter).

`readv(file, vector, count, offset)`

Reads bytes from a file and puts the results in the buffers described by `vector`; the number of buffers is specified by `count`.

`writev(file, vector, count, offset)`

Writes bytes into a file from the buffers described by `vector`; the number of buffers is specified by `count`.

`sendfile(in_file, offset, count, file_send_actor, out_file)`

Transfers data from `in_file` to `out_file` (introduced to support the `sendfile( )` system call).

`sendpage(file, page, offset, size, pointer, fill)`

Transfers data from `file` to the page cache's `page`; this is a low-level method used by `sendfile( )` and by the networking code for sockets.

`get_unmapped_area(file, addr, len, offset, flags)`

Gets an unused address range to map the file.

`check_flags(flags)`

Method invoked by the service routine of the `fcntl( )` system call to perform additional checks when setting the status flags of a file (`F_SETFL` command). Currently used only by the NFS network filesystem.

`dir_notify(file, arg)`

> Method invoked by the service routine of the `fcntl( )` system call when establishing a directory change notification (`F_NOTIFY` command). Currently used only by the Common Internet File System (CIFS) network filesystem.

`flock(file, flag, lock)`

> Used to customize the behavior of the `flock()` system call. No official Linux filesystem makes use of this method.

The methods just described are available to all possible file types. However, only a subset of them apply to a specific file type; the fields corresponding to unimplemented methods are set to `NULL`.

12.2.4. dentry Objects

We mentioned in the section "The Common File Model" that the VFS considers each directory a file that contains a list of files and other directories. We will discuss in Chapter 18 how directories are implemented on a specific filesystem. Once a directory entry is read into memory, however, it is transformed by the VFS into a dentry object based on the `dentry` structure, whose fields are described in Table 12-5. The kernel creates a dentry object for every component of a pathname that a process looks up; the dentry object associates the component to its corresponding inode. For example, when looking up the */tmp/test* pathname, the kernel creates a dentry object for the / root directory, a second dentry object for the *tmp* entry of the root directory, and a third dentry object for the *test* entry of the */tmp* directory.

Notice that dentry objects have no corresponding image on disk, and hence no field is included in the `dentry` structure to specify that the object has been modified. Dentry objects are stored in a slab allocator cache whose descriptor is `dentry_cache`; dentry objects are thus created and destroyed by invoking `kmem_cache_alloc( )` and `kmem_cache_free( )`.

Table 12-5. The fields of the dentry object

Type	Field	Description
`atomic_t`	`d_count`	Dentry object usage counter
`unsigned int`	`d_flags`	Dentry cache flags
spinlock_t	d_lock	Spin lock protecting the dentry object
`struct inode *`	`d_inode`	Inode associated with filename
`struct dentry *`	`d_parent`	Dentry object of parent directory
`struct qstr`	`d_name`	Filename
`struct list_head`	`d_lru`	Pointers for the list of unused dentries

Table 12-5. The fields of the dentry object

Type	Field	Description
`struct list_head`	`d_child`	For directories, pointers for the list of directory dentries in the same parent directory
`struct list_head`	`d_subdirs`	For directories, head of the list of subdirectory dentries
`struct list_head`	`d_alias`	Pointers for the list of dentries associated with the same inode (alias)
`unsigned long`	`d_time`	Used by `d_revalidate` method
`struct dentry_operations*`	`d_op`	Dentry methods
`struct super_block *`	`d_sb`	Superblock object of the file
`void *`	`d_fsdata`	Filesystem-dependent data
struct rcu_head	d_rcu	The RCU descriptor used when reclaiming the dentry object (see the section "Read-Copy Update (RCU)" in Chapter 5)
struct dcookie_struct *	d_cookie	Pointer to structure used by kernel profilers
`struct hlist_node`	`d_hash`	Pointer for list in hash table entry
`int`	`d_mounted`	For directories, counter for the number of filesystems mounted on this dentry
`unsigned char[]`	`d_iname`	Space for short filename

Each dentry object may be in one of four states:

Free

The dentry object contains no valid information and is not used by the VFS. The corresponding memory area is handled by the slab allocator.

Unused

The dentry object is not currently used by the kernel. The `d_count` usage counter of the object is 0, but the `d_inode` field still points to the associated inode. The dentry object contains valid information, but its contents may be discarded if necessary in order to reclaim memory.

In use

> The dentry object is currently used by the kernel. The `d_count` usage counter is positive, and the `d_inode` field points to the associated inode object. The dentry object contains valid information and cannot be discarded.

Negative

> The inode associated with the dentry does not exist, either because the corresponding disk inode has been deleted or because the dentry object was created by resolving a pathname of a nonexistent file. The `d_inode` field of the dentry object is set to `NULL`, but the object still remains in the dentry cache, so that further lookup operations to the same file pathname can be quickly resolved. The term "negative" is somewhat misleading, because no negative value is involved.

The methods associated with a dentry object are called *dentry operations* ; they are described by the `dentry_operations` structure, whose address is stored in the `d_op` field. Although some filesystems define their own dentry methods, the fields are usually `NULL` and the VFS replaces them with default functions. Here are the methods, in the order they appear in the `dentry_operations` table:

`d_revalidate(dentry, nameidata)`

> Determines whether the dentry object is still valid before using it for translating a file pathname. The default VFS function does nothing, although network filesystems may specify their own functions.

`d_hash(dentry, name)`

> Creates a hash value; this function is a filesystem-specific hash function for the dentry hash table. The `dentry` parameter identifies the directory containing the component. The `name` parameter points to a structure containing both the pathname component to be looked up and the value produced by the hash function.

`d_compare(dir, name1, name2)`

> Compares two filenames ; `name1` should belong to the directory referenced by `dir`. The default VFS function is a normal string match. However, each filesystem can implement this method in its own way. For instance, MS-DOS does not distinguish capital from lowercase letters.

`d_delete(dentry)`

> Called when the last reference to a dentry object is deleted (`d_count` becomes 0). The default VFS function does nothing.

`d_release(dentry)`

> Called when a dentry object is going to be freed (released to the slab allocator). The default VFS function does nothing.

`d_iput(dentry, ino)`

> Called when a dentry object becomes "negative"that is, it loses its inode. The default VFS function invokes `iput( )` to release the inode object.

12.2.5. The dentry Cache

Because reading a directory entry from disk and constructing the corresponding dentry object requires considerable time, it makes sense to keep in memory dentry objects that you've finished with but might need later. For instance, people often edit a file and then compile it, or edit and print it, or copy it and then edit the copy. In such cases, the same file needs to be repeatedly accessed.

To maximize efficiency in handling dentries, Linux uses a dentry cache, which consists of two kinds of data structures:

- A set of dentry objects in the in-use, unused, or negative state.
- A hash table to derive the dentry object associated with a given filename and a given directory quickly. As usual, if the required object is not included in the dentry cache, the search function returns a null value.

The dentry cache also acts as a controller for an *inode cache* . The inodes in kernel memory that are associated with unused dentries are not discarded, because the dentry cache is still using them. Thus, the inode objects are kept in RAM and can be quickly referenced by means of the corresponding dentries.

All the "unused" dentries are included in a doubly linked "Least Recently Used" list sorted by time of insertion. In other words, the dentry object that was last released is put in front of the list, so the least recently used dentry objects are always near the end of the list. When the dentry cache has to shrink, the kernel removes elements from the tail of this list so that the most recently used objects are preserved. The addresses of the first and last elements of the LRU list are stored in the `next` and `prev` fields of the `dentry_unused` variable of type `list_head`. The `d_lru` field of the dentry object contains pointers to the adjacent dentries in the list.

Each "in use" dentry object is inserted into a doubly linked list specified by the i_dentry field of the corresponding inode object (because each inode could be associated with several hard links, a list is required). The d_alias field of the dentry object stores the addresses of the adjacent elements in the list. Both fields are of type struct list_head.

An "in use" dentry object may become "negative" when the last hard link to the corresponding file is deleted. In this case, the dentry object is moved into the LRU list of unused dentries. Each time the kernel shrinks the dentry cache, negative dentries move toward the tail of the LRU list so that they are gradually freed (see the section "Reclaiming Pages of Shrinkable Disk Caches" in Chapter 17).

The hash table is implemented by means of a dentry_hashtable array. Each element is a pointer to a list of dentries that hash to the same hash table value. The array's size usually depends on the amount of RAM installed in the system; the default value is 256 entries per megabyte of RAM. The d_hash field of the dentry object contains pointers to the adjacent elements in the list associated with a single hash value. The hash function produces its value from both the dentry object of the directory and the filename.

The dcache_lock spin lock protects the dentry cache data structures against concurrent accesses in multiprocessor systems. The d_lookup( ) function looks in the hash table for a given parent dentry object and filename; to avoid race conditions, it makes use of a seqlock (see the section "Seqlocks" in Chapter 5). The __d_lookup( ) function is similar, but it assumes that no race condition can happen, so it does not use the seqlock.

12.2.6. Files Associated with a Process

We mentioned in the section "An Overview of the Unix Filesystem" in Chapter 1 that each process has its own current working directory and its own root directory. These are only two examples of data that must be maintained by the kernel to represent the interactions between a process and a filesystem. A whole data structure of type fs_struct is used for that purpose (see Table 12-6), and each process descriptor has an fs field that points to the process fs_struct structure.

Table 12-6. The fields of the fs_struct structure

Type	Field	Description
atomic_t	count	Number of processes sharing this table
rwlock_t	lock	Read/write spin lock for the table fields
int	umask	Bit mask used when opening the file to set the file permissions
struct dentry *	root	Dentry of the root directory
struct dentry *	pwd	Dentry of the current working directory

Table 12-6. The fields of the fs_struct structure

Type	Field	Description
struct dentry *	altroot	Dentry of the emulated root directory (always NULL for the 80 x 86 architecture)
struct vfsmount *	rootmnt	Mounted filesystem object of the root directory
struct vfsmount *	pwdmnt	Mounted filesystem object of the current working directory
struct vfsmount *	altrootmnt	Mounted filesystem object of the emulated root directory (always NULL for the 80 x 86 architecture)

A second table, whose address is contained in the files field of the process descriptor, specifies which files are currently opened by the process. It is a files_struct structure whose fields are illustrated in Table 12-7.

Table 12-7. The fields of the files_struct structure

Type	Field	Description
atomic_t	count	Number of processes sharing this table
rwlock_t	file_lock	Read/write spin lock for the table fields
int	max_fds	Current maximum number of file objects
int	max_fdset	Current maximum number of file descriptors
int	next_fd	Maximum file descriptors ever allocated plus 1
struct file **	fd	Pointer to array of file object pointers
fd_set *	close_on_exec	Pointer to file descriptors to be closed on exec( )
fd_set *	open_fds	Pointer to open file descriptors
fd_set	close_on_exec_init	Initial set of file descriptors to be closed on exec( )
fd_set	open_fds_init	Initial set of file descriptors
struct file *[]	fd_array	Initial array of file object pointers

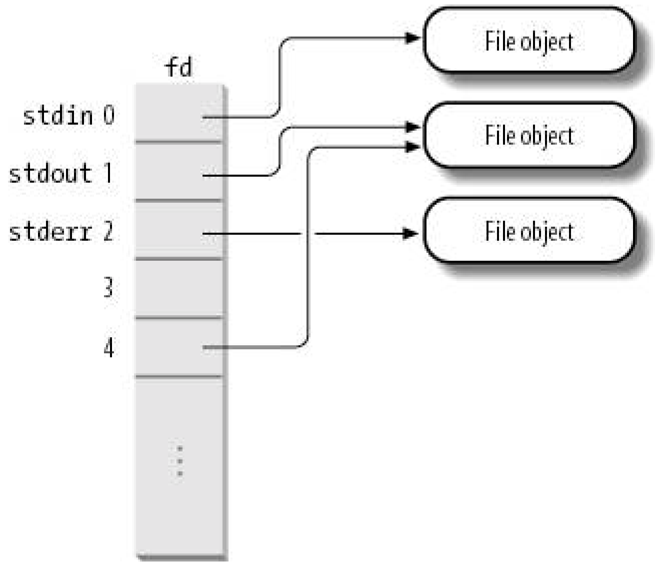
The fd field points to an array of pointers to file objects. The size of the array is stored in the max_fds field. Usually, fd points to the fd_array field of the files_struct structure, which includes 32 file object pointers. If the process opens more than 32 files, the kernel allocates a new, larger array of file pointers and stores its address in the fd fields; it also updates the max_fds field.

For every file with an entry in the fd array, the array index is the *file descriptor*. Usually, the first element (index 0) of the array is associated with the standard input of the process, the second with the standard output, and the third with the standard error (see Figure 12-3). Unix processes use the file descriptor as the main file identifier. Notice that, thanks to the dup( ) , dup2( ) , and fcntl( ) system calls, two file descriptors may refer to the same opened filethat is, two elements of the array could point to the same file object. Users see this all the time when they use shell constructs such as 2>&1 to redirect the standard error to the standard output.

A process cannot use more than NR_OPEN (usually, 1, 048, 576) file descriptors. The kernel also enforces a dynamic bound on the maximum number of file descriptors in the signal->rlim[RLIMIT_NOFILE] structure of the process descriptor; this value is usually 1,024, but it can be raised if the process has root privileges.

The open_fds field initially contains the address of the open_fds_init field, which is a bitmap that identifies the file descriptors of currently opened files. The max_fdset field stores the number of bits in the bitmap. Because the fd_set data structure includes 1,024 bits, there is usually no need to expand the size of the bitmap. However, the kernel may dynamically expand the size of the bitmap if this turns out to be necessary, much as in the case of the array of file objects.

Figure 12-3. The fd array



The kernel provides an fget( ) function to be invoked when the kernel starts using a file object. This function receives as its parameter a file descriptor fd. It returns the address in current->files->fd[fd] (that is, the address of the corresponding file object), or NULL if no file corresponds to fd. In the first case, fget( ) increases the file object usage counter f_count by 1.

The kernel also provides an fput( ) function to be invoked when a kernel control path finishes using a file object. This function receives as its parameter the address of a file object and decreases its usage counter, f_count. Moreover, if this field becomes 0, the function invokes the release method of the file operations (if defined), decreases the `i_writecount` field in the inode object (if the file was opened for writing), removes the file object from the superblock's list, releases the file object to the slab allocator, and decreases the usage counters of the associated dentry object and of the filesystem descriptor (see the later section "Filesystem Mounting).

The `fget_light( )` and `fput_light( )` functions are faster versions of `fget( )` and `fput( )`: the kernel uses them when it can safely assume that the current process already owns the file objectthat is, the process has already previously increased the file object's reference counter. For instance, they are used by the service routines of the system calls that receive a file descriptor as an argument, because the file object's reference counter has been increased by a previous `open( )` system call.

12.3. Filesystem Types

The Linux kernel supports many different types of filesystems. In the following, we introduce a few special types of filesystems that play an important role in the internal design of the Linux kernel.

Next, we'll discuss filesystem registrationthat is, the basic operation that must be performed, usually during system initialization, before using a filesystem type. Once a filesystem is registered, its specific functions are available to the kernel, so that type of filesystem can be mounted on the system's directory tree.

12.3.1. Special Filesystems

While network and disk-based filesystems enable the user to handle information stored outside the kernel, special filesystems may provide an easy way for system programs and administrators to manipulate the data structures of the kernel and to implement special features of the operating system. Table 12-8 lists the most common special filesystems used in Linux; for each of them, the table reports its suggested mount point and a short description.

Notice that a few filesystems have no fixed mount point (keyword "any" in the table). These filesystems can be freely mounted and used by the users. Moreover, some other special filesystems do not have a mount point at all (keyword "none" in the table). They are not for user interaction, but the kernel can use them to easily reuse some of the VFS layer code; for instance, we'll see in Chapter 19 that, thanks to the *pipefs* special filesystem, pipes can be treated in the same way as FIFO files.

Table 12-8. Most common special filesystems

Name	Mount point	Description
bdev	none	Block devices (see Chapter 13)
binfmt_misc	any	Miscellaneous executable formats (see Chapter 20)
devpts	*/dev/pts*	Pseudoterminal support (Open Group's Unix98 standard)
eventpollfs	none	Used by the efficient event polling mechanism
futexfs	none	Used by the futex (Fast Userspace Locking) mechanism

Table 12-8. Most common special filesystems

Name	Mount point	Description
pipefs	none	Pipes (see Chapter 19)
proc	*/proc*	General access point to kernel data structures
rootfs	none	Provides an empty root directory for the bootstrap phase
shm	none	IPC-shared memory regions (see Chapter 19)
mqueue	any	Used to implement POSIX message queues (see Chapter 19)
sockfs	none	Sockets
sysfs	*/sys*	General access point to system data (see Chapter 13)
tmpfs	any	Temporary files (kept in RAM unless swapped)
usbfs	*/proc/bus/usb*	USB devices

Special filesystems are not bound to physical block devices. However, the kernel assigns to each mounted special filesystem a fictitious block device that has the value 0 as major number and an arbitrary value (different for each special filesystem) as a minor number. The `set_anon_super( )` function is used to initialize superblocks of special filesystems; this function essentially gets an unused minor number `dev` and sets the `s_dev` field of the new superblock with major number 0 and minor number `dev`. Another function called `kill_anon_super( )` removes the superblock of a special filesystem. The `unnamed_dev_idr` variable includes pointers to auxiliary structures that record the minor numbers currently in use. Although some kernel designers dislike the fictitious block device identifiers, they help the kernel to handle special filesystems and regular ones in a uniform way.

We'll see a practical example of how the kernel defines and initializes a special filesystem in the later section "Mounting a Generic Filesystem."

12.3.2. Filesystem Type Registration

Often, the user configures Linux to recognize all the filesystems needed when compiling the kernel for his system. But the code for a filesystem actually may either be included in the kernel image or dynamically loaded as a module (see Appendix B). The VFS must keep track of all filesystem types whose code is currently included in the kernel. It does this by performing *filesystem type registration* .

Each registered filesystem is represented as a `file_system_type` object whose fields are illustrated in Table 12-9.

Table 12-9. The fields of the file_system_type object

Type	Field	Description
const char *	name	Filesystem name
int	fs_flags	Filesystem type flags
struct super_block * (*)( )	get_sb	Method for reading a superblock
void (*)( )	kill_sb	Method for removing a superblock
struct module *	owner	Pointer to the module implementing the filesystem (see Appendix B)
struct file_system_type *	next	Pointer to the next element in the list of filesystem types
struct list_head	fs_supers	Head of a list of superblock objects having the same filesystem type

All filesystem-type objects are inserted into a singly linked list. The file_systems variable points to the first item, while the next field of the structure points to the next item in the list. The file_systems_lock read/write spin lock protects the whole list against concurrent accesses.

The fs_supers field represents the head (first dummy element) of a list of superblock objects corresponding to mounted filesystems of the given type. The backward and forward links of a list element are stored in the s_instances field of the superblock object.

The get_sb field points to the filesystem-type-dependent function that allocates a new superblock object and initializes it (if necessary, by reading a disk). The kill_sb field points to the function that destroys a superblock.

The fs_flags field stores several flags, which are listed in Table 12-10.

Table 12-10. The filesystem type flags

Name	Description
FS_REQUIRES_DEV	Every filesystem of this type must be located on a physical disk device.
FS_BINARY_MOUNTDATA	The filesystem uses binary mount data.
FS_REVAL_DOT	Always revalidate the "." and ".." paths in the dentry cache (for network filesystems).
FS_ODD_RENAME	"Rename" operations are "move" operations (for network filesystems).

During system initialization, the `register_filesystem( )` function is invoked for every filesystem specified at compile time; the function inserts the corresponding `file_system_type` object into the filesystem-type list.

The `register_filesystem( )` function is also invoked when a module implementing a filesystem is loaded. In this case, the filesystem may also be unregistered (by invoking the `unregister_filesystem( )` function) when the module is unloaded.

The `get_fs_type( )` function, which receives a filesystem name as its parameter, scans the list of registered filesystems looking at the `name` field of their descriptors, and returns a pointer to the corresponding `file_system_type` object, if it is present.

12.4. Filesystem Handling

Like every traditional Unix system, Linux makes use of a *system's root filesystem* : it is the filesystem that is directly mounted by the kernel during the booting phase and that holds the system initialization scripts and the most essential system programs.

Other filesystems can be mountedeither by the initialization scripts or directly by the userson directories of already mounted filesystems. Being a tree of directories, every filesystem has its own *root directory*. The directory on which a filesystem is mounted is called the *mount point*. A mounted filesystem is a *child* of the mounted filesystem to which the mount point directory belongs. For instance, the */proc* virtual filesystem is a child of the system's root filesystem (and the system's root filesystem is the *parent* of */proc*). The root directory of a mounted filesystem hides the content of the mount point directory of the parent filesystem, as well as the whole subtree of the parent filesystem below the mount point.[*]

[*] The root directory of a filesystem can be different from the root directory of a process: as we have seen in the earlier section "Files Associated with a Process," the process's root directory is the directory corresponding to the "/" pathname. By default, the process' root directory coincides with the root directory of the system's root filesystem (or more precisely, with the root directory of the root filesystem in the namespace of the process, described in the following section), but it can be changed by invoking the `chroot( )` system call.

12.4.1. Namespaces

In a traditional Unix system, there is only one tree of mounted filesystems: starting from the system's root filesystem, each process can potentially access every file in a mounted filesystem by specifying the proper pathname. In this respect, Linux 2.6 is more refined: every process might have its own tree of mounted filesystemsthe so-called *namespace* of the process.

Usually most processes share the same namespace, which is the tree of mounted filesystems that is rooted at the system's root filesystem and that is used by the *init* process. However, a process gets a new namespace if it is created by the `clone( )` system call with the `CLONE_NEWNS` flag set (see the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3). The new namespace is then inherited by children processes if the parent creates them without the `CLONE_NEWNS` flag.

When a process mountsor unmountsa filesystem, it only modifies its namespace. Therefore, the change is visible to all processes that share the same namespace, and only to them. A process can even change the root filesystem of its namespace by using the Linux-specific `pivot_root( )` system call.

The namespace of a process is represented by a `namespace` structure pointed to by the `namespace` field of the process descriptor. The fields of the `namespace` structure are shown in Table 12-11.

Table 12-11. The fields of the namespace structure

Type	Field	Description
`atomic_t`	`count`	Usage counter (how many processes share the namespace)
`struct vfsmount *`	`root`	Mounted filesystem descriptor for the root directory of the namespace
`struct list_head`	`list`	Head of list of all mounted filesystem descriptors
`struct rw_semaphore`	`sem`	Read/write semaphore protecting this structure

The `list` field is the head of a doubly linked circular list collecting all mounted filesystems that belong to the namespace. The `root` field specifies the mounted filesystem that represents the root of the tree of mounted filesystems of this namespace. As we will see in the next section, mounted filesystems are represented by `vfsmount` structures.

12.4.2. Filesystem Mounting

In most traditional Unix-like kernels, each filesystem can be mounted only once. Suppose that an Ext2 filesystem stored in the */dev/fd0* floppy disk is mounted on */flp* by issuing the command:

```
    mount -t ext2 /dev/fd0 /flp
```

Until the filesystem is unmounted by issuing a `umount` command, every other mount command acting on */dev/fd0* fails.

However, Linux is different: it is possible to mount the same filesystem several times. Of course, if a filesystem is mounted *n* times, its root directory can be accessed through *n* mount points, one per mount operation. Although the same filesystem can be accessed by using different mount points, it is really unique. Thus, there is only one superblock object for all of them, no matter of how many times it has been mounted.

Mounted filesystems form a hierarchy: the mount point of a filesystem might be a directory of a second filesystem, which in turn is already mounted over a third filesystem, and so on.[*]

[*] Quite surprisingly, the mount point of a filesystem might be a directory of the same filesystem, provided that it was already mounted. For instance:

```
mount -t ext2 /dev/fd0 /flp; touch /flp/foo
mkdir /flp/mnt; mount -t ext2 /dev/fd0 /flp/mnt
```

Now, the empty *foo* file on the floppy filesystem can be accessed both as */flp/foo* and */flp/mnt/foo*.

It is also possible to stack multiple mounts on a single mount point. Each new mount on the same mount point hides the previously mounted filesystem, although processes already using the files and directories under the old mount can continue to do so. When the topmost mounting is removed, then the next lower mount is once more made visible.

As you can imagine, keeping track of mounted filesystems can quickly become a nightmare. For each mount operation, the kernel must save in memory the mount point and the mount flags, as well as the relationships between the filesystem to be mounted and the other mounted filesystems. Such information is stored in a *mounted filesystem descriptor* of type `vfsmount`. The fields of this descriptor are shown in Table 12-12.

Table 12-12. The fields of the vfsmount data structure

Type	Field	Description
`struct list_head`	`mnt_hash`	Pointers for the hash table list.
`struct vfsmount *`	`mnt_parent`	Points to the parent filesystem on which this filesystem is mounted.
`struct dentry *`	`mnt_mountpoint`	Points to the `dentry` of the mount point directory where the filesystem is mounted.
`struct dentry *`	`mnt_root`	Points to the `dentry` of the root directory of this filesystem.
`struct super_block *`	`mnt_sb`	Points to the superblock object of this filesystem.
`struct list_head`	`mnt_mounts`	Head of a list including all filesystem descriptors mounted on directories of this filesystem.
`struct list_head`	`mnt_child`	Pointers for the `mnt_mounts` list of mounted filesystem descriptors.
`atomic_t`	`mnt_count`	Usage counter (increased to forbid filesystem unmounting).
`int`	`mnt_flags`	Flags.
int	mnt_expiry_mark	Flag set to true if the filesystem is marked as expired (the filesystem can be automatically unmounted if the flag is set and no one is using it).
`char *`	`mnt_devname`	Device filename.

Table 12-12. The fields of the vfsmount data structure

Type	Field	Description
`struct list_head`	`mnt_list`	Pointers for namespace's list of mounted filesystem descriptors.
`struct list_head`	mnt_fslink	Pointers for the filesystem-specific expire list.
struct namespace *	mnt_namespace	Pointer to the namespace of the process that mounted the filesystem.

The `vfsmount` data structures are kept in several doubly linked circular lists:

- A hash table indexed by the address of the `vfsmount` descriptor of the parent filesystem and the address of the dentry object of the mount point directory. The hash table is stored in the `mount_hashtable` array, whose size depends on the amount of RAM in the system. Each item of the table is the head of a circular doubly linked list storing all descriptors that have the same hash value. The `mnt_hash` field of the descriptor contains the pointers to adjacent elements in this list.
- For each namespace, a circular doubly linked list including all mounted filesystem descriptors belonging to the namespace. The `list` field of the `namespace` structure stores the head of the list, while the `mnt_list` field of the `vfsmount` descriptor contains the pointers to adjacent elements in the list.
- For each mounted filesystem, a circular doubly linked list including all child mounted filesystems. The head of each list is stored in the `mnt_mounts` field of the mounted filesystem descriptor; moreover, the `mnt_child` field of the descriptor stores the pointers to the adjacent elements in the list.

The `vfsmount_lock` spin lock protects the lists of mounted filesystem objects from concurrent accesses.

The `mnt_flags` field of the descriptor stores the value of several flags that specify how some kinds of files in the mounted filesystem are handled. These flags, which can be set through options of the *mount* command, are listed in Table 12-13.

Table 12-13. Mounted filesystem flags

Name	Description
`MNT_NOSUID`	Forbid `setuid` and `setgid` flags in the mounted filesystem
`MNT_NODEV`	Forbid access to device files in the mounted filesystem
`MNT_NOEXEC`	Disallow program execution in the mounted filesystem

Here are some functions that handle the mounted filesystem descriptors:

`alloc_vfsmnt(name)`

Allocates and initializes a mounted filesystem descriptor

`free_vfsmnt(mnt)`

Frees a mounted filesystem descriptor pointed to by `mnt`

`lookup_mnt(mnt, dentry)`

Looks up a descriptor in the hash table and returns its address

12.4.3. Mounting a Generic Filesystem

We'll now describe the actions performed by the kernel in order to mount a filesystem. We'll start by considering a filesystem that is going to be mounted over a directory of an already mounted filesystem (in this discussion we will refer to this new filesystem as "generic").

The `mount( )` system call is used to mount a generic filesystem; its `sys_mount( )` service routine acts on the following parameters:

- The pathname of a device file containing the filesystem, or `NULL` if it is not required (for instance, when the filesystem to be mounted is network-based)
- The pathname of the directory on which the filesystem will be mounted (the mount point)
- The filesystem type, which must be the name of a registered filesystem
- The mount flags (permitted values are listed in Table 12-14)
- A pointer to a filesystem-dependent data structure (which may be `NULL`)

Table 12-14. Flags used by the mount() system call

Macro	Description
`MS_RDONLY`	Files can only be read
`MS_NOSUID`	Forbid `setuid` and `setgid` flags
`MS_NODEV`	Forbid access to device files
`MS_NOEXEC`	Disallow program execution
`MS_SYNCHRONOUS`	Write operations on files and directories are immediate
`MS_REMOUNT`	Remount the filesystem changing the mount flags
`MS_MANDLOCK`	Mandatory locking allowed

Table 12-14. Flags used by the mount() system call

Macro	Description
`MS_DIRSYNC`	Write operations on directories are immediate
`MS_NOATIME`	Do not update file access time
`MS_NODIRATIME`	Do not update directory access time
`MS_BIND`	Create a "bind mount," which allows making a file or directory visible at another point of the system directory tree (option `--bind` of the *mount* command)
`MS_MOVE`	Atomically move a mounted filesystem to another mount point (option `--move` of the *mount* command)
`MS_REC`	Recursively create "bind mounts" for a directory subtree
`MS_VERBOSE`	Generate kernel messages on mount errors

The `sys_mount( )` function copies the value of the parameters into temporary kernel buffers, acquires the big kernel lock , and invokes the `do_mount( )` function. Once `do_mount( )` returns, the service routine releases the big kernel lock and frees the temporary kernel buffers.

The `do_mount( )` function takes care of the actual mount operation by performing the following operations:

1. If some of the `MS_NOSUID`, `MS_NODEV`, or `MS_NOEXEC` mount flags are set, it clears them and sets the corresponding flag (`MNT_NOSUID`, `MNT_NODEV`, `MNT_NOEXEC`) in the mounted filesystem object.
2. Looks up the pathname of the mount point by invoking `path_lookup( )`; this function stores the result of the pathname lookup in the local variable `nd` of type `nameidata` (see the later section "Pathname Lookup").
3. Examines the mount flags to determine what has to be done. In particular:
    a. If the `MS_REMOUNT` flag is specified, the purpose is usually to change the mount flags in the `s_flags` field of the superblock object and the mounted filesystem flags in the `mnt_flags` field of the mounted filesystem object. The `do_remount( )` function performs these changes.
    b. Otherwise, it checks the `MS_BIND` flag. If it is specified, the user is asking to make visible a file or directory on another point of the system directory tree.
    c. Otherwise, it checks the `MS_MOVE` flag. If it is specified, the user is asking to change the mount point of an already mounted filesystem. The `do_move_mount( )` function does this atomically.
    d. Otherwise, it invokes `do_new_mount( )`. This is the most common case. It is triggered when the user asks to mount either a special filesystem or a regular filesystem stored in a disk partition. `do_new_mount( )` invokes the `do_kern_mount( )` function passing to it the filesystem type, the mount flags, and the block device name. This function, which takes care of the actual mount operation and returns the address of a new

mounted filesystem descriptor, is described below. Next, `do_new_mount( )` invokes `do_add_mount( )`, which essentially performs the following actions:

    1. Acquires for writing the `namespace->sem` semaphore of the current process, because the function is going to modify the namespace.
    2. The `do_kern_mount( )` function might put the current process to sleep; meanwhile, another process might mount a filesystem on the very same mount point as ours or even change our root filesystem (`current->namespace->root`). Verifies that the lastly mounted filesystem on this mount point still refers to the `current`'s namespace; if not, releases the read/write semaphore and returns an error code.
    3. If the filesystem to be mounted is already mounted on the mount point specified as parameter of the system call, or if the mount point is a symbolic link, it releases the read/write semaphore and returns an error code.
    4. Initializes the flags in the `mnt_flags` field of the new mounted filesystem object allocated by `do_kern_mount( )`.
    5. Invokes `graft_tree( )` to insert the new mounted filesystem object in the namespace list, in the hash table, and in the children list of the parent-mounted filesystem.
    6. Releases the `namespace->sem` read/write semaphore and returns.
4. Invokes `path_release( )` to terminate the pathname lookup of the mount point (see the later section "Pathname Lookup") and returns 0.

12.4.3.1. The do_kern_mount( ) function

The core of the mount operation is the `do_kern_mount( )` function, which checks the filesystem type flags to determine how the mount operation is to be done. This function receives the following parameters:

`fstype`

The name of the filesystem type to be mounted

`flags`

The mount flags (see Table 12-14)

`name`

The pathname of the block device storing the filesystem (or the filesystem type name for special filesystems) `data`

Pointer to additional data to be passed to the `read_super` method of the filesystem

The function takes care of the actual mount operation by performing essentially the following operations:

1. Invokes `get_fs_type( )` to search in the list of filesystem types and locate the name stored in the `fstype` parameter; `get_fs_type( )` returns in the local variable `type` the address of the corresponding `file_system_type` descriptor.
2. Invokes `alloc_vfsmnt( )` to allocate a new mounted filesystem descriptor and stores its address in the `mnt` local variable.
3. Invokes the `type->get_sb( )` filesystem-dependent function to allocate a new superblock and to initialize it (see below).
4. Initializes the `mnt->mnt_sb` field with the address of the new superblock object.
5. Initializes the `mnt->mnt_root` field with the address of the dentry object corresponding to the root directory of the filesystem, and increases the usage counter of the dentry object.
6. Initializes the `mnt->mnt_parent` field with the value in `mnt` (for generic filesystems, the proper value of `mnt_parent` will be set when the mounted filesystem descriptor is inserted in the proper lists by `graft_tree( )`; see step 3d5 of `do_mount( )`).
7. Initializes the `mnt->mnt_namespace` field with the value in `current->namespace`.
8. Releases the `s_umount` read/write semaphore of the superblock object (it was acquired when the object was allocated in step 3).
9. Returns the address `mnt` of the mounted filesystem object.

12.4.3.2. Allocating a superblock object

The `get_sb` method of the filesystem object is usually implemented by a one-line function. For instance, in the Ext2 filesystem the method is implemented as follows:

```
    struct super_block * ext2_get_sb(struct file_system_type *type,
                            int flags, const char *dev_name, void
*data)
    {
        return get_sb_bdev(type, flags, dev_name, data,
ext2_fill_super);
    }
```

The `get_sb_bdev( )` VFS function allocates and initializes a new superblock suitable for disk-based filesystems ; it receives the address of the `ext2_fill_super( )` function, which reads the disk superblock from the Ext2 disk partition.

To allocate superblocks suitable for special filesystems , the VFS also provides the `get_sb_pseudo( )` function (for special filesystems with no mount point such as *pipefs*), the `get_sb_single( )` function (for special filesystems with single mount point such as *sysfs*), and the `get_sb_nodev( )` function (for special filesystems that can be mounted several times such as *tmpfs* ; see below).

The most important operations performed by `get_sb_bdev( )` are the following:

1. Invokes `open_bdev_excl( )` to open the block device having device file name `dev_name` (see the section "Character Device Drivers" in Chapter 13).
2. Invokes `sget( )` to search the list of superblock objects of the filesystem (`type->fs_supers`, see the earlier section "Filesystem Type Registration"). If a superblock relative to the block device is already present, the function returns its address. Otherwise, it allocates and initializes a new superblock object, inserts it into the filesystem list and in the global list of superblocks, and returns its address.
3. If the superblock is not new (it was not allocated in the previous step, because the filesystem is already mounted), it jumps to step 6.
4. Copies the value of the `flags` parameter into the `s_flags` field of the superblock and sets the `s_id`, `s_old_blocksize`, and `s_blocksize` fields with the proper values for the block device.
5. Invokes the filesystem-dependent function passed as last argument to `get_sb_bdev( )` to access the superblock information on disk and fill the other fields of the new superblock object.
6. Returns the address of the new superblock object.

12.4.4. Mounting the Root Filesystem

Mounting the root filesystem is a crucial part of system initialization. It is a fairly complex procedure, because the Linux kernel allows the root filesystem to be stored in many different places, such as a hard disk partition, a floppy disk, a remote filesystem shared via NFS, or even a *ramdisk* (a fictitious block device kept in RAM).

To keep the description simple, let's assume that the root filesystem is stored in a partition of a hard disk (the most common case, after all). While the system boots, the kernel finds the major number of the disk that contains the root filesystem in the `ROOT_DEV` variable (see Appendix A). The root filesystem can be specified as a device file in the */dev* directory either when compiling the kernel or by passing a suitable *"root"* option to the initial bootstrap loader. Similarly, the mount flags of the root filesystem are stored in the `root_mountflags` variable. The user specifies these flags either by using the *rdev* external program on a compiled kernel image or by passing a suitable *rootflags* option to the initial bootstrap loader (see Appendix A).

Mounting the root filesystem is a two-stage procedure, shown in the following list:

1. The kernel mounts the special *rootfs* filesystem, which simply provides an empty directory that serves as initial mount point.
2. The kernel mounts the real root filesystem over the empty directory.

Why does the kernel bother to mount the *rootfs* filesystem before the real one? Well, the *rootfs* filesystem allows the kernel to easily change the real root filesystem. In fact, in some cases, the kernel mounts and unmounts several root filesystems, one after the other. For instance, the initial bootstrap CD of a distribution might load in RAM a kernel with a minimal set of drivers, which mounts as root a minimal filesystem stored in a ramdisk. Next, the programs in this initial root filesystem probe the hardware of the system (for instance, they determine whether the hard disk is EIDE, SCSI, or whatever), load all needed kernel modules, and remount the root filesystem from a physical block device.

12.4.4.1. Phase 1: Mounting the rootfs filesystem

The first stage is performed by the `init_rootfs( )` and `init_mount_tree( )` functions, which are executed during system initialization.

The `init_rootfs( )` function registers the special filesystem type *rootfs*:

```
struct file_system_type rootfs_fs_type = {
    .name = "rootfs";
    .get_sb = rootfs_get_sb;
    .kill_sb = kill_litter_super;
};
register_filesystem(&rootfs_fs_type);
```

The `init_mount_tree( )` function executes the following operations:

1. Invokes `do_kern_mount( )` passing to it the string "`rootfs`" as filesystem type, and stores the address of the mounted filesystem descriptor returned by this function in the `mnt` local variable. As explained in the previous section, `do_kern_mount( )` ends up invoking the `get_sb` method of the *rootfs* filesystem, that is, the `rootfs_get_sb( )` function:

```
struct superblock *rootfs_get_sb(struct file_system_type *fs_type,
                                 int flags, const char *dev_name, void *data)
{
    return get_sb_nodev(fs_type, flags|MS_NOUSER, data,
                        ramfs_fill_super);
}
```

   The `get_sb_nodev( )` function, in turn, executes the following steps:

   a. Invokes `sget( )` to allocate a new superblock passing as parameter the address of the `set_anon_super( )` function (see the earlier section "Special Filesystems"). As a result, the `s_dev` field of the superblock is set in the appropriate way: major number 0, minor number different from those of other mounted special filesystems.
   b. Copies the value of the `flags` parameter into the `s_flags` field of the superblock.
   c. Invokes `ramfs_fill_super( )` to allocate an inode object and a corresponding dentry object, and to fill the superblock fields. Because *rootfs* is a special filesystem that has no disk superblock, only a couple of superblock operations need to be implemented.

   d. Returns the address of the new superblock.

8. Allocates a `namespace` object for the namespace of process 0, and inserts into it the mounted filesystem descriptor returned by `do_kern_mount( )`:

```
9.      namespace = kmalloc(sizeof(*namespace), GFP_KERNEL);
10.     list_add(&mnt->mnt_list, &namespace->list);
11.     namespace->root = mnt;
12.     mnt->mnt_namespace = init_task.namespace = namespace;
```

13. Sets the `namespace` field of every other process in the system to the address of the namespace object; also initializes the `namespace->count` usage counter. (By default, all processes share the same, initial namespace.)
14. Sets the root directory and the current working directory of process 0 to the root filesystem.

12.4.4.2. Phase 2: Mounting the real root filesystem

The second stage of the mount operation for the root filesystem is performed by the kernel near the end of the system initialization. There are several ways to mount the real root filesystem, according to the options selected when the kernel has been compiled and to the boot options passed by the kernel loader. For the sake of brevity, we consider the case of a disk-based filesystem whose device file name has been passed to the kernel by means of the "*root*" boot parameter. We also assume that no initial special filesystem is used, except the *rootfs* filesystem.

The `prepare_namespace( )` function executes the following operations:

1. Sets the `root_device_name` variable with the device filename obtained from the "*root*" boot parameter. Also, sets the `ROOT_DEV` variable with the major and minor numbers of the same device file.
2. Invokes the `mount_root( )` function, which in turn:
   a. Invokes `sys_mknod( )` (the service routine of the `mknod( )` system call) to create a */dev/root* device file in the *rootfs* initial root filesystem, having the major and minor numbers as in `ROOT_DEV`.
   b. Allocates a buffer and fills it with a list of filesystem type names. This list is either passed to the kernel in the "*rootfstype*" boot parameter or built by scanning the elements in the singly linked list of filesystem types.
   c. Scans the list of filesystem type names built in the previous step. For each name, it invokes *sys_mount( )* to try to mount the given filesystem type on the root device. Because each filesystem-specific method uses a different magic number, all `get_sb( )` invocations will fail except the one that attempts to fill the superblock by using the function of the filesystem really used on the root device. The filesystem is mounted on a directory named */root* of the *rootfs* filesystem.
   d. Invokes `sys_chdir("/root")` to change the current directory of the process.
3. Moves the mount point of the mounted filesystem on the root directory of the *rootfs* filesystem:

4.
```
     sys_mount(".", "/", NULL, MS_MOVE, NULL);
    sys_chroot(".");
```

Notice that the *rootfs* special filesystem is not unmounted: it is only hidden under the disk-based root filesystem.

12.4.5. Unmounting a Filesystem

The `umount( )` system call is used to unmount a filesystem. The corresponding `sys_umount( )` service routine acts on two parameters: a filename (either a mount point directory or a block device filename) and a set of flags. It performs the following actions:

1. Invokes `path_lookup( )` to look up the mount point pathname; this function returns the results of the lookup operation in a local variable `nd` of type `nameidata` (see next section).
2. If the resulting directory is not the mount point of a filesystem, it sets the `retval` return code to `-EINVAL` and jumps to step 6. This check is done by verifying that `nd->mnt->mnt_root` contains the address of the dentry object pointed to by `nd.dentry`.
3. If the filesystem to be unmounted has not been mounted in the namespace, it sets the `retval` return code to `-EINVAL` and jumps to step 6. (Recall that some special filesystems have no mount point.) This check is done by invoking the `check_mnt( )` function on `nd->mnt`.
4. If the user does not have the privileges required to unmount the filesystem, it sets the `retval` return code to `-EPERM` and jumps to step 6.
5. Invokes `do_umount( )` passing as parameters `nd.mnt` (the mounted filesystem object) and `flags` (the set of flags). This function performs essentially the following operations:
    a. Retrieves the address of the `sb` superblock object from the `mnt_sb` field of the mounted filesystem object.
    b. If the user asked to force the unmount operation, it interrupts any ongoing mount operation by invoking the `umount_begin` superblock operation.
    c. If the filesystem to be unmounted is the root filesystem and the user didn't ask to actually detach it, it invokes `do_remount_sb( )` to remount the root filesystem read-only and terminates.
    d. Acquires for writing the `namespace->sem` read/write semaphore of the current process, and gets the `vfsmount_lock` spin lock.
    e. If the mounted filesystem does not include mount points for any child mounted filesystem, or if the user asked to forcibly detach the filesystem, it invokes `umount_tree( )` to unmount the filesystem (together with all children filesystems).
    f. Releases the `vfsmount_lock` spin lock and the `namespace->sem` read/write semaphore of the current process.
6. Decreases the usage counters of the dentry object corresponding to the root directory of the filesystem and of the mounted filesystem descriptor; these counters were increased by `path_lookup( )`.
7. Returns the `retval` value.

12.5. Pathname Lookup

When a process must act on a file, it passes its file pathname to some VFS system call, such as `open( )` , `mkdir( )`, `rename( )` , or `stat( )` . In this section, we illustrate how the VFS performs a *pathname lookup* , that is, how it derives an inode from the corresponding file pathname.

The standard procedure for performing this task consists of analyzing the pathname and breaking it into a sequence of filenames . All filenames except the last must identify directories.

If the first character of the pathname is /, the pathname is absolute, and the search starts from the directory identified by `current->fs->root` (the process root directory). Otherwise, the pathname is relative, and the search starts from the directory identified by `current->fs->pwd` (the process-current directory).

Having in hand the dentry, and thus the inode, of the initial directory, the code examines the entry matching the first name to derive the corresponding inode. Then the directory file that has that inode is read from disk and the entry matching the second name is examined to derive the corresponding inode. This procedure is repeated for each name included in the path.

The dentry cache considerably speeds up the procedure, because it keeps the most recently used dentry objects in memory. As we saw before, each such object associates a filename in a specific directory to its corresponding inode. In many cases, therefore, the analysis of the pathname can avoid reading the intermediate directories from disk.

However, things are not as simple as they look, because the following Unix and VFS filesystem features must be taken into consideration:

- The access rights of each directory must be checked to verify whether the process is allowed to read the directory's content.
- A filename can be a symbolic link that corresponds to an arbitrary pathname; in this case, the analysis must be extended to all components of that pathname.
- Symbolic links may induce circular references; the kernel must take this possibility into account and break endless loops when they occur.
- A filename can be the mount point of a mounted filesystem. This situation must be detected, and the lookup operation must continue into the new filesystem.
- Pathname lookup has to be done inside the namespace of the process that issued the system call. The same pathname used by two processes with different namespaces may specify different files.

Pathname lookup is performed by the `path_lookup( )` function, which receives three parameters:

name

A pointer to the file pathname to be resolved.

flags

The value of flags that represent how the looked-up file is going to be accessed. The allowed values are included later in Table 12-16.

nd

The address of a nameidata data structure, which stores the results of the lookup operation and whose fields are shown in Table 12-15.

When path_lookup( ) returns, the nameidata structure pointed to by nd is filled with data pertaining to the pathname lookup operation.

Table 12-15. The fields of the nameidata data structure

Type	Field	Description
struct dentry *	dentry	Address of the dentry object
struct vfs_mount *	mnt	Address of the mounted filesystem object
struct qstr	last	Last component of the pathname (used when the LOOKUP_PARENT flag is set)
unsigned int	flags	Lookup flags
int	last_type	Type of last component of the pathname (used when the LOOKUP_PARENT flag is set)
unsigned int	depth	Current level of symbolic link nesting (see below); it must be smaller than 6
char[ ] *	saved_names	Array of pathnames associated with nested symbolic links
union	intent	One-member union specifying how the file will be accessed

The dentry and mnt fields point respectively to the dentry object and the mounted filesystem object of the last resolved component in the pathname. These two fields "describe" the file that is identified by the given pathname.

Because the dentry object and the mounted filesystem object returned by the path_lookup( ) function in the nameidata structure represent the result of a lookup operation, both objects should not be freed until the caller of path_lookup( ) finishes using them. Therefore, path_lookup( ) increases the usage counters of both objects. If the caller wants to release these objects, it invokes the path_release( ) function passing as parameter the address of a nameidata structure.

The flags field stores the value of some flags used in the lookup operation; they are listed in Table 12-16. Most of these flags can be set by the caller in the flags parameter of path_lookup( ).

Table 12-16. The flags of the lookup operation

Macro	Description
LOOKUP_FOLLOW	If the last component is a symbolic link, interpret (follow) it
LOOKUP_DIRECTORY	The last component must be a directory
LOOKUP_CONTINUE	There are still filenames to be examined in the pathname
LOOKUP_PARENT	Look up the directory that includes the last component of the pathname
LOOKUP_NOALT	Do not consider the emulated root directory (useless in the 80x86 architecture)
LOOKUP_OPEN	Intent is to open a file
LOOKUP_CREATE	Intent is to create a file (if it doesn't exist)
LOOKUP_ACCESS	Intent is to check user's permission for a file

The path_lookup( ) function executes the following steps:

1. Initializes some fields of the nd parameter as follows:
    a. Sets the last_type field to LAST_ROOT (this is needed if the pathname is a slash or a sequence of slashes; see the later section "Parent Pathname Lookup").
    b. Sets the flags field to the value of the flags parameter
    c. Sets the depth field to 0.
2. Acquires for reading the current->fs->lock read/write semaphore of the current process.
3. If the first character in the pathname is a slash (/), the lookup operation must start from the root directory of current: the function gets the addresses of the corresponding mounted filesystem object (current->fs->rootmnt) and dentry object (current->fs->root), increases their usage counters, and stores the addresses in nd->mnt and nd->dentry, respectively.
4. Otherwise, if the first character in the pathname is not a slash, the lookup operation must start from the current working directory of current: the function gets the addresses of the corresponding mounted filesystem object (current->fs->pwdmnt) and dentry object (current->fs->pwd), increases their

usage counters, and stores the addresses in `nd->mnt` and `nd->dentry`, respectively.

5. Releases the `current->fs->lock` read/write semaphore of the current process.
6. Sets the `total_link_count` field in the descriptor of the current process to 0 (see the later section "Lookup of Symbolic Links").
7. Invokes the `link_path_walk( )` function to take care of the undergoing lookup operation:

    ```
    return link_path_walk(name, nd);
    ```

We are now ready to describe the core of the pathname lookup operation, namely the `link_path_walk( )` function. It receives as its parameters a pointer `name` to the pathname to be resolved and the address `nd` of a `nameidata` data structure.

To make things a bit easier, we first describe what `link_path_walk( )` does when `LOOKUP_PARENT` is not set and the pathname does not contain symbolic links (standard pathname lookup). Next, we discuss the case in which `LOOKUP_PARENT` is set: this type of lookup is required when creating, deleting, or renaming a directory entry, that is, during a parent pathname lookup. Finally, we explain how the function resolves symbolic links.

12.5.1. Standard Pathname Lookup

When the `LOOKUP_PARENT` flag is cleared, `link_path_walk( )` performs the following steps.

1. Initializes the `lookup_flags` local variable with `nd->flags`.
2. Skips all leading slashes (/) before the first component of the pathname.
3. If the remaining pathname is empty, it returns the value 0. In the `nameidata` data structure, the `dentry` and `mnt` fields point to the objects relative to the last resolved component of the original pathname.
4. If the `depth` field of the `nd` descriptor is positive, it sets the `LOOKUP_FOLLOW` flag in the `lookup_flags` local variable (see the section "Lookup of Symbolic Links").
5. Executes a cycle that breaks the pathname passed in the `name` parameter into components (the intermediate slashes are treated as filename separators); for each component found, the function:
    a. Retrieves the address of the inode object of the last resolved component from `nd->dentry->d_inode`. (In the first iteration, the inode refers to the directory from where to start the pathname lookup.)
    b. Checks that the permissions of the last resolved component stored into the inode allow execution (in Unix, a directory can be traversed only if it is executable). If the inode has a custom `permission` method, the function executes it; otherwise, it executes the `exec_permission_lite( )` function, which examines the access mode stored in the `i_mode` inode field and the privileges of the running process. In both cases, if the last resolved component does not allow execution, `link_path_walk( )` breaks out of the cycle and returns an error code.

c. Considers the next component to be resolved. From its name, the function computes a 32-bit hash value to be used when looking in the dentry cache hash table.
d. Skips any trailing slash (/) after the slash that terminates the name of the component to be resolved.
e. If the component to be resolved is the last one in the original pathname, it jumps to step 6.
f. If the name of the component is "." (a single dot), it continues with the next component ("." refers to the current directory, so it has no effect inside a pathname).
g. If the name of the component is ".." (two dots), it tries to climb to the parent directory:
    1. If the last resolved directory is the process's root directory (`nd->dentry` is equal to `current->fs->root` and `nd->mnt` is equal to `current->fs->rootmnt`), then climbing is not allowed: it invokes `follow_mount( )` on the last resolved component (see below) and continues with the next component.
    2. If the last resolved directory is the root directory of the `nd->mnt` filesystem (`nd->dentry` is equal to `nd->mnt->mnt_root`) and the `nd->mnt` filesystem is not mounted on top of another filesystem (`nd->mnt` is equal to `nd->mnt->mnt_parent`), then the `nd->mnt` filesystem is usually[*] the namespace's root filesystem: in this case, climbing is impossible, thus invokes `follow_mount( )` on the last resolved component (see below) and continues with the next component.

        [*] This case can also occur for network filesystems disconnected from the namespace's directory tree.

    3. If the last resolved directory is the root directory of the `nd->mnt` filesystem and the `nd->mnt` filesystem is mounted on top of another filesystem, a filesystem switch is required. So, the function sets `nd->dentry` to `nd->mnt->mnt_mountpoint`, and `nd->mnt` to `nd->mnt->mnt_parent`, then restarts step 5g (recall that several filesystems can be mounted on the same mount point).
    4. If the last resolved directory is not the root directory of a mounted filesystem, then the function must simply climb to the parent directory: it sets `nd->dentry` to `nd->dentry->d_parent`, invokes `follow_mount( )` on the parent directory, and continues with the next component.

The `follow_mount( )` function checks whether `nd->dentry` is a mount point for some filesystem (`nd->dentry->d_mounted` is greater than zero); in this case, it invokes `lookup_mnt( )` to search the root directory of the mounted filesystem in the dentry cache , and updates `nd->dentry` and `nd->mnt` with the object addresses corresponding to the mounted filesystem; then, it repeats the whole operation (there can be several filesystems mounted on the same mount point). Essentially, invoking the `follow_mount( )` function when climbing to the parent directory is required because the process could start the pathname lookup from a directory included in a filesystem hidden by another filesystem mounted over the parent directory.

h. The component name is neither "." nor "..", so the function must look it up in the dentry cache. If the low-level filesystem has a custom `d_hash` dentry method, the function invokes it to modify the hash value already computed in step 5c.
i. Sets the `LOOKUP_CONTINUE` flag in `nd->flags` to denote that there is a next component to be analyzed.
j. Invokes `do_lookup( )` to derive the dentry object associated with a given parent directory (`nd->dentry`) and filename (the pathname component being resolved). The function essentially invokes `__d_lookup( )` first to search the dentry object of the component in the dentry cache. If no such object exists, `do_lookup( )` invokes `real_lookup( )`. This latter function reads the directory from disk by executing the `lookup` method of the inode, creates a new dentry object and inserts it in the dentry cache, then creates a new inode object and inserts it into the inode cache .[*] At the end of this step, the `dentry` and `mnt` fields of the `next` local variable will point, respectively, to the dentry object and the mounted filesystem object of the component name to be resolved in this cycle.

[*] In a few cases, the function might find the required inode already in the inode cache. This happens when the pathname component is the last one and it does not refer to a directory, the corresponding file has several hard links, and finally the file has been recently accessed through a hard link different from the one used in this pathname.

k. Invokes the `follow_mount( )` function to check whether the component just resolved (`next.dentry`) refers to a directory that is a mount point for some filesystem (`next.dentry->d_mounted` is greater than zero). `follow_mount( )` updates `next.dentry` and `next.mnt` so that they point to the dentry object and mounted filesystem object of the upmost filesystem mounted on the directory specified by this pathname component (see step 5g).
l. Checks whether the component just resolved refers to a symbolic link (`next.dentry->d_inode` has a custom `follow_link` method). We'll deal with this case in the later section "Lookup of Symbolic Links."
m. Checks whether the component just resolved refers to a directory (`next.dentry->d_inode` has a custom `lookup` method). If not, returns the error `-ENOTDIR`, because the component is in the middle of the original pathname.
n. Sets `nd->dentry` to `next.dentry` and `nd->mnt` to `next.mnt`, then continues with the next component of the pathname.

6. Now all components of the original pathname are resolved except the last one. Clears the `LOOKUP_CONTINUE` flag in `nd->flags`.
7. If the pathname has a trailing slash, it sets the `LOOKUP_FOLLOW` and `LOOKUP_DIRECTORY flags` in the `lookup_flags` local variable to force the last component to be interpreted by later functions as a directory name.
8. Checks the value of the `LOOKUP_PARENT` flag in the `lookup_flags` variable. In the following, we assume that the flag is set to 0, and we postpone the opposite case to the next section.
9. If the name of the last component is "." (a single dot), terminates the execution and returns the value 0 (no error). In the `nameidata` structure that `nd` points to, the `dentry` and `mnt` fields refer to the objects relative to the next-

to-last component of the pathname (each component "." has no effect inside a pathname).

10. If the name of the last component is ".." (two dots), it tries to climb to the parent directory:
    a. If the last resolved directory is the process's root directory (`nd->dentry` is equal to `current->fs->root` and `nd->mnt` is equal to `current->fs->rootmnt`), it invokes `follow_mount( )` on the next-to-last component and terminates the execution and returns the value 0 (no error). `nd->dentry` and `nd->mnt` refer to the objects relative to the next-to-last component of the pathnamethat is, to the root directory of the process.
    b. If the last resolved directory is the root directory of the `nd->mnt` filesystem (`nd->dentry` is equal to `nd->mnt->mnt_root`) and the `nd->mnt` filesystem is not mounted on top of another filesystem (`nd->mnt` is equal to `nd->mnt->mnt_parent`), then climbing is impossible, thus invokes `follow_mount( )` on the next-to-last component and terminates the execution and returns the value 0 (no error).
    c. If the last resolved directory is the root directory of the `nd->mnt` filesystem and the `nd->mnt` filesystem is mounted on top of another filesystem, it sets `nd->dentry` to `nd->mnt->mnt_mountpoint` and `nd->mnt` to `nd->mnt->mnt_parent`, then restarts step 10.
    d. If the last resolved directory is not the root directory of a mounted filesystem, it sets `nd->dentry` to `nd->dentry->d_parent`, invokes `follow_mount( )` on the parent directory, and terminates the execution and returns the value 0 (no error). `nd->dentry` and `nd->mnt` refer to the objects relative to the component preceding the next-to-last component of the pathname.
11. The name of the last component is neither "." nor "..", so the function must look it up in the dentry cache. If the low-level filesystem has a custom `d_hash` dentry method, the function invokes it to modify the hash value already computed in step 5c.
12. Invokes `do_lookup( )` to derive the dentry object associated with the parent directory and the filename (see step 5j). At the end of this step, the `next` local variable contains the pointers to both the dentry and the mounted filesystem descriptor relative to the last component name.
13. Invokes `follow_mount( )` to check whether the last component is a mount point for some filesystem and, if this is the case, to update the `next` local variable with the addresses of the dentry object and mounted filesystem object relative to the root directory of the upmost mounted filesystem.
14. Checks whether the `LOOKUP_FOLLOW` flag is set in `lookup_flags` and the inode object `next.dentry->d_inode` has a custom `follow_link` method. If this is the case, the component is a symbolic link that must be interpreted, as described in the later section "Lookup of Symbolic Links."
15. The component is not a symbolic link or the symbolic link should not be interpreted. Sets the `nd->mnt` and `nd->dentry` fields with the value stored in `next.mnt` and `next.dentry`, respectively. The final dentry object is the result of the whole lookup operation.
16. Checks whether `nd->dentry->d_inode` is `NULL`. This happens when there is no inode associated with the dentry object, usually because the pathname refers to a nonexistent file. In this case, the function returns the error code `-ENOENT`.
17. There is an inode associated with the last component of the pathname. If the `LOOKUP_DIRECTORY` flag is set in `lookup_flags`, it checks that the inode has a

custom `lookup` methodthat is, it is a directory. If not, the function returns the error code `-ENOTDIR`.

18. Returns the value 0 (no error). `nd->dentry` and `nd->mnt` refer to the last component of the pathname.

12.5.2. Parent Pathname Lookup

In many cases, the real target of a lookup operation is not the last component of the pathname, but the next-to-last one. For example, when a file is created, the last component denotes the filename of the not yet existing file, and the rest of the pathname specifies the directory in which the new link must be inserted. Therefore, the lookup operation should fetch the dentry object of the next-to-last component. For another example, unlinking a file identified by the pathname */foo/bar* consists of removing *bar* from the directory *foo*. Thus, the kernel is really interested in accessing the directory *foo* rather than *bar*.

The `LOOKUP_PARENT` flag is used whenever the lookup operation must resolve the directory containing the last component of the pathname, rather than the last component itself.

When the `LOOKUP_PARENT` flag is set, the `link_path_walk( )` function also sets up the `last` and `last_type` fields of the `nameidata` data structure. The `last` field stores the name of the last component in the pathname. The `last_type` field identifies the type of the last component; it may be set to one of the values shown in Table 12-17.

Table 12-17. The values of the last_type field in the nameidata data structure

Value	Description
`LAST_NORM`	Last component is a regular filename
`LAST_ROOT`	Last component is "/ " (that is, the entire pathname is "/ ")
`LAST_DOT`	Last component is "."
`LAST_DOTDOT`	Last component is ".."
`LAST_BIND`	Last component is a symbolic link into a special filesystem

The `LAST_ROOT` flag is the default value set by `path_lookup( )` when the whole pathname lookup operation starts (see the description at the beginning of the section "Pathname Lookup"). If the pathname turns out to be simply "/ ", the kernel does not change the initial value of the `last_type` field.

The remaining values of the `last_type` field are set by `link_path_walk( )` when the `LOOKUP_PARENT` flag is set; in this case, the function performs the same steps described in the previous section up to step 8. From step 8 onward, however, the lookup operation for the last component of the pathname is different:

1. Sets `nd->last` to the name of the last component.

2. Initializes `nd->last_type` to `LAST_NORM`.
3. If the name of the last component is "." (a single dot), it sets `nd->last_type` to `LAST_DOT`.
4. If the name of the last component is ".." (two dots), it sets `nd->last_type` to `LAST_DOTDOT`.
5. Returns the value 0 (no error).

As you can see, the last component is not interpreted at all. Thus, when the function terminates, the `dentry` and `mnt` fields of the `nameidata` data structure point to the objects relative to the directory that includes the last component.

12.5.3. Lookup of Symbolic Links

Recall that a symbolic link is a regular file that stores a pathname of another file. A pathname may include symbolic links, and they must be resolved by the kernel.

For example, if */foo/bar* is a symbolic link pointing to (containing the pathname) *../dir*, the pathname */foo/bar/file* must be resolved by the kernel as a reference to the file */dir/file*. In this example, the kernel must perform two different lookup operations. The first one resolves */foo/bar*: when the kernel discovers that *bar* is the name of a symbolic link, it must retrieve its content and interpret it as another pathname. The second pathname operation starts from the directory reached by the first operation and continues until the last component of the symbolic link pathname has been resolved. Next, the original lookup operation resumes from the dentry reached in the second one and with the component following the symbolic link in the original pathname.

To further complicate the scenario, the pathname included in a symbolic link may include other symbolic links. You might think that the kernel code that resolves the symbolic links is hard to understand, but this is not true; the code is actually quite simple because it is recursive.

However, untamed recursion is intrinsically dangerous. For instance, suppose that a symbolic link points to itself. Of course, resolving a pathname including such a symbolic link may induce an endless stream of recursive invocations, which in turn quickly leads to a kernel stack overflow. The `link_count` field in the descriptor of the current process is used to avoid the problem: the field is increased before each recursive execution and decreased right after. If a sixth nested lookup operation is attempted, the whole lookup operation terminates with an error code. Therefore, the level of nesting of symbolic links can be at most 5.

Furthermore, the `total_link_count` field in the descriptor of the current process keeps track of how many symbolic links (even nonnested) were followed in the original lookup operation. If this counter reaches the value 40, the lookup operation aborts. Without this counter, a malicious user could create a pathological pathname including many consecutive symbolic links that freeze the kernel in a very long lookup operation.

This is how the code basically works: once the `link_path_walk( )` function retrieves the dentry object associated with a component of the pathname, it checks whether the corresponding inode object has a custom `follow_link` method (see step 5l and step 14 in the section "Standard Pathname Lookup"). If so, the inode is a symbolic link that must be interpreted before proceeding with the lookup operation of the original pathname.

In this case, the `link_path_walk( )` function invokes `do_follow_link( )`, passing to it the address `dentry` of the dentry object of the symbolic link and the address `nd` of the `nameidata` data structure. In turn, `do_follow_link( )` performs the following steps:

1. Checks that `current->link_count` is less than 5; otherwise, it returns the error code `-ELOOP`.
2. Checks that `current->total_link_count` is less than 40; otherwise, it returns the error code `-ELOOP`.
3. Invokes `cond_resched( )` to perform a process switch if required by the current process (flag `TIF_NEED_RESCHED` in the `tHRead_info` descriptor of the current process set).
4. Increases `current->link_count`, `current->total_link_count`, and `nd->depth`.
5. Updates the access time of the inode object associated with the symbolic link to be resolved.
6. Invokes the filesystem-dependent function that implements the `follow_link` method passing to it the `dentry` and `nd` parameters. This function extracts the pathname stored in the symbolic link's inode, and saves this pathname in the proper entry of the `nd->saved_names` array.
7. Invokes the `_ _vfs_follow_link( )` function passing to it the address `nd` and the address of the pathname in the `nd->saved_names` array (see below).
8. If defined, executes the `put_link` method of the inode object, thus releasing the temporary data structures allocated by the `follow_link` method.
9. Decreases the `current->link_count` and `nd->depth` fields.
10. Returns the error code returned by the `_ _vfs_follow_link( )` function (0 for no error).

In turn, the `_ _vfs_follow_link( )` does essentially the following:

1. Checks whether the first character of the pathname stored in the symbolic link is a slash: in this case an absolute pathname has been found, so there is no need to keep in memory any information about the previous path. If so, invokes `path_release( )` on the `nameidata` structure, thus releasing the objects resulting from the previous lookup steps; then, the function sets the `dentry` and `mnt` fields of the `nameidata` data structure to the current process root directory.
2. Invokes `link_path_walk( )` to resolve the symbolic link pathname, passing to it as parameters the pathname and `nd`.
3. Returns the value taken from `link_path_walk( )`.

When `do_follow_link( )` finally terminates, it has set the `dentry` field of the `next` local variable with the address of the dentry object referred to by the symbolic link to the original execution of `link_path_walk( )`. The `link_path_walk( )` function can then proceed with the next step.

12.6. Implementations of VFS System Calls

For the sake of brevity, we cannot discuss the implementation of all the VFS system calls listed in Table 12-1. However, it could be useful to sketch out the implementation of a few system calls, in order to show how VFS's data structures interact.

Let's reconsider the example proposed at the beginning of this chapter: a user issues a shell command that copies the MS-DOS file */floppy/TEST* to the Ext2 file */tmp/test*. The command shell invokes an external program such as *cp*, which we assume executes the following code fragment:

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test", O_WRONLY | O_CREAT | O_TRUNC, 0600);
do {
    len = read(inf, buf, 4096);
    write(outf, buf, len);
} while (len);
close(outf);
close(inf);
```

Actually, the code of the real *cp* program is more complicated, because it must also check for possible error codes returned by each system call. In our example, we focus our attention on the "normal" behavior of a copy operation.

12.6.1. The open( ) System Call

The `open( )` system call is serviced by the `sys_open( )` function, which receives as its parameters the pathname `filename` of the file to be opened, some access mode flags `flags`, and a permission bit mask `mode` if the file must be created. If the system call succeeds, it returns a file descriptorthat is, the index assigned to the new file in the `current->files->fd` array of pointers to file objects; otherwise, it returns -1.

In our example, `open( )` is invoked twice; the first time to open */floppy/TEST* for reading (`O_RDONLY` flag) and the second time to open */tmp/test* for writing (`O_WRONLY` flag). If */tmp/test* does not already exist, it is created (`O_CREAT` flag) with exclusive read and write access for the owner (octal `0600` number in the third parameter).

Conversely, if the file already exists, it is rewritten from scratch (`O_TRUNC` flag). Table 12-18 lists all flags of the `open( )` system call.

Table 12-18. The flags of the open( ) system call

Flag name	Description
`O_RDONLY`	Open for reading
`O_WRONLY`	Open for writing

Table 12-18. The flags of the open( ) system call

Flag name	Description
O_RDWR	Open for both reading and writing
O_CREAT	Create the file if it does not exist
O_EXCL	With O_CREAT, fail if the file already exists
O_NOCTTY	Never consider the file as a controlling terminal
O_TRUNC	Truncate the file (remove all existing contents)
O_APPEND	Always write at end of the file
O_NONBLOCK	No system calls will block on the file
O_NDELAY	Same as O_NONBLOCK
O_SYNC	Synchronous write (block until physical write terminates)
FASYNC	I/O event notification via signals
O_DIRECT	Direct I/O transfer (no kernel buffering)
O_LARGEFILE	Large file (size greater than 2 GB)
O_DIRECTORY	Fail if file is not a directory
O_NOFOLLOW	Do not follow a trailing symbolic link in pathname
O_NOATIME	Do not update the inode's last access time

Let's describe the operation of the sys_open( ) function. It performs the following steps:

1. Invokes getname( ) to read the file pathname from the process address space.
2. Invokes get_unused_fd( ) to find an empty slot in current->files->fd. The corresponding index (the new file descriptor) is stored in the fd local variable.
3. Invokes the filp_open( ) function, passing as parameters the pathname, the access mode flags, and the permission bit mask. This function, in turn, executes the following steps:
    a. Copies the access mode flags into namei_flags, but encodes the access mode flags O_RDONLY, O_WRONLY, and O_RDWR with a special format: the bit at index 0 (lowest-order) of namei_flags is set only if the file access requires read privileges; similarly, the bit at index 1 is set only if the file access requires write privileges. Notice that it is not possible to specify in the open( ) system call that a file access does not require either read or write privileges; this makes sense, however, in a pathname lookup operation involving symbolic links.
    b. Invokes open_namei( ), passing to it the pathname, the modified access mode flags, and the address of a local nameidata data structure. The function performs the lookup operation in the following manner:
        - If O_CREAT is not set in the access mode flags, starts the lookup operation with the LOOKUP_PARENT flag not set and the LOOKUP_OPEN flag set. Moreover, the LOOKUP_FOLLOW flag is set

only if `O_NOFOLLOW` is cleared, while the `LOOKUP_DIRECTORY` flag is set only if the `O_DIRECTORY` flag is set.

- If `O_CREAT` is set in the access mode flags, starts the lookup operation with the `LOOKUP_PARENT`, `LOOKUP_OPEN`, and `LOOKUP_CREATE` flags set. Once the `path_lookup( )` function successfully returns, checks whether the requested file already exists. If not, allocates a new disk inode by invoking the `create` method of the parent inode.

The `open_namei( )` function also executes several security checks on the file located by the lookup operation. For instance, the function checks whether the inode associated with the dentry object found really exists, whether it is a regular file, and whether the current process is allowed to access it according to the access mode flags. Also, if the file is opened for writing, the function checks that the file is not locked by other processes.

c. Invokes the `dentry_open( )` function, passing to it the addresses of the dentry object and the mounted filesystem object located by the lookup operation, and the access mode flags. In turn, this function:

1. Allocates a new file object.
2. Initializes the `f_flags` and `f_mode` fields of the file object according to the access mode flags passed to the `open( )` system call.
3. Initializes the `f_dentry` and `f_vfsmnt` fields of the file object according to the addresses of the dentry object and the mounted filesystem object passed as parameters.
4. Sets the `f_op` field to the contents of the `i_fop` field of the corresponding inode object. This sets up all the methods for future file operations.
5. Inserts the file object into the list of opened files pointed to by the `s_files` field of the filesystem's superblock.
6. If the `open` method of the file operations is defined, the function invokes it.
7. Invokes `file_ra_state_init( )` to initialize the read-ahead data structures (see Chapter 16).
8. If the `O_DIRECT` flag is set, it checks whether direct I/O operations can be performed on the file (see Chapter 16).
9. Returns the address of the file object.

d. Returns the address of the file object.

4. Sets `current->files->fd[fd]` to the address of the file object returned by `dentry_open( )`.
5. Returns `fd`.

12.6.2. The read( ) and write( ) System Calls

Let's return to the code in our *cp* example. The `open( )` system calls return two file descriptors, which are stored in the `inf` and `outf` variables. Then the program starts a loop: at each iteration, a portion of the */floppy/TEST* file is copied into a local buffer (`read( )` system call), and then the data in the local buffer is written into the */tmp/test* file (`write( )` system call).

The `read( )` and `write( )` system calls are quite similar. Both require three parameters: a file descriptor `fd`, the address `buf` of a memory area (the buffer containing the data to be transferred), and a number `count` that specifies how many bytes should be transferred. Of course, `read( )` transfers the data from the file into the buffer, while `write( )` does the opposite. Both system calls return either the number of bytes that were successfully transferred or -1 to signal an error condition.

A return value less than `count` does not mean that an error occurred. The kernel is always allowed to terminate the system call even if not all requested bytes were transferred, and the user application must accordingly check the return value and reissue, if necessary, the system call. Typically, a small value is returned when reading from a pipe or a terminal device, when reading past the end of the file, or when the system call is interrupted by a signal. The end-of-file condition (EOF) can easily be recognized by a zero return value from `read( )`. This condition will not be confused with an abnormal termination due to a signal, because if `read( )` is interrupted by a signal before a data is read, an error occurs.

The read or write operation always takes place at the file offset specified by the current file pointer (field `f_pos` of the file object). Both system calls update the file pointer by adding the number of transferred bytes to it.

In short, both `sys_read( )` (the `read( )`'s service routine) and `sys_write( )` (the `write( )`'s service routine) perform almost the same steps:

1. Invokes `fget_light( )` to derive from `fd` the address `file` of the corresponding file object (see the earlier section "Files Associated with a Process").
2. If the flags in `file->f_mode` do not allow the requested access (read or write operation), it returns the error code `-EBADF`.
3. If the `file` object does not have a `read( )` or `aio_read( )` (`write( )` or `aio_write( )`) file operation, it returns the error code `-EINVAL`.
4. Invokes `access_ok()` to perform a coarse check on the `buf` and `count` parameters (see the section "Verifying the Parameters" in Chapter 10).
5. Invokes `rw_verify_area( )` to check whether there are conflicting mandatory locks for the file portion to be accessed. If so, it returns an error code, or puts the current process to sleep if the lock has been requested with a `F_SETLKW` command (see the section "File Locking" later in this chapter).
6. If defined, it invokes either the `file->f_op->read` or `file->f_op->write` method to transfer the data; otherwise, invokes either the `file->f_op->aio_read` or `file->f_op->aio_write` method. All these methods, which are discussed in Chapter 16, return the number of bytes that were actually transferred. As a side effect, the file pointer is properly updated.
7. Invokes `fput_light( )` to release the file object.
8. Returns the number of bytes actually transferred.

12.6.3. The close( ) System Call

The loop in our example code terminates when the `read( )` system call returns the value 0that is, when all bytes of */floppy/TEST* have been copied into */tmp/test*. The program can then close the open files, because the copy operation has completed.

The `close( )` system call receives as its parameter `fd`, which is the file descriptor of the file to be closed. The `sys_close( )` service routine performs the following operations:

1. Gets the file object address stored in `current->files->fd[fd]`; if it is `NULL`, returns an error code.
2. Sets `current->files->fd[fd]` to `NULL`. Releases the file descriptor `fd` by clearing the corresponding bits in the `open_fds` and `close_on_exec` fields of `current->files` (see Chapter 20 for the Close on Execution flag).
3. Invokes `filp_close( )`, which performs the following operations:
    a. Invokes the `flush` method of the file operations, if defined.
    b. Releases all mandatory locks on the file, if any (see next section).
    c. Invokes `fput( )` to release the file object.
4. Returns 0 or an error code. An error code can be raised by the `flush` method or by an error in a previous write operation on the file.

12.7. File Locking

When a file can be accessed by more than one process, a synchronization problem occurs. What happens if two processes try to write in the same file location? Or again, what happens if a process reads from a file location while another process is writing into it?

In traditional Unix systems, concurrent accesses to the same file location produce unpredictable results. However, Unix systems provide a mechanism that allows the processes to *lock* a file region so that concurrent accesses may be easily avoided.

The POSIX standard requires a file-locking mechanism based on the `fcntl( )` system call. It is possible to lock an arbitrary region of a file (even a single byte) or to lock the whole file (including data appended in the future). Because a process can choose to lock only a part of a file, it can also hold multiple locks on different parts of the file.

This kind of lock does not keep out another process that is ignorant of locking. Like a semaphore used to protect a critical region in code, the lock is considered "advisory" because it doesn't work unless other processes cooperate in checking the existence of a lock before accessing the file. Therefore, POSIX's locks are known as *advisory locks* .

Traditional BSD variants implement advisory locking through the `flock( )` system call. This call does not allow a process to lock a file region, only the whole file. Traditional System V variants provide the `lockf( )` library function, which is simply an interface to `fcntl( )`.

More importantly, System V Release 3 introduced *mandatory locking*: the kernel checks that every invocation of the `open( )` , `read( )` , and `write( )` system calls does not violate a mandatory lock on the file being accessed. Therefore, mandatory locks are enforced even between noncooperative processes.[*]

[*] Oddly enough, a process may still unlink (delete) a file even if some other process owns a mandatory lock on it! This perplexing situation is possible because when a process deletes a file hard link, it does not modify its contents, but only the contents of its parent directory.

Whether processes use advisory or mandatory locks, they can use both shared *read locks* and exclusive *write locks* . Several processes may have read locks on some file region, but only one process can have a write lock on it at the same time. Moreover, it is not possible to get a write lock when another process owns a read lock for the same file region, and vice versa.

12.7.1. Linux File Locking

Linux supports all types of file locking: advisory and mandatory locks, plus the `fcntl( )` and `flock( )` system calls (`lockf( )` is implemented as a standard library function).

The expected behavior of the `flock( )` system call in every Unix-like operating system is to produce advisory locks only, without regard for the `MS_MANDLOCK` mount flag. In Linux, however, a special kind of `flock( )`'s mandatory lock is used to support some proprietary network filesystems . It is the so-called *share-mode mandatory lock*; when set, no other process may open a file that would conflict with the access mode of the lock. Use of this feature for native Unix applications is discouraged, because the resulting source code will be nonportable.

Another kind of `fcntl( )`-based mandatory lock called *lease* has been introduced in Linux. When a process tries to open a file protected by a lease, it is blocked as usual. However, the process that owns the lock receives a signal. Once informed, it should first update the file so that its content is consistent, and then release the lock. If the owner does not do this in a well-defined time interval (tunable by writing a number of seconds into */proc /sys/fs/lease-break-time*, usually 45 seconds), the lease is automatically removed by the kernel and the blocked process is allowed to continue.

A process can get or release an advisory file lock on a file in two possible ways:

- By issuing the `flock( )` system call. The two parameters of the system call are the `fd` file descriptor, and a command to specify the lock operation. The lock applies to the whole file.
- By using the `fcntl( )` system call. The three parameters of the system call are the `fd` file descriptor, a command to specify the lock operation, and a pointer to a `flock` structure (see Table 12-20). A couple of fields in this structure allow the process to specify the portion of the file to be locked. Processes can thus hold several locks on different portions of the same file.

Both the `fcntl( )` and the `flock( )` system call may be used on the same file at the same time, but a file locked through `fcntl( )` does not appear locked to `flock( )`, and vice versa. This has been done on purpose in order to avoid the deadlocks occurring when an application using a type of lock relies on a library that uses the other type.

Handling mandatory file locks is a bit more complex. Here are the steps to follow:

1. Mount the filesystem where mandatory locking is required using the `-o mand` option in the *mount* command, which sets the `MS_MANDLOCK` flag in the `mount( )` system call. The default is to disable mandatory locking.

2. Mark the files as candidates for mandatory locking by setting their set-group bit (SGID) and clearing the group-execute permission bit. Because the set-group bit makes no sense when the group-execute bit is off, the kernel interprets that combination as a hint to use mandatory locks instead of advisory ones.
3. Uses the `fcntl( )` system call (see below) to get or release a file lock.

Handling leases is much simpler than handling mandatory locks: it is sufficient to invoke a `fcntl( )` system call with a `F_SETLEASE` or `F_GETLEASE` command. Another `fcntl( )` invocation with the `F_SETSIG` command may be used to change the type of signal to be sent to the lease process holder.

Besides the checks in the `read( )` and `write( )` system calls, the kernel takes into consideration the existence of mandatory locks when servicing all system calls that could modify the contents of a file. For instance, an `open( )` system call with the `O_TRUNC` flag set fails if any mandatory lock exists for the file.

The following section describes the main data structure used by the kernel to handle file locks issued by means of the `flock( )` system call (`FL_FLOCK` locks) and of the `fcntl( )` system call (`FL_POSIX` locks).

12.7.2. File-Locking Data Structures

All type of Linux locks are represented by the same `file_lock` data structure whose fields are shown in Table 12-19.

Table 12-19. The fields of the file_lock data structure

Type	Field	Description
`struct file_lock *`	`fl_next`	Next element in list of locks associated with the inode
`struct list_head`	`fl_link`	Pointers for active or blocked list
`struct list_head`	`fl_block`	Pointers for the lock's waiters list
`struct files_struct *`	`fl_owner`	Owner's `files_struct`
`unsigned int`	`fl_pid`	PID of the process owner
`wait_queue_head_t`	`fl_wait`	Wait queue of blocked processes
`struct file *`	`fl_file`	Pointer to file object
`unsigned char`	`fl_flags`	Lock flags
`unsigned char`	`fl_type`	Lock type
`loff_t`	`fl_start`	Starting offset of locked region
`loff_t`	`fl_end`	Ending offset of locked region
`struct fasync_struct *`	`fl_fasync`	Used for lease break notifications
unsigned long	fl_break_time	Remaining time before end of lease

Table 12-19. The fields of the file_lock data structure

Type	Field	Description
struct file_lock_operations *	fl_ops	Pointer to file lock operations
struct lock_manager_operations *	fl_mops	Pointer to lock manager operations
`union`	`fl_u`	Filesystem-specific information

All `lock_file` structures that refer to the same file on disk are collected in a singly linked list, whose first element is pointed to by the `i_flock` field of the inode object. The `fl_next` field of the `lock_file` structure specifies the next element in the list.

When a process issues a blocking system call to require an exclusive lock while there are shared locks on the same file, the lock request cannot be satisfied immediately and the process must be suspended. The process is thus inserted into a wait queue pointed to by the `fl_wait` field of the blocked lock's `file_lock` structure. Two lists are used to distinguish lock requests that have been satisfied (*active locks*) from those that cannot be satisfied right away (*blocked locks*).

All active locks are linked together in the "global file lock list" whose head element is stored in the `file_lock_list` variable. Similarly, all blocked locks are linked together in the "blocked list" whose head element is stored in the `blocked_list` variable. The `fl_link` field is used to insert a `lock_file` structure in either one of these two lists.

Last but not least, the kernel must keep track of all blocked locks (the "waiters") associated with a given active lock (the "blocker"): this is the purpose of a list that links together all waiters with respect to a given blocker. The `fl_block` field of the blocker is the dummy head of the list, while the `fl_block` fields of the waiters store the pointers to the adjacent elements in the list.

12.7.3. FL_FLOCK Locks

An `FL_FLOCK` lock is always associated with a file object and is thus owned by the process that opened the file (or by all clone processes sharing the same opened file). When a lock is requested and granted, the kernel replaces every other lock that the process is holding on the same file object with the new lock. This happens only when a process wants to change an already owned read lock into a write one, or vice versa. Moreover, when a file object is being freed by the `fput( )` function, all `FL_FLOCK` locks that refer to the file object are destroyed. However, there could be other `FL_FLOCK` read locks set by other processes for the same file (inode), and they still remain active.

The `flock( )` system call allows a process to apply or remove an advisory lock on an open file. It acts on two parameters: the `fd` file descriptor of the file to be acted upon and a `cmd` parameter that specifies the lock operation. A `cmd` parameter of `LOCK_SH` requires a shared lock for reading, `LOCK_EX` requires an exclusive lock for writing, and `LOCK_UN` releases the lock.[*]

[*] Actually, the `flock( )` system call can also establish share-mode mandatory locks by specifying the command `LOCK_MAND`. However, we'll not further discuss this case.

Usually this system call blocks the current process if the request cannot be immediately satisfied, for instance if the process requires an exclusive lock while some other process has already acquired the same lock. However, if the `LOCK_NB` flag is passed together with the `LOCK_SH` or `LOCK_EX` operation, the system call does not block; in other words, if the lock cannot be immediately obtained, the system call returns an error code.

When the `sys_flock( )` service routine is invoked, it performs the following steps:

1. Checks whether `fd` is a valid file descriptor; if not, returns an error code. Gets the address `filp` of the corresponding file object.
2. Checks that the process has read and/or write permission on the open file; if not, returns an error code.
3. Gets a new `file_lock` object `lock` and initializes it in the appropriate way: the `fl_type` field is set according to the value of the parameter `cmd`, the `fl_file` field is set to the address `filp` of the file object, the `fl_flags` field is set to `FL_FLOCK`, the `fl_pid` field is set to `current->tgid`, and the `fl_end` field is set to `-1` to denote the fact that locking refers to the whole file (and not to a portion of it).
4. If the `cmd` parameter does not include the `LOCK_NB` bit, it adds to the `fl_flags` field the `FL_SLEEP` flag.
5. If the file has a `flock` file operation, the routine invokes it, passing as its parameters the file object pointer `filp`, a flag (`F_SETLKW` or `F_SETLK` depending on the value of the `LOCK_NB` bit), and the address of the new `file_lock` object `lock`.
6. Otherwise, if the `flock` file operation is not defined (the common case), invokes `flock_lock_file_wait( )` to try to perform the required lock operation. Two parameters are passed: `filp`, a file object pointer, and `lock`, the address of the new `file_lock` object created in step 3.
7. If the `file_lock` descriptor has not been inserted in the active or blocked lists in the previous step, the routine releases it.
8. Returns 0 in case of success.

The `flock_lock_file_wait( )` function executes a cycle consisting of the following steps:

1. Invokes `flock_lock_file( )` passing as parameters the file object pointer `filp` and the address of the new `file_lock` object `lock`. This function performs, in turn, the following operations:
    a. Searches the list that `filp->f_dentry->d_inode->i_flock` points to. If an `FL_FLOCK` lock for the same file object is found, checks its type (`LOCK_SH` or `LOCK_EX`): if it is equal to the type of the new lock, returns 0 (nothing has to be done). Otherwise, the function removes the old element from the list of locks on the inode and the global file lock list, wakes up all processes sleeping in the wait queues of the locks in the `fl_block` list, and frees the `file_lock` structure.

    b. If the process is performing an unlock (LOCK_UN), nothing else needs to be done: the lock was nonexisting or it has already been released, thus returns 0.
    c. If an FL_FLOCK lock for the same file object has been foundthus the process is changing an already owned read lock into a write one (or vice versa)gives some other higher-priority process, in particular every process previously blocked on the old file lock, a chance to run by invoking cond_resched( ).
    d. Searches the list of locks on the inode again to verify that no existing FL_FLOCK lock conflicts with the requested one. There must be no FL_FLOCK write lock in the list, and moreover, there must be no FL_FLOCK lock at all if the process is requesting a write lock.
    e. If no conflicting lock exists, it inserts the new file_lock structure into the inode's lock list and into the global file lock list, then returns 0 (success).
    f. A conflicting lock has been found: if the FL_SLEEP flag in the fl_flags field is set, it inserts the new lock (the waiter lock) in the circular list of the blocker lock and in the global blocked list.
    g. Returns the error code -EAGAIN.
2. Checks the return code of flock_lock_file( ):
    a. If the return code is 0 (no conflicting looks), it returns 0 (success).
    b. There are incompatibilities. If the FL_SLEEP flag in the fl_flags field is cleared, it releases the lock file_lock descriptor and returns -EAGAIN.
    c. Otherwise, there are incompatibilities but the process can sleep: invokes wait_event_interruptible( ) to insert the current process in the lock->fl_wait wait queue and to suspend it. When the process is awakened (right after the blocker lock has been released), it jumps to step 1 to retry the operation.

12.7.4. FL_POSIX Locks

An FL_POSIX lock is always associated with a process *and* with an inode; the lock is automatically released either when the process dies or when a file descriptor is closed (even if the process opened the same file twice or duplicated a file descriptor). Moreover, FL_POSIX locks are never inherited by a child across a fork( ).

When used to lock files, the fcntl( ) system call acts on three parameters: the fd file descriptor of the file to be acted upon, a cmd parameter that specifies the lock operation, and an fl pointer to a flock data structure[*] stored in the User Mode process address space; its fields are described in Table 12-20.

[*] Linux also defines a flock64 structure, which uses 64-bit long integers for the offset and length fields. In the following, we focus on the flock data structure, but the description is valid for flock64 too.

Table 12-20. The fields of the flock data structure

Type	Field	Description
short	l_type	F_RDLOCK (requests a shared lock), F_WRLOCK (requests an exclusive lock), F_UNLOCK (releases the lock)

Table 12-20. The fields of the flock data structure

Type	Field	Description
`short`	`l_whence`	`SEEK_SET` (from beginning of file), `SEEK_CURRENT` (from current file pointer), `SEEK_END` (from end of file)
`off_t`	`l_start`	Initial offset of the locked region relative to the value of `l_whence`
`off_t`	`l_len`	Length of locked region (0 means that the region includes all potential writes past the current end of the file)
`pid_t`	`l_pid`	PID of the owner

The `sys_fcntl( )` service routine behaves differently, depending on the value of the flag set in the `cmd` parameter:

`F_GETLK`

Determines whether the lock described by the `flock` structure conflicts with some `FL_POSIX` lock already obtained by another process. In this case, the `flock` structure is overwritten with the information about the existing lock.

`F_SETLK`

Sets the lock described by the `flock` structure. If the lock cannot be acquired, the system call returns an error code.

`F_SETLKW`

Sets the lock described by the `flock` structure. If the lock cannot be acquired, the system call blocks; that is, the calling process is put to sleep until the lock is available.

`F_GETLK64`, `F_SETLK64`, `F_SETLKW64`

Identical to the previous ones, but the `flock64` data structure is used rather than `flock`.

The `sys_fcntl( )` service routine gets first a file object corresponding to the `fd` parameter and invokes then `fcntl_getlk( )` or `fcntl_setlk( )`, depending on the command passed as its parameter (`F_GETBLK` for the former function, `F_SETLK` or `F_SETLKW` for the latter one). We'll consider the second case only.

The `fcntl_setlk( )` function acts on three parameters: a `filp` pointer to the file object, a `cmd` command (`F_SETLK` or `F_SETLKW`), and a pointer to a `flock` data structure. The steps performed are the following:

1. Reads the structure pointed to by the `fl` parameter in a local variable of type `flock`.
2. Checks whether the lock should be a mandatory one and the file has a shared memory mapping (see the section "Memory Mapping" in Chapter 16). In this case, the function refuses to create the lock and returns the `-EAGAIN` error code, because the file is already being accessed by another process.
3. Initializes a new `file_lock` structure according to the contents of the user's `flock` structure and to the file size stored in the file's inode.
4. If the command is `F_SETLKW`, the function sets the `FL_SLEEP` flag in the `fl_flags` field of the `file_lock` structure.
5. If the `l_type` field in the `flock` structure is equal to `F_RDLCK`, it checks whether the process is allowed to read from the file; similarly, if `l_type` is equal to `F_WRLCK`, checks whether the process is allowed to write into the file. If not, it returns an error code.
6. Invokes the `lock` method of the file operations, if defined. Usually for disk-based filesystems , this method is not defined.
7. Invokes `_ _posix_lock_file( )` passing as parameters the address of the file's inode object and the address of the `file_lock` object. This function performs, in turn, the following operations:
    a. Invokes `posix_locks_conflict( )` for each `FL_POSIX` lock in the inode's lock list. The function checks whether the lock conflicts with the requested one. Essentially, there must be no `FL_POSIX` write lock for the same region in the inode list, and there may be no `FL_POSIX` lock at all for the same region if the process is requesting a write lock. However, locks owned by the same process never conflict; this allows a process to change the characteristics of a lock it already owns.
    b. If a conflicting lock is found, the function checks whether `fcntl( )` was invoked with the `F_SETLKW` command. If so, the current process must be suspended: invokes `posix_locks_deadlock( )` to check that no deadlock condition is being created among processes waiting for `FL_POSIX` locks, then inserts the new lock (waiter lock) both in the blocker list of the conflicting lock (blocker lock) and in the blocked list, and finally returns an error code. Otherwise, if `fcntl( )` was invoked with the `F_SETLK` command, returns an error code.
    c. As soon as the inode's lock list includes no conflicting lock, the function checks all the `FL_POSIX` locks of the current process that overlap the file region that the current process wants to lock, and combines and splits adjacent areas as required. For example, if the process requested a write lock for a file region that falls inside a read-locked wider region, the previous read lock is split into two parts covering the nonoverlapping areas, while the central region is protected by the new write lock. In case of overlaps, newer locks always replace older ones.
    d. Inserts the new `file_lock` structure in the global file lock list and in the inode list.

    e. Returns the value 0 (success).
8. Checks the return code of _ `_posix_lock_file( )`:
    a. If the return code is 0 (no conflicting locks), it returns 0 (success).
    b. There are incompatibilities. If the `FL_SLEEP` flag in the `fl_flags` field is cleared, it releases the new `file_lock` descriptor and returns `-EAGAIN`.
    c. Otherwise, if there are incompatibilities but the process can sleep, it invokes `wait_event_interruptible( )` to insert the current process in the `lock->fl_wait` wait queue and to suspend it. When the process is awakened (right after the blocker lock has been released), it jumps to step 7 to retry the operation.

9. Chapter 16. Accessing Files

10. Accessing a disk-based file is a complex activity that involves the VFS abstraction layer (Chapter 12), handling block devices (Chapter 14), and the use of the page cache (Chapter 15). This chapter shows how the kernel builds on all those facilities to carry out file reads and writes. The topics covered in this chapter apply both to regular files stored in disk-based filesystems and to block device files; these two kinds of files will be referred to simply as "files."
11. The stage we are working at in this chapter starts after the proper read or write method of a particular file has been called (as described in Chapter 12). We show here how each read ends with the desired data delivered to a User Mode process and how each write ends with data marked ready for transfer to disk. The rest of the transfer is handled by the facilities described in Chapter 14 and Chapter 15.
12. There are many different ways to access a file. In this chapter we will consider the following cases:
13.
14. *Canonical mode*
15. The file is opened with the `O_SYNC` and `O_DIRECT` flags cleared, and its content is accessed by means of the `read( )` and `write( )` system calls. In this case, the `read( )` system call blocks the calling process until the data is copied into the User Mode address space (however, the kernel is always allowed to return fewer bytes than requested!). The `write( )` system call is different, because it terminates as soon as the data is copied into the page cache (deferred write). This case is covered in the section "Reading and Writing a File."
16.
17. *Synchronous mode*
18. The file is opened with the `O_SYNC` flag setor the flag is set at a later time by the `fcntl( )` system call. This flag affects only the write operation (read operations are always blocking), which blocks the calling process until the data is effectively written to disk. The section "Reading and Writing a File" covers this case, too.
19.
20. *Memory mapping mode*
21. After opening the file, the application issues an `mmap( )` system call to map the file into memory. As a result, the file appears as an array of bytes in RAM, and the application accesses directly the array elements instead of using `read( )` , `write( )`, or `lseek( )`. This case is discussed in the section "Memory Mapping."
22.
23. *Direct I/O mode*

24. The file is opened with the `O_DIRECT` flag set. Any read or write operation transfers data directly from the User Mode address space to disk, or vice versa, bypassing the page cache. We discuss this case in the section "Direct I/O Transfers." (The values of the `O_SYNC` and `O_DIRECT` flags can be combined in four meaningful ways.)
25.
26. *Asynchronous mode*
27. The file is accessedeither through a group of POSIX APIs or by means of Linux-specific system callsin such a way to perform "asynchronous I/O:" this means the requests for data transfers never block the calling process; rather, they are carried on "in the background" while the application continues its normal execution. We discuss this case in the section "Asynchronous I/O."

16.1. Reading and Writing a File

The section "The read( ) and write( ) System Calls" in Chapter 12 described how the `read( )` and `write( )` system calls are implemented. The corresponding service routines end up invoking the file object's `read` and `write` methods, which may be filesystem-dependent. For disk-based filesystems, these methods locate the physical blocks that contain the data being accessed and activate the block device driver to start the data transfer.

Reading a file is page-based: the kernel always transfers whole pages of data at once. If a process issues a `read( )` system call to get a few bytes, and that data is not already in RAM, the kernel allocates a new page frame, fills the page with the suitable portion of the file, adds the page to the page cache, and finally copies the requested bytes into the process address space. For most filesystems, reading a page of data from a file is just a matter of finding what blocks on disk contain the requested data. Once this is done, the kernel fills the pages by submitting the proper I/O operations to the generic block layer. In practice, the `read` method of all disk-based filesystems is implemented by a common function named `generic_file_read( )`.

Write operations on disk-based files are slightly more complicated to handle, because the file size could increase, and therefore the kernel might allocate some physical blocks on the disk. Of course, how this is precisely done depends on the filesystem type. However, many disk-based filesystems implement their `write` methods by means of a common function named `generic_file_write( )`. Examples of such filesystems are Ext2, System V /Coherent /Xenix , and MINIX . On the other hand, several other filesystems, such as journaling and network filesystems , implement the `write` method by means of custom functions.

16.1.1. Reading from a File

The `generic_file_read( )` function is used to implement the `read` method for block device files and for regular files of almost all disk-based filesystems. This function acts on the following parameters:

`filp`

Address of the file object

`buf`

Linear address of the User Mode memory area where the characters read from the file must be stored

`count`

Number of characters to be read

`ppos`

Pointer to a variable that stores the offset from which reading must start (usually the `f_pos` field of the `filp` file object)

As a first step, the function initializes two descriptors. The first descriptor is stored in the local variable `local_iov` of type `iovec`; it contains the address (`buf`) and the length (`count`) of the User Mode buffer that shall receive the data read from the file. The second descriptor is stored in the local variable `kiocb` of type `kiocb`; it is used to keep track of the completion status of an ongoing synchronous or asynchronous I/O operation. The main fields of the `kiocb` descriptor are shown in Table 16-1.

Table 16-1. The main fields of the kiocb descriptor

Type	Field	Description
`struct list_head`	`ki_run_list`	Pointers for the list of I/O operations to be retried later
`long`	`ki_flags`	Flags of the `kiocb` descriptor
`int`	`ki_users`	Usage counter of the `kiocb` descriptor
`unsigned int`	`ki_key`	Identifier of the asynchronous I/O operation, or `KIOCB_SYNC_KEY` (`0xffffffff`) for synchronous I/O operations
struct file *	ki_filp	Pointer to the file object associated with the ongoing I/O operation
struct kioctx *	ki_ctx	Pointer to the asynchronous I/O context descriptor for this operation (see the section "Asynchronous I/O" later in this chapter)

Table 16-1. The main fields of the kiocb descriptor

Type	Field	Description
int (*) (struct kiocb *, struct io_event *)	ki_cancel	Method invoked when canceling an asynchronous I/O operation
ssize_t (*) (struct kiocb *)	ki_retry	Method invoked when retrying an asynchronous I/O operation
void (*) (struct kiocb *)	ki_dtor	Method invoked when destroying the kiocb descriptor
struct list_head	ki_list	Pointers for the list of active ongoing I/O operation on an asynchronous I/O context
union	ki_obj	For synchronous operations, pointer to the process descriptor that issued the I/O operation; for asynchronous operations, pointer to the iocb User Mode data structure
_ _ u64	ki_user_data	Value to be returned to the User Mode process
loff_t	ki_pos	Current file position of the ongoing I/O operation
unsigned short	ki_opcode	Type of operation (read, write, or sync)
size_t	ki_nbytes	Number of bytes to be transferred
char *	ki_buf	Current position in the User Mode buffer
size_t	ki_left	Number of bytes yet to be transferred
wait_queue_t	ki_wait	Wait queue used for asynchronous I/O operations
void *	private	Freely usable by the filesystem layer

The generic_file_read( ) function initializes the kiocb descriptor by executing the init_sync_kiocb macro, which sets the fields of the object for a synchronous operation. In particular, the macro sets the ki_key field to KIOCB_SYNC_KEY, the ki_filp field to filp, and the ki_obj field to current.

Then, generic_file_read( ) invokes _ _generic_file_aio_read( ) passing to it the addresses of the iovec and kiocb descriptors just filled. The latter function returns a value, which is usually the number of bytes effectively read from the file; generic_file_read( ) terminates by returning this value.

The `_ _generic_file_aio_read( )` function is a general-purpose routine used by all filesystems to implement both synchronous and asynchronous read operations. The function receives four parameters: the address `iocb` of a `kiocb` descriptor, the address `iov` of an array of `iovec` descriptors, the length of this array, and the address `ppos` of a variable that stores the file's current pointer. When invoked by `generic_file_read( )`, the array of `iovec` descriptors is composed of just one element describing the User Mode buffer that will receive the data.[*]

[*] A variant of the `read( )` system callnamed `readv( )` allows an application to define multiple User Mode buffers in which the kernel scatters the data read from the file; the `_ _generic_file_aio_read( )` function handles this case, too. In the following, we will assume that the data read from the file will be copied into just one User Mode buffer; however, guessing the additional steps to be performed when using multiple buffers is straightforward.

We now explain the actions of the `_ _generic_file_aio_read( )` function; for the sake of simplicity, we restrict the description to the most common case: a synchronous operation raised by a `read( )` system call on a page-cached file. Later in this chapter we describe how this function behaves in other cases. As usual, we do not discuss how errors and anomalous conditions are handled.

Here are the steps performed by the function:

1. Invokes `access_ok( )` to verify that the User Mode buffer described by the `iovec` descriptor is valid. Because the starting address and length have been received from the `sys_read( )` service routine, they must be checked before using them (see the section "Verifying the Parameters" in Chapter 10). If the parameters are not valid, returns the `-EFAULT` error code.
2. Sets up a *read operation descriptor* namely, a data structure of type `read_descriptor_t` that stores the current status of the ongoing file read operation relative to a single User Mode buffer. The fields of this descriptor are shown in Table 16-2.
3. Invokes `do_generic_file_read( )`, passing to it the file object pointer `filp`, the pointer to the file offset `ppos`, the address of the just allocated read operation descriptor, and the address of the `file_read_actor( )` function (see later).
4. Returns the number of bytes copied into the User Mode buffer; that is, the value found in the `written` field of the `read_descriptor_t` data structure.

Table 16-2. The fields of the read operation descriptor

Type	Field	Description
`size_t`	`written`	How many bytes have been copied into the User Mode buffer
`size_t`	`count`	How many bytes are yet to be transferred
`char *`	`arg.buf`	Current position in the User Mode buffer
`int`	`error`	Error code of the read operation (0 for no error)

The `do_generic_file_read( )` function reads the requested pages from disk and copies them into the User Mode buffer. In particular, the function performs the following actions:

1. Gets the address_space object corresponding to the file being read; its address is stored in filp->f_mapping.
2. Gets the owner of the address_space object, that is, the inode object that will own the pages to be filled with file's data; its address is stored in the host field of the address_space object. If the file being read is a block device file, the owner is an inode in the *bdev* special filesystem rather than the inode pointed to by filp->f_dentry->d_inode (see "The address_space Object" in Chapter 15).
3. Considers the file as subdivided in pages of data (4,096 bytes per page). The function derives from the file pointer *ppos the logical number of the page that includes the first requested bytethat is, the page's index in the address spaceand stores it in the index local variable. The function also stores in the offset local variable the displacement inside the page of the first requested byte.
4. Starts a cycle to read all pages that include the requested bytes; the number of bytes to be read is stored in the count field of the read_descriptor_t descriptor. During a single iteration, the function transfers a page of data by performing the following substeps:
    a. If index*4096+offset exceeds the file size stored in the i_size field of the inode object, it exits from the cycle and goes to step 5.
    b. Invokes cond_resched( ) to check the TIF_NEED_RESCHED flag of the current process and, if the flag is set, to invoke the schedule( ) function.
    c. If additional pages must be read in advance, it invokes page_cache_readahead( ) to read them. We defer discussing read-ahead until the later section "Read-Ahead of Files."
    d. Invokes find_get_page( ) passing as parameters a pointer to the address_space object and the value of index; the function looks up the page cache to find the descriptor of the page that stores the requested data, if any.
    e. If find_get_page( ) returned a NULL pointer, the page requested is not in the page cache. In that case, it performs the following actions:
        1. Invokes handle_ra_miss( ) to tune the parameters used by the read-ahead system.
        2. Allocates a new page.
        3. Inserts the descriptor of the new page into the page cache by invoking add_to_page_cache( ). Remember that this function sets the PG_locked flag of the new page.
        4. Inserts the descriptor of the new page into the LRU list by invoking lru_cache_add( ) (see Chapter 17).
        5. Jumps to step 4j to start reading the file's data.
    f. If the function has reached this point, the page is in the page cache. Checks the PG_uptodate flag; if it is set, the data stored in the page is up-to-date, hence there is no need to read it from disk: jumps to step 4m.
    g. The data on the page is not valid, so it must be read from disk. The function gains exclusive access to the page by invoking the lock_page( ) function. As described in the section "Page Cache Handling Functions" in Chapter 15, lock_page( ) suspends the current process if the PG_locked flag is already set, until that bit is cleared.
    h. Now the page is locked by the current process. However, another process might have removed the page from the page cache right

before the previous step; hence, it checks whether the `mapping` field of the page descriptor is `NULL`; in this case, it unlocks the page by invoking `unlock_page( )`, decreases its usage counter (it was increased by `find_get_page( )`), and jumps back to step 4a starting over with the same page.

i. If the function has reached this point, the page is locked and still present in the page cache. Checks the `PG_uptodate` flag again, because another kernel control path could have completed the necessary read between steps 4f and 4g. If the flag is set, it invokes `unlock_page( )` and jumps to step 4m to skip the read operation.

j. Now the actual I/O operation can be started. Invokes the `readpage` method of the `address_space` object of the file. The corresponding function takes care of activating the I/O data transfer from the disk to the page. We discuss later what this function does for regular files and block device files.

k. If the `PG_uptodate` flag is still cleared, it waits until the page has been effectively read by invoking the `lock_page( )` function. The page, which was locked in step 4g, will be unlocked as soon as the read operation finishes. Therefore, the current process sleeps until the I/O data transfer terminates.

l. If `index` exceeds the file size in pages (this number is obtained by dividing the value of the `i_size` field of the inode object by 4,096), it decreases the page's usage counter, and exits from the cycle jumping to step 5. This case occurs when the file being read is concurrently truncated by another process.

m. Stores in the `nr` local variable the number of bytes in the page that should be copied into the User Mode buffer. This value is equal to the page size (4,096 bytes) unless either `offset` is not zerothis can happen only for the first or last page of requested dataor the file does not contain all requested bytes.

n. Invokes `mark_page_accessed( )` to set the `PG_referenced` or the `PG_active` flag, hence denoting the fact that the page is being used and should not be swapped out (see Chapter 17). If the same page (or part thereof) is read several times in successive executions of `do_generic_file_read( )`, this step is executed only during the first read.

o. Now it is time to copy the data on the page into the User Mode buffer. To do this, `do_generic_file_read( )` invokes the `file_read_actor( )` function, whose address has been passed as a parameter. In turn, `file_read_actor( )` essentially executes the following steps:

    1. Invokes `kmap( )`, which establishes a permanent kernel mapping for the page if it is in high memory (see the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8).
    2. Invokes `_ _copy_to_user( )`, which copies the data on the page in the User Mode address space (see the section "Accessing the Process Address Space" in Chapter 10). Notice that this operation might block the process because of page faults while accessing the User Mode address space.
    3. Invokes `kunmap( )` to release any permanent kernel mapping of the page.
    4. Updates the `count`, `written`, and `buf` fields of the `read_descriptor_t` descriptor.

    p. Updates the index and offset local variables according to the number of bytes effectively transferred in the User Mode buffer. Typically, if the last byte in the page has been copied into the User Mode buffer, index is increased by one and offset is set to zero; otherwise, index is not increased and offset is set to the number of bytes in the page that have been copied into the User Mode buffer.
    q. Decreases the page descriptor usage counter.
    r. If the count field of the read_descriptor_t descriptor is not zero, there is other data to be read from the file: jumps to step 4a to continue the loop with the next page of data in the file.
5. All requestedor availablebytes have been read. The function updates the filp->f_ra read-ahead data structure to record the fact that data is being read sequentially from the file (see the later section "Read-Ahead of Files").
6. Assigns to *ppos the value index*4096+offset, thus storing the next position where a sequential access is to occur for a future invocation of the read( ) and write( ) system calls.
7. Invokes update_atime( ) to store the current time in the i_atime field of the file's inode and to mark the inode as dirty, and returns.

16.1.1.1. The readpage method for regular files

As we saw, the readpage method is used repeatedly by do_generic_file_read( ) to read individual pages from disk into memory.

The readpage method of the address_space object stores the address of the function that effectively activates the I/O data transfer from the physical disk to the page cache. For regular files, this field typically points to a wrapper that invokes the mpage_readpage( ) function. For instance, the readpage method of the Ext3 filesystem is implemented by the following function:

```
int ext3_readpage(struct file *file, struct page *page)
{
    return mpage_readpage(page, ext3_get_block);
}
```

The wrapper is needed because the mpage_readpage( ) function receives as its parameters the descriptor page of the page to be filled and the address get_block of a function that helps mpage_readpage( ) find the right block. The wrapper is filesystem-specific and can therefore supply the proper function to get a block. This function translates the block numbers relative to the beginning of the file into logical block numbers relative to positions of the block in the disk partition (for an example, see Chapter 18). Of course, the latter parameter depends on the type of filesystem to which the regular file belongs; in the previous example, the parameter is the address of the ext3_get_block( ) function. The function passed as get_block always uses a buffer head to store precious information about the block device (b_dev field), the position of the requested data on the device (b_blocknr field), and the block status (b_state field).

The mpage_readpage( ) function chooses between two different strategies when reading a page from disk. If the blocks that contain the requested data are contiguously located on disk, then the function submits the read I/O operation to the generic block layer by using a single bio descriptor. In the opposite case, each block in the page is read by using a different bio descriptor. The filesystem-dependent `get_block` function plays the crucial role of determining whether the next block in the file is also the next block on the disk.

Specifically, `mpage_readpage( )` performs the following steps:

1. Checks the `PG_private` field of the page descriptor: if it is set, the page is a buffer page, that is, the page is associated with a list of buffer heads describing the blocks that compose the page (see the section "Storing Blocks in the Page Cache" in Chapter 15). This means that the page has already been read from disk in the past, and that the blocks in the page are not adjacent on disk: jumps to step 11 to read the page one block at a time.
2. Retrieves the block size (stored in the `page->mapping->host->i_blkbits` inode field), and computes two values required to access all blocks on that page: the number of blocks stored in the page and the file block number of the first block in the pagethat is, the index of the first block in the page relative to the beginning of the file.
3. For each block in the page, invokes the filesystem-dependent `get_block` function passed as a parameter to get the logical block number, that is, the index of the block relative to the beginning of the disk or partition. The logical block numbers of all blocks in the page are stored in a local array.
4. Checks for any anomalous condition that could occur while executing the previous step. In particular, if some blocks are not adjacent on disk, or some block falls inside a "file hole" (see the section "File Holes" in Chapter 18), or a block buffer has been already filled by the `get_block` function, then jumps to step 11 to read the page one block at a time.
5. If the function has reached this point, all blocks on the page are adjacent on disk. However, the page could be the last page of data in the file, hence some of the blocks in the page might not have an image on disk. If so, it fills the corresponding block buffers in the page with zeros; otherwise, it sets the `PG_mappedtodisk` flag of the page descriptor.
6. Invokes `bio_alloc( )` to allocate a new bio descriptor consisting of a single segment and to initialize its `bi_bdev` and `bi_sector` fields with the address of the block device descriptor and the logical block number of the first block in the page, respectively. Both pieces of information have been determined in step 3 above.
7. Sets the `bio_vec` descriptor of the bio's segment with the initial address of the page, the offset of the first byte to be read (zero), and the total number of bytes to be read.
8. Stores the address of the `mpage_end_io_read( )` function in the `bio->bi_end_io` field (see below).
9. Invokes `submit_bio( )`, which sets the `bi_rw` flag with the direction of the data transfer, updates the `page_states` per-CPU variable to keep track of the number of read sectors, and invokes the `generic_make_request( )` function on the bio descriptor (see the section "Issuing a Request to the I/O Scheduler" in Chapter 14).
10. Returns the value zero (success).
11. If the function jumps here, the page contains blocks that are not adjacent on disk. If the page is up-to-date (`PG_uptodate` flag set), the function invokes `unlock_page( )` to unlock the page; otherwise, it invokes

block_read_full_page( ) to start reading the page one block at a time (see below).
12. Returns the value zero (success).

The mpage_end_io_read( ) function is the completion method of the bio; it is executed as soon as the I/O data transfer terminates. Assuming that there was no I/O error, the function essentially sets the PG_uptodate flag of the page descriptor, invokes unlock_page( ) to unlock the page and to wake up any process sleeping for this event, and invokes bio_put( ) to destroy the bio descriptor.

16.1.1.2. The readpage method for block device files

In the sections "VFS Handling of Device Files" in Chapter 13 and "Opening a Block Device File" in Chapter 14, we discussed how the kernel handles requests to open a block device file. We saw how the init_special_inode( ) function sets up the device inode and how the blkdev_open( ) function completes the opening phase.

Block devices use an address_space object that is stored in the i_data field of the corresponding block device inode in the *bdev* special filesystem. Unlike regular files whose readpage method in the address_space object depends on the filesystem type to which the file belongs the readpage method of block device files is always the same. It is implemented by the blkdev_readpage( ) function, which calls block_read_full_page( ):

```
int blkdev_readpage(struct file * file, struct * page page)
{
    return block_read_full_page(page, blkdev_get_block);
}
```

As you can see, the function is once again a wrapper, this time for the block_read_full_page( ) function. This time the second parameter points to a function that translates the file block number relative to the beginning of the file into a logical block number relative to the beginning of the block device. For block device files, however, the two numbers coincide; therefore, the blkdev_get_block( ) function performs the following steps:

1. Checks whether the number of the first block in the page exceeds the index of the last block in the block device (this index is obtained by dividing the size of the block device stored in bdev->bd_inode->i_size by the block size stored in bdev->bd_block_size; bdev points to the descriptor of the block device). If so, it returns -EIO for a write operation, or zero for a read operation. (Reading beyond the end of a block device is not allowed, either, but the error code should not be returned here: the kernel could just be trying to dispatch a read request for the last data of a block device, and the corresponding buffer page is only partially mapped.)
2. Sets the b_dev field of the buffer head to bdev.
3. Sets the b_blocknr field of the buffer head to the file block number, which was passed as a parameter of the function.
4. Sets the BH_Mapped flag of the buffer head to state that the b_dev and b_blocknr fields of the buffer head are significant.

The `block_read_full_page( )` function reads a page of data one block at a time. As we have seen, it is used both when reading block device files and when reading pages of regular files whose blocks are not adjacent on disk. It performs the following steps:

1. Checks the `PG_private` flag of the page descriptor; if it is set, the page is associated with a list of buffer heads describing the blocks that compose the page (see the section "Storing Blocks in the Page Cache" in Chapter 15). Otherwise, the function invokes `create_empty_buffers( )` to allocate buffer heads for all block buffers included in the page. The address of the buffer head for the first buffer in the page is stored in the `page->private` field. The `b_this_page` field of each buffer head points to the buffer head of the next buffer in the page.
2. Derives from the file offset relative to the page (`page->index` field) the file block number of the first block in the page.
3. For each buffer head of the buffers in the page, it performs the following substeps:
    a. If the `BH_Uptodate` flag is set, it skips the buffer and continues with the next buffer in the page.
    b. If the `BH_Mapped` flag is not set and the block is not beyond the end of the file, it invokes the filesystem-dependent `get_block` function whose address has been passed as a parameter. For a regular file, the function looks in the on-disk data structures of the filesystem and finds the logical block number of the buffer relative to the beginning of the disk or partition. Conversely, for a block device file, the function regards the file block number as the logical block number. In both cases the function stores the logical block number in the `b_blocknr` field of the corresponding buffer head and sets the `BH_Mapped` flag.[*]

       [*] When accessing a regular file, the `get_block` function might not find the block if it falls in a "file hole" (see the section "File Holes" in Chapter 18). In this case, the function fills the block buffer with zeros and sets the `BH_Uptodate` flag of the buffer head.

    c. Tests again the `BH_Uptodate` flag because the filesystem-dependent `get_block` function could have triggered a block I/O operation that updated the buffer. If `BH_Uptodate` is set, it continues with the next buffer in the page.
    d. Stores the address of the buffer head in `arr` local array, and continues with the next buffer in the page.
4. If no file hole has been encountered in the previous step, the function sets the `PG_mappedtodisk` flag of the page.
5. Now the `arr` local array stores the addresses of the buffer heads that correspond to the buffers whose content is not up-to-date. If this array is empty, all buffers in the page are valid. So the function sets the `PG_uptodate` flag of the page descriptor, unlocks the page by invoking `unlock_page( )`, and returns.
6. The `arr` local array is not empty. For each buffer head in the array, `block_read_full_page( )` performs the following substeps:
    a. Sets the `BH_Lock` flag. If the flag was already set, the function waits until the buffer is released.

    b. Sets the `b_end_io` field of the buffer head to the address of the `end_buffer_async_read( )` function (see below) and sets the `BH_Async_Read` flag of the buffer head.
7. For each buffer head in the `arr` local array, it invokes the `submit_bh( )` function on it, specifying the operation type `READ`. As we saw earlier, this function triggers the I/O data transfer of the corresponding block.
8. Returns 0.

The `end_buffer_async_read( )` function is the completion method of the buffer head; it is executed as soon as the I/O data transfer on the block buffer terminates. Assuming that there was no I/O error, the function sets the `BH_Uptodate` flag of the buffer head and clears the `BH_Async_Read` flag. Then, the function gets the descriptor of the buffer page containing the block buffer (its address is stored in the `b_page` field of the buffer head) and checks whether all blocks in the page are up-to-date; if so, the function sets the `PG_uptodate` flag of the page and invokes `unlock_page( )`.

16.1.2. Read-Ahead of Files

Many disk accesses are sequential. As we will see in Chapter 18, regular files are stored on disk in large groups of adjacent sectors, so that they can be retrieved quickly with few moves of the disk heads. When a program reads or copies a file, it often accesses it sequentially, from the first byte to the last one. Therefore, many adjacent sectors on disk are likely to be fetched when handling a series of a process's read requests on the same file.

Read-ahead consists of reading several adjacent pages of data of a regular file or block device file *before* they are actually requested. In most cases, read-ahead significantly enhances disk performance, because it lets the disk controller handle fewer commands, each of which refers to a larger chunk of adjacent sectors. Moreover, it improves system responsiveness. A process that is sequentially reading a file does not usually have to wait for the requested data because it is already available in RAM.

However, read-ahead is of no use when an application performs random accesses to files; in this case, it is actually detrimental because it tends to waste space in the page cache with useless information. Therefore, the kernel reducesor stopsread-ahead when it determines that the most recently issued I/O access is not sequential to the previous one.

Read-ahead of files requires a sophisticated algorithm for several reasons:

- Because data is read page by page, the read-ahead algorithm does not have to consider the offsets inside the page, but only the positions of the accessed pages inside the file.
- Read-ahead may be gradually increased as long as the process keeps accessing the file sequentially.
- Read-ahead must be scaled down or even disabled when the current access is not sequential with respect to the previous one (random access).
- Read-ahead should be stopped when a process keeps accessing the same pages over and over again (only a small portion of the file is being used), or when almost all pages of the file are already in the page cache.

- The low-level I/O device driver should be activated at the proper time, so that the future pages will have been transferred when the process needs them.

The kernel considers a file access as *sequential* with respect to the previous file access if the first page requested is the page following the last page requested in the previous access.

While accessing a given file, the read-ahead algorithm makes use of two sets of pages, each of which corresponds to a contiguous portion of the file. These two sets are called the *current window* and the *ahead window* .

The current window consists of pages requested by the process or read in advance by the kernel and included in the page cache. (A page in the current window is not necessarily up-to-date, because its I/O data transfer could be still in progress.) The current window contains both the last pages sequentially accessed by the process and possibly some of the pages that have been read in advance by the kernel but that have not yet been requested by the process.

The ahead window consists of pagesfollowing the ones in the current windowthat are being currently being read in advance by the kernel. No page in the ahead window has yet been requested by the process, but the kernel assumes that sooner or later the process will request them.

When the kernel recognizes a sequential access and the initial page belongs to the current window, it checks whether the ahead window has already been set up. If not, the kernel creates a new ahead window and triggers the read operations for the corresponding pages. In the ideal case, the process still requests pages from the current window while the pages in the ahead window are being transferred. When the process requests a page included in the ahead window, the ahead window becomes the new current window.

The main data structure used by the read-ahead algorithm is the `file_ra_state` descriptor whose fields are listed in Table 16-3. Each file object includes such a descriptor in its `f_ra` field.

Table 16-3. The fields of the file_ra_state descriptor

Type	Field	Description
`unsigned long`	`start`	Index of first page in the current window
`unsigned long`	`size`	Number of pages included in the current window (-1 for read-ahead temporarily disabled, 0 for empty current window)
`unsigned long`	`flags`	Flags used to control the read-ahead
unsigned long	cache_hit	Number of consecutive cache hits (pages requested by the process and found in the page cache)
`unsigned long`	`prev_page`	Index of the last page requested by the process

Table 16-3. The fields of the file_ra_state descriptor

Type	Field	Description
`unsigned long`	ahead_start	Index of the first page in the ahead window
`unsigned long`	ahead_size	Number of pages in the ahead window (0 for an empty ahead window)
`unsigned long`	ra_pages	Maximum size in pages of a read-ahead window (0 for read-ahead permanently disabled)
`unsigned long`	mmap_hit	Read-ahead hit counter (used for memory mapped files)
`unsigned long`	mmap_miss	Read-ahead miss counter (used for memory mapped files)

When a file is opened, all the fields of its `file_ra_state` descriptor are set to zero except the `prev_page` and `ra_pages` fields.

The `prev_page` field stores the index of the last page requested by the process in the previous read operation; initially, the field contains the value `-1`.

The `ra_pages` field represents the maximum size in pages for the current window, that is, the maximum read-ahead allowed for the file. The initial (default) value for this field is stored in the `backing_dev_info` descriptor of the block device that includes the file (see the section "Request Queue Descriptors" in Chapter 14). An application can tune the read-ahead algorithm for a given opened file by modifying the `ra_pages` field; this can be done by invoking the `posix_fadvise( )` system call, passing to it the commands `POSIX_FADV_NORMAL` (set read-ahead maximum size to default, usually 32 pages), `POSIX_FADV_SEQUENTIAL` (set read-ahead maximum size to two times the default), and `POSIX_FADV_RANDOM` (set read-ahead maximum size to zero, thus permanently disabling read-ahead).

The `flags` field contains two flags called `RA_FLAG_MISS` and `RA_FLAG_INCACHE` that play an important role. The first flag is set when a page that has been read in advance is not found in the page cache (likely because it has been reclaimed by the kernel in order to free memory; see Chapter 17): in this case, the size of the next ahead window to be created is somewhat reduced. The second flag is set when the kernel determines that the last 256 pages requested by the process have all been found in the page cache (the value of consecutive cache hits is stored in the `ra->cache_hit` field). In this case, read-ahead is turned off because the kernel assumes that all the pages required by the process are already in the cache.

When is the read-ahead algorithm executed? This happens in the following cases:

- When the kernel handles a User Mode request to read pages of file data; this event triggers the invocation of the `page_cache_readahead( )` function (see step 4c in the description of the `do_generic_file_read( )` function in the section "Reading from a File" earlier in this chapter).

- When the kernel allocates a page for a file memory mapping (see the filemap_nopage( ) function in the section "Demand Paging for Memory Mapping" later in this chapter, which again invokes the page_cache_readahead( ) function).
- When a User Mode application executes the readahead( ) system call, which explicitly triggers some read-ahead activity on a file descriptor.
- When a User Mode application executes the posix_fadvise( ) system call with the POSIX_FADV_NOREUSE or POSIX_FADV_WILLNEED commands, which inform the kernel that a given range of file pages will be accessed in the near future.
- When a User Mode application executes the madvise( ) system call with the MADV_WILLNEED command, which informs the kernel that a given range of pages in a file memory mapping region will be accessed in the near future.

16.1.2.1. The page_cache_readahead( ) function

The page_cache_readahead( ) function takes care of all read-ahead operations that are not explicitly triggered by ad-hoc system calls. It replenishes the current and ahead windows, updating their sizes according to the number of read-ahead hits, that is, according to how successful the read-ahead strategy was in the past accesses to the file.

The function is invoked when the kernel must satisfy a read request for one or more pages of a file, and acts on five parameters:

mapping

Pointer to the address_space object that describes the owner of the page

ra

Pointer to the file_ra_state descriptor of the file containing the page

filp

Address of the file object

offset

Offset of the page within the file

`req_size`

> Number of pages yet to be read to complete the current read operation[*]
>
> [*] Actually, if the read operation involves a number of pages larger than the maximum size of the read-ahead window, the `page_cache_readahead( )` function is invoked several times. Thus, the `req_size` parameter might be smaller than the number of pages yet to be read to complete the read operation.

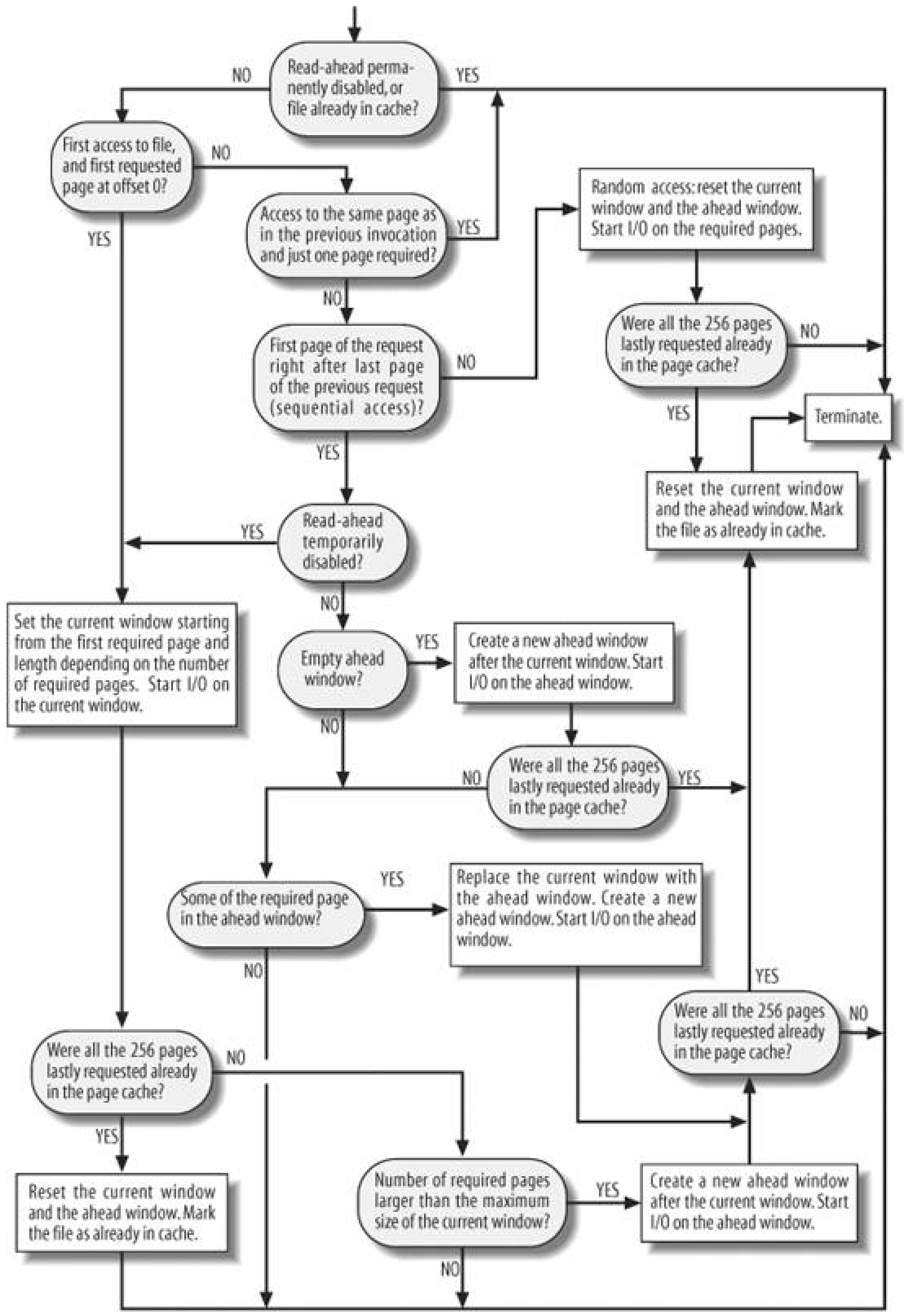
Figure 16-1 shows the flow diagram of `page_cache_readahead( )`. The function essentially acts on the fields of the `file_ra_state` descriptor; thus, although the description of the actions in the flow diagram is quite informal, you can easily determine the actual steps performed by the function. For instance, in order to check whether the requested page is the same as the page previously read, the function checks whether the values of the `ra->prev_page` field and of the `offset` parameter coincide (see Table 16-3 earlier).

When the process accesses the file for the first time and the first requested page is the page at offset zero in the file, the function assumes that the process will perform sequential accesses. Thus, the function creates a new current window starting from the first page. The length of the initial current windowalways a power of twois somewhat related to the number of pages requested by the process in the first read operation: the higher the number of requested pages, the larger the current window, up to the maximum value stored in the `ra->ra_pages` field. Conversely, when the process accesses the file for the first time but the first requested page is not at offset zero, the function assumes that the process will not perform sequential accesses. Thus, the function temporarily disables read-ahead (`ra->size` field is set to -1). However, a new current window is created when the function recognizes a sequential access while read-ahead is temporarily disabled.

If the ahead window does not already exist, it is created as soon as the function recognizes that the process has performed a sequential access in the current window. The ahead window always starts from the page following the last page of the current window. Its length, however, is related to the length of the current window as follows: if the `RA_FLAG_MISS` flag is set, the length of the ahead window is the length of the current window minus 2, or four pages if the result is less than four; otherwise, the length of the ahead window is either four times or two times the length of the current window. If the process continues to access the file in a sequential way, eventually the ahead window becomes the new current window, and a new ahead window is created. Thus, read-ahead is aggressively enhanced if the process reads the file sequentially.

As soon as the function recognizes a file access that is not sequential with respect to the previous one, the current and ahead windows are cleared (emptied) and the read-ahead is temporarily disabled. Read-ahead is restarted from scratch as soon as the process performs a read operation that is sequential with respect to the previous file access.

Figure 16-1. The flow diagram of the page_cache_readahead( ) function


Read-ahead permanently disabled, or file already in cache?
NO
YES
First access to file, and first requested page at offset 0?
NO
YES
Access to the same page as in the previous invocation and just one page required?
YES
NO
Random access: reset the current window and the ahead window. Start I/O on the required pages.
First page of the request right after last page of the previous request (sequential access)?
NO
YES
Were all the 256 pages lastly requested already in the page cache?
NO
YES
Terminate.
Reset the current window and the ahead window. Mark the file as already in cache.
Read-ahead temporarily disabled?
YES
NO
Set the current window starting from the first required page and length depending on the number of required pages. Start I/O on the current window.
Empty ahead window?
YES
NO
Create a new ahead window after the current window. Start I/O on the ahead window.
Were all the 256 pages lastly requested already in the page cache?
NO
YES
Some of the required page in the ahead window?
YES
NO
Replace the current window with the ahead window. Create a new ahead window. Start I/O on the ahead window.
Were all the 256 pages lastly requested already in the page cache?
YES
NO
Were all the 256 pages lastly requested already in the page cache?
NO
YES
Reset the current window and the ahead window. Mark the file as already in cache.
Number of required pages larger than the maximum size of the current window?
YES
NO
Create a new ahead window after the current window. Start I/O on the ahead window.

Every time `page_cache_readahead( )` creates a new window, it starts the read operations for the included pages. In order to read a chunk of pages, `page_cache_readahead( )` invokes the `blockable_page_cache_readahead( )` function. To reduce kernel overhead, the latter function adopts the following clever features:

- No reading is performed if the request queue that services the block device is read-congested (it does not make sense to increase congestion and block read-ahead).
- The page cache is checked against each page to be read; if the page is already in the page cache, it is simply skipped over.
- All the page frames needed by the read request are allocated at once before performing the read from disk. If not all page frames can be obtained, the read-ahead operation is performed only on the available pages. Again, there is little sense in deferring read-ahead until all page frames become available.
- Whenever possible, the read operations are submitted to the generic block layer by using multi-segment bio descriptors (see the section "Segments" in Chapter 14). This is done by the specialized `readpages` method of the `address_space` object, if defined; otherwise, it is done by repeatedly invoking the `readpage` method. The `readpage` method is described in the earlier section "Reading from a File" for the single-segment case only, but it is easy to adapt the description for the multi-segment case.

16.1.2.2. The handle_ra_miss( ) function

In some cases, the kernel must correct the read-ahead parameters, because the read-ahead strategy does not seem very effective. Let us consider the `do_generic_file_read( )` function described in the section "Reading from a File" earlier in this chapter. The `page_cache_readahead( )` function is invoked in step 4c. The flow diagram in Figure 16-1 depicts two cases: either the requested page is in the current window or in the ahead window, hence it should have been read in advance, or it is not, and the function invokes `blockable_page_cache_readahead( )` to read it. In both cases, `do_generic_file_read( )` should find the page in the page cache in step 4d. If it is not found, this means that the page frame reclaiming algorithm has removed the page from the cache. In this case, `do_generic_file_read( )` invokes the `handle_ra_miss( )` function, which tunes the read-ahead algorithm by setting the `RA_FLAG_MISS` flag and by clearing the `RA_FLAG_INCACHE` flag.

16.1.3. Writing to a File

Recall that the `write( )` system call involves moving data from the User Mode address space of the calling process into the kernel data structures, and then to disk. The `write` method of the file object permits each filesystem type to define a specialized write operation. In Linux 2.6, the `write` method of each disk-based filesystem is a procedure that basically identifies the disk blocks involved in the write operation, copies the data from the User Mode address space into some pages belonging to the page cache, and marks the buffers in those pages as dirty.

Many filesystems (including Ext2 or JFS) implement the `write` method of the file object by means of the `generic_file_write( )` function, which acts on the following parameters:

`file`

File object pointer

`buf`

Address in the User Mode address space where the characters to be written into the file must be fetched

`count`

Number of characters to be written

`ppos`

Address of a variable storing the file offset from which writing must start

The function performs the following steps:

1. Initializes a local variable of type `iovec` containing the address and length of the User Mode buffer (see also the description of the `generic_file_read( )` function in the section "Reading from a File" earlier in this chapter).
2. Determines the address `inode` of the inode object that corresponds to the file to be written (`file->f_mapping->host`) and acquires the semaphore `inode->i_sem`. Thanks to this semaphore, only one process at a time can issue a `write( )` system call on the file.
3. Invokes the `init_sync_kiocb` macro to initialize a local variable of type `kiocb`. As explained in the section "Reading from a File" earlier in this chapter, the macro sets the `ki_key` field to `KIOCB_SYNC_KEY` (synchronous I/O operation), the `ki_filp` field to `file`, and the `ki_obj` field to `current`.
4. Invokes `_ _generic_file_aio_write_nolock( )` (see below) to mark the affected pages as dirty, passing the address of the local variables of type `iovec` and `kiocb`, the number of segments for the User Mode bufferonly one in this caseand the parameter `ppos`.
5. Releases the `inode->i_sem` semaphore.
6. Checks the `O_SYNC` flag of the file, the `S_SYNC` flag of the inode, and the `MS_SYNCHRONOUS` flag of the superblock; if at least one of them is set, it invokes the `sync_page_range( )` function to force the kernel to flush all pages in the page cache that have been touched in step 4, blocking the current process until the I/O data transfers terminate. In turn, `sync_page_range( )` executes either the `writepages` method of the `address_space` object, if defined, or the `mpage_writepages( )` function (see the section "Writing Dirty Pages to Disk"

later in this chapter) to start the I/O transfers for the dirty pages; then, it invokes `generic_osync_inode( )` to flush to disk the inode and the associated buffers, and finally invokes `wait_on_page_bit( )` to suspend the current process until all `PG_writeback` bits of the flushed pages are cleared.
7. Returns the code returned by `_ _generic_file_aio_write_nolock( )`, usually the number of bytes effectively written.

The `_ _generic_file_aio_write_nolock( )` function receives four parameters: the address `iocb` of a `kiocb` descriptor, the address `iov` of an array of `iovec` descriptors, the length of this array, and the address `ppos` of a variable that stores the file's current pointer. When invoked by `generic_file_write( )`, the array of `iovec` descriptors is composed of just one element describing the User Mode buffer that contains the data to be written.[*]

[*] A variant of the `write( )` system callnamed `writev( )` allows an application to define multiple User Mode buffers from which the kernel fetches the data to be written on the file; the `generic_file_aio_write_nolock( )` function handles this case too. In the following pages, we will assume that the data will be fetched from just one User Mode buffer; however, guessing the additional steps to be performed when using multiple buffers is straightforward.

We now explain the actions of the `_ _generic_file_aio_write_nolock( )` function; for the sake of simplicity, we restrict the description to the most common case: a common mode operation raised by a `write( )` system call on a page-cached file. Later in this chapter we describe how this function behaves in other cases. As usual, we do not discuss how errors and anomalous conditions are handled.

The function executes the following steps:

1. Invokes `access_ok( )` to verify that the User Mode buffer described by the `iovec` descriptor is valid (the starting address and length have been received from the `sys_write( )` service routine, thus they must be checked before using them; see the section "Verifying the Parameters" in Chapter 10). If the parameters are not valid, it returns the `-EFAULT` error code.
2. Determines the address `inode` of the inode object that corresponds to the file to be written (`file->f_mapping->host`). Remember that if the file is a block device file, this is an inode in the *bdev* special filesystem (see Chapter 14).
3. Sets `current->backing_dev_info` to the address of the `backing_dev_info` descriptor of the file (`file->f_mapping->backing_dev_info`). Essentially, this setting allows the current process to write back the dirty pages owned by `file->f_mapping` even if the corresponding request queue is congested; see Chapter 17.
4. If the `O_APPEND` flag of `file->flags` is on and the file is regular (not a block device file), it sets `*ppos` to the end of the file so that all new data is appended to it.
5. Performs several checks on the size of the file. For instance, the write operation must not enlarge a regular file so much as to exceed the per-user limit stored in `current->signal->rlim[RLIMIT_FSIZE]` (see the section "Process Resource Limits" in Chapter 3) and the filesystem limit stored in `inode->i_sb->s_maxbytes`. Moreover, if the file is not a "large file" (flag `O_LARGEFILE` of `file->f_flags` cleared), its size cannot exceed 2 GB. If any of these constraints is not enforced, it reduces the number of bytes to be written.

6. If set, it clears the `suid` flag of the file; also clears the `sgid` flag if the file is executable (see the section "Access Rights and File Mode" in Chapter 1). We don't want users to be able to modify *setuid* files.
7. Stores the current time of day in the `inode->mtime` field (the time of last file write operation) and in the `inode->ctime` field (the time of last inode change), and marks the inode object as dirty.
8. Starts a cycle to update all the pages of the file involved in the write operation. During each iteration, it performs the following substeps:
    a. Invokes `find_lock_page( )` to search the page in the page cache (see the section "Page Cache Handling Functions" in Chapter 15). If this function finds the page, it increases its usage counter and sets its `PG_locked` flag.
    b. If the page is not in the page cache, it allocates a new page frame and invokes `add_to_page_cache( )` to insert the page into the page cache; as explained in the section "Page Cache Handling Functions" in Chapter 15, this function also increases the usage counter and sets the `PG_locked` flag. Moreover, the function inserts the new page into the inactive list of the memory zone (see Chapter 17).
    c. Invokes the `prepare_write` method of the `address_space` object of the inode (`file->f_mapping`). The corresponding function takes care of allocating and initializing buffer heads for the page. We'll discuss in subsequent sections what this function does for regular files and block device files.
    d. If the buffer is in high memory, it establishes a kernel mapping of the User Mode buffer (see the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8). Then, it invokes `_ _copy_from_user( )` to copy the characters from the User Mode buffer to the page, and releases the kernel mapping.
    e. Invokes the `commit_write` method of the `address_space` object of the inode (`file->f_mapping`). The corresponding function marks the underlying buffers as dirty so they are written to disk later. We discuss what this function does for regular files and block device files in the next two sections.
    f. Invokes `unlock_page( )` to clear the `PG_locked` flag and wake up any process that is waiting for the page.
    g. Invokes `mark_page_accessed( )` to update the page status for the memory reclaiming algorithm (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).
    h. Decreases the page usage counter to undo the increment in step 8a or 8b.
    i. In this iteration, yet another page has been dirtied: it checks whether the ratio of dirty pages in the page cache has risen above a fixed threshold (usually, 40% of the pages in the system); if so, it invokes `writeback_inodes( )` to start flushing a few tens of pages to disk (see the section "Looking for Dirty Pages To Be Flushed" in Chapter 15).
    j. Invokes `cond_resched( )` to check the `TIF_NEED_RESCHED` flag of the current process and, if the flag is set, to invoke the `schedule( )` function.
9. Now all pages of the file involved in the write operation have been handled.Updates the value of `*ppos` to point right after the last character written.
10. Sets `current->backing_dev_info` to `NULL` (see step 3).

11. Terminates by returning the number of bytes effectively written.

16.1.3.1. The prepare_write and commit_write methods for regular files

The prepare_write and commit_write methods of the address_space object specialize the generic write operation implemented by generic_file_write( ) for regular files and block device files. Both of them are invoked once for every page of the file that is affected by the write operation.

Each disk-based filesystem defines its own prepare_write method. As with read operations, this method is simply a wrapper for a common function. For instance, the Ext2 filesystem usually implements the prepare_write method by means of the following function:

```
int ext2_prepare_write(struct file *file, struct page *page,
                       unsigned from, unsigned to)
{
    return block_prepare_write(page, from, to, ext2_get_block);
}
```

The ext2_get_block( ) function was already mentioned in the earlier section "Reading from a File"; it translates the block number relative to the file into a logical block number, which represents the position of the data on the physical block device.

The block_prepare_write( ) function takes care of preparing the buffers and the buffer heads of the file's page by performing essentially the following steps:

1. Checks if the page is a buffer page (flag PG_Private set); if this flag is cleared, invokes create_empty_buffers( ) to allocate buffer heads for all buffers included in the page (see the section "Buffer Pages" in Chapter 15).
2. For each buffer head relative to a buffer included in the page and affected by the write operation, the following is performed:
    a. Resets the BH_New flag, if it is set (see below).
    b. If the BH_Mapped flag is not set, the function performs the following substeps:
        1. Invokes the filesystem-dependent function whose address get_block was passed as a parameter. This function looks in the on-disk data structures of the filesystem and finds the logical block number of the buffer (relative to the beginning of the disk partition rather than the beginning of the regular file). The filesystem-dependent function stores this number in the b_blocknr field of the corresponding buffer head and sets its BH_Mapped flag. The get_block function could allocate a new physical block for the file (for instance, if the accessed block falls inside a "hole" of the regular file; see the section "File Holes" in Chapter 18). In this case, it sets the BH_New flag.
        2. Checks the value of the BH_New flag; if it is set, invokes unmap_underlying_metadata( ) to check whether some block device buffer page in the page cache includes a buffer

referencing the same block on disk.[*] This function essentially invokes `_ _find_get_block( )` to look up the old block in the page cache (see the section "Searching Blocks in the Page Cache" in Chapter 15). If such a block is found, the function clears its `BH_Dirty` flag and waits until any I/O data transfer on that buffer completes. Moreover, if the write operation does not rewrite the whole buffer in the page, it fills the unwritten portion with zero's. Then it considers the next buffer in the page.

[*] Although unlikely, this case might happen if a user writes blocks directly on the block device file, thus bypassing the filesystem.

   c. If the write operation does not rewrite the whole buffer and its `BH_Delay` and `BH_Uptodate` flags are not set (that is, the block has been allocated in the on-disk filesystem data structures and the buffer in RAM does not contain a valid image of the data), the function invokes `ll_rw_block( )` on the block to read its content from disk (see the section "Submitting Buffer Heads to the Generic Block Layer" in Chapter 15).

3. Blocks the current process until all read operations triggered in step 2c have been completed.
4. Returns 0.

Once the `prepare_write` method returns, the `generic_file_write( )` function updates the page with the data stored in the User Mode address space. Next, it invokes the `commit_write` method of the `address_space` object. This method is implemented by the `generic_commit_write( )` function for almost all disk-based non-journaling filesystems.

The `generic_commit_write( )` function performs essentially the following steps:

1. Invokes the `_ _block_commit_write( )` function. In turn, this function does the following:
   a. Considers all buffers in the page that are affected by the write operation; for each of them, sets the `BH_Uptodate` and `BH_Dirty` flags of the corresponding buffer head.
   b. Marks the corresponding inode as dirty. As seen in the section "Looking for Dirty Pages To Be Flushed" in Chapter 15, this activity may require adding the inode to the list of dirty inodes of the superblock.
   c. If all buffers in the buffer page are now up-to-date, it sets the `PG_uptodate` flag of the page.
   d. Sets the `PG_dirty` flag of the page, and tags the page as dirty in its radix tree (see the section "The Radix Tree" in Chapter 15).
2. Checks whether the write operation enlarged the file. In this case, the function updates the `i_size` field of the file's inode.
3. Returns 0.

16.1.3.2. The prepare_write and commit_write methods for block device files

Write operations into block device files are very similar to the corresponding operations on regular files. In fact, the prepare_write method of the address_space object of block device files is usually implemented by the following function:

```
int blkdev_prepare_write(struct file *file, struct page *page,
                         unsigned from, unsigned to)
{
    return block_prepare_write(page, from, to, blkdev_get_block);
}
```

As you can see, the function is simply a wrapper to the block_prepare_write( ) function already discussed in the previous section. The only difference, of course, is in the second parameter, which points to the function that must translate the file block number relative to the beginning of the file to a logical block number relative to the beginning of the block device. Remember that for block device files, the two numbers coincide. (See the earlier section "Reading from a File" for a discussion of the blkdev_get_block( ) function.)

The commit_write method for block device files is implemented by the following simple wrapper function:

```
int blkdev_commit_write(struct file *file, struct page *page,
                        unsigned from, unsigned to)
{
    return block_commit_write(page, from, to);
}
```

As you can see, the commit_write method for block device files does essentially the same things as the commit_write method for regular files (we described the block_commit_write( ) function in the previous section). The only difference is that the method does not check whether the write operation has enlarged the file; you simply cannot enlarge a block device file by appending characters to its last position.

16.1.4. Writing Dirty Pages to Disk

The net effect of the write( ) system call consists of modifying the contents of some pages in the page cacheoptionally allocating the pages and adding them to the page cache if they were not already present. In some cases (for instance, if the file has been opened with the O_SYNC flag), the I/O data transfers start immediately (see step 6 of generic_file_write( ) in the section "Writing to a File" earlier in this chapter). Usually, however, the I/O data transfer is delayed, as explained in the section "Writing Dirty Pages to Disk" in Chapter 15.

When the kernel wants to effectively start the I/O data transfer, it ends up invoking the writepages method of the file's address_space object, which searches for dirty pages in the radix-tree and flushes them to disk. For instance, the Ext2 filesystem implements the writepages method by means of the following function:

```
int ext2_writepages(struct address_space *mapping,
                     struct writeback_control *wbc)
{
    return mpage_writepages(mapping, wbc, ext2_get_block);
}
```

As you can see, this function is a simple wrapper for the general-purpose `mpage_writepages( )` function; as a matter of fact, if a filesystem does not define the `writepages` method, the kernel invokes directly `mpage_writepages( )` passing `NULL` as third argument. The `ext2_get_block( )` function was already mentioned in the earlier section "Reading from a File;" it is the filesystem-dependent function that translates a file block number into a logical block number.

The `writeback_control` data structure is a descriptor that controls how the writeback operation has to be performed; we have already described it in the section "Looking for Dirty Pages To Be Flushed" in Chapter 15.

The `mpage_writepages( )` function essentially performs the following actions:

1. If the request queue is write-congested and the process does not want to block, it returns without writing any page to disk.
2. Determines the file's initial page to be considered. If the `writeback_control` descriptor specifies the initial position in the file, the function translates it into a page index. Otherwise, if the `writeback_control` descriptor specifies that the process does not want to wait for the I/O data transfer to complete, it sets the initial page index to the value stored in `mapping->writeback_index` (that is, scanning begins from the last page considered in the previous writeback operation). Finally, if the process must wait until I/O data transfers complete, scanning starts from the first page of the file.
3. Invokes `find_get_pages_tag( )` to look up the descriptor of the dirty pages in the page cache (see the section "The Tags of the Radix Tree" in Chapter 15).
4. For each page descriptor retrieved in the previous step, the function performs the following steps:
    a. Invokes `lock_page( )` to lock up the page.
    b. Checks that the page is still valid and in the page cache (because another kernel control path could have acted upon the page between steps 3 and 4a).
    c. Checks the `PG_writeback` flag of the page. If it is set, the page is already being flushed to disk. If the process must wait for the I/O data transfer to complete, it invokes `wait_on_page_bit( )` to block the current process until the `PG_writeback` flag is cleared; when this function terminates, any previously ongoing writeback operation is terminated. Otherwise, if the process does not want to wait, it checks the `PG_dirty` flag: if it is now cleared, the on-going writeback will take care of the page, thus unlocks it and jumps back to step 4a to continue with the next page.
    d. If the `get_block` parameter is `NULL` (no `writepages` method defined), it invokes the `mapping->writepage` method of the `address_space` object of the file to flush the page to disk. Otherwise, if the `get_block` parameter is not `NULL`, it invokes the `mpage_writepage( )` function. See step 8 for details.

5. Invokes `cond_resched( )` to check the `TIF_NEED_RESCHED` flag of the current process and, if the flag is set, to invoke the `schedule( )` function.
6. If the function has not scanned all pages in the given range, or if the number of pages effectively written to disk is smaller than the value originally specified in the `writeback_control` descriptor, it jumps back to step 3.
7. If the `writeback_control` descriptor does not specify the initial position in the file, it sets the `mapping->writeback_index` field with the index of the last scanned page.
8. If the `mpage_writepage( )` function has been invoked in step 4d, and if that function returned the address of a bio descriptor, it invokes `mpage_bio_submit( )` (see below).

A typical filesystem such as Ext2 implements the `writepage` method as a wrapper for the general-purpose `block_write_full_page( )` function, passing to it the address of the filesystem-dependent `get_block` function. In turn, the `block_write_full_page( )` function is similar to `block_read_full_page( )` described in the section "Reading from a File" earlier in this chapter: it allocates buffer heads for the page (if the page was not already a buffer page), and invokes the `submit_bh( )` function on each of them, specifying the `WRITE` operation. As far as block device files are concerned, they implement the `writepage` method by using `blkdev_writepage( )`, which is a wrapper for `block_write_full_page( )`.

Many non-journaling filesystems rely on the `mpage_writepage( )` function rather than on the custom `writepage` method. This can improve performance because the `mpage_writepage( )` function tries to submit the I/O transfers by collecting as many pages as possible in the same bio descriptor; in turn, this allows the block device drivers to exploit the scatter-gather DMA capabilities of the modern hard disk controllers.

To make a long story short, the `mpage_writepage( )` function checks whether the page to be written contains blocks that are not adjacent to disk, or whether the page includes a file hole, or whether some block on the page is not dirty or not up-to-date. If at least one of these conditions holds, the function falls back on the filesystem-dependent `writepage` method, as above. Otherwise, the function adds the page as a segment of a bio descriptor. The address of the bio descriptor is passed as parameter to the function; if it is `NULL`, `mpage_writepage( )` initializes a new bio descriptor and returns its address to the calling function, which in turn passes it back in the future invocations of `mpage_writepage( )`. In this way, several pages can be added to the same bio. If a page is not adjacent to the last added page in the bio, `mpage_writepage( )` invokes `mpage_bio_submit( )` to start the I/O data transfer on the bio, and allocates a new bio for the page.

The `mpage_bio_submit( )` function sets the `bi_end_io` method of the bio to the address of `mpage_end_io_write( )`, then invokes `submit_bio( )` to start the transfer (see the section "Submitting Buffer Heads to the Generic Block Layer" in Chapter 15). Once the data transfer successfully terminates, the completion function `mpage_end_io_write( )` wakes up any process waiting for the page transfer to complete, and destroys the bio descriptor.

16.2. Memory Mapping

As already mentioned in the section "Memory Regions" in Chapter 9, a memory region can be associated with some portion of either a regular file in a disk-based filesystem or a block device file. This means that an access to a byte within a page of the memory region is translated by the kernel into an operation on the corresponding byte of the file. This technique is called *memory mapping*.

Two kinds of memory mapping exist:

Shared

> Each write operation on the pages of the memory region changes the file on disk; moreover, if a process writes into a page of a shared memory mapping, the changes are visible to all other processes that map the same file.

Private

> Meant to be used when the process creates the mapping just to read the file, not to write it. For this purpose, private mapping is more efficient than shared mapping. But each write operation on a privately mapped page will cause it to stop mapping the page in the file. Thus, a write does not change the file on disk, nor is the change visible to any other processes that access the same file. However, pages of a private memory mapping that have not been modified by the process are affected by file updates performed by other processes.

A process can create a new memory mapping by issuing an `mmap( )` system call (see the section "Creating a Memory Mapping" later in this chapter). Programmers must specify either the `MAP_SHARED` flag or the `MAP_PRIVATE` flag as a parameter of the system call; as you can easily guess, in the former case the mapping is shared, while in the latter it is private. Once the mapping is created, the process can read the data stored in the file by simply reading from the memory locations of the new memory region. If the memory mapping is shared, the process can also modify the corresponding file by simply writing into the same memory locations. To destroy or shrink a memory mapping, the process may use the `munmap( )` system call (see the later section "Destroying a Memory Mapping").

As a general rule, if a memory mapping is shared, the corresponding memory region has the `VM_SHARED` flag set; if it is private, the `VM_SHARED` flag is cleared. As we'll see later, an exception to this rule exists for read-only shared memory mappings.

16.2.1. Memory Mapping Data Structures

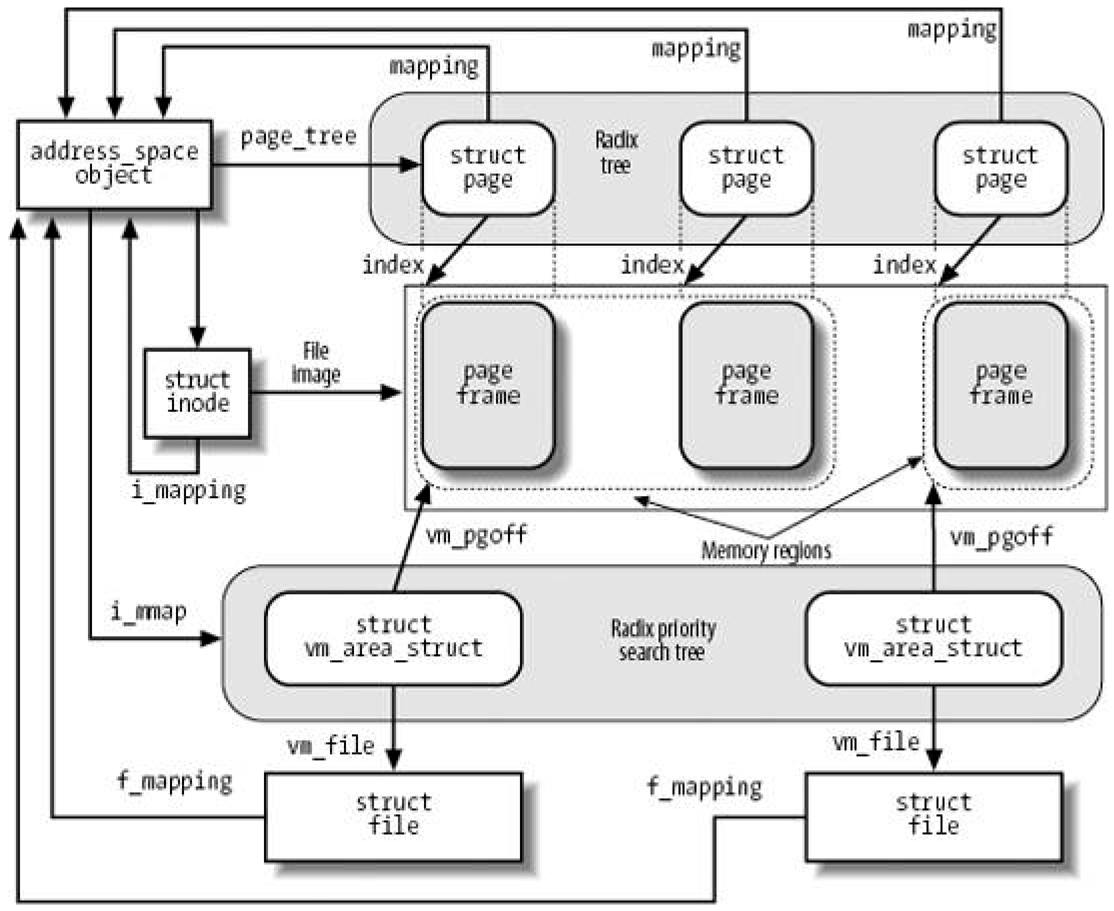
A memory mapping is represented by a combination of the following data structures :

- The inode object associated with the mapped file
- The address_space object of the mapped file
- A file object for each different mapping performed on the file by different processes
- A vm_area_struct descriptor for each different mapping on the file
- A page descriptor for each page frame assigned to a memory region that maps the file

Figure 16-2 illustrates how the data structures are linked. On the left side of the image we show the inode, which identifies the file. The i_mapping field of each inode object points to the address_space object of the file. In turn, the page_tree field of each address_space object points to the radix tree of pages belonging to the address space (see the section "The Radix Tree" in Chapter 15), while the i_mmap field points to a second tree called the radix priority search tree (PST) of memory regions belonging to the address space. The main use of PST is for performing "reverse mapping," that is, for identifying quickly all processes that share a given page. We'll cover in detail PSTs in the next chapter, because they are used for page frame reclaiming. The link between file objects relative to the same file and the inode is established by means of the f_mapping field.

Each memory region descriptor has a vm_file field that links it to the file object of the mapped file (if that field is null, the memory region is not used in a memory mapping). The position of the first mapped location is stored into the vm_pgoff field of the memory region descriptor; it represents the file offset as a number of page-size units. The length of the mapped file portion is simply the length of the memory region, which can be computed from the vm_start and vm_end fields.

Figure 16-2. Data structures for file memory mapping



Pages of shared memory mappings are always included in the page cache; pages of private memory mappings are included in the page cache as long as they are unmodified. When a process tries to modify a page of a private memory mapping, the kernel duplicates the page frame and replaces the original page frame with the duplicate in the process Page Table; this is one of the applications of the Copy On Write mechanism that we discussed in Chapter 8. The original page frame still remains in the page cache, although it no longer belongs to the memory mapping since it is replaced by the duplicate. In turn, the duplicate is not inserted into the page cache because it no longer contains valid data representing the file on disk.

Figure 16-2 also shows a few page descriptors of pages included in the page cache that refer to the memory-mapped file. Notice that the first memory region in the figure is three pages long, but only two page frames are allocated for it; presumably, the process owning the memory region has never accessed the third page.

The kernel offers several hooks to customize the memory mapping mechanism for every different filesystem. The core of memory mapping implementation is delegated to a file object's method named `mmap`. For most disk-based filesystems and for block device files, this method is implemented by a general function called `generic_file_mmap( )`, which is described in the next section.

File memory mapping depends on the demand paging mechanism described in the section "Demand Paging" in Chapter 9. In fact, a newly established memory mapping is a memory region that doesn't include any page; as the process references an address inside the region, a Page Fault occurs and the Page Fault handler checks whether the `nopage` method of the memory region is defined. If `nopage` is not defined, the memory region doesn't map a file on disk; otherwise, it does, and the method takes care of reading the page by accessing the block device. Almost all disk-based filesystems and block device files implement the `nopage` method by means of the `filemap_nopage( )` function.

16.2.2. Creating a Memory Mapping

To create a new memory mapping, a process issues an `mmap( )` system call, passing the following parameters to it:

- A file descriptor identifying the file to be mapped.
- An offset inside the file specifying the first character of the file portion to be mapped.
- The length of the file portion to be mapped.
- A set of flags. The process must explicitly set either the `MAP_SHARED` flag or the `MAP_PRIVATE` flag to specify the kind of memory mapping requested.[*]

[*] The process could also set the `MAP_ANONYMOUS` flag to specify that the new memory region is anonymous that is, not associated with any disk-based file (see the section "Demand Paging" in Chapter 9). A process can also create a memory region that is both `MAP_SHARED` and `MAP_ANONYMOUS`: in this case, the region maps a special file in the *tmpfs* filesystem (see the section "IPC Shared Memory" in Chapter 19), which can be accessed by all the process's descendants.

- A set of permissions specifying one or more types of access to the memory region: read access (`PROT_READ`), write access (`PROT_WRITE`), or execution access (`PROT_EXEC`).
- An optional linear address, which is taken by the kernel as a hint of where the new memory region should start. If the `MAP_FIXED` flag is specified and the kernel cannot allocate the new memory region starting from the specified linear address, the system call fails.

The `mmap( )` system call returns the linear address of the first location in the new memory region. For compatibility reasons, in the 80 x 86 architecture, the kernel reserves two entries in the system call table for `mmap( )` : one at index 90 and the other at index 192. The former entry corresponds to the `old_mmap( )` service routine (used by older C libraries), while the latter one corresponds to the `sys_mmap2( )` service routine (used by recent C libraries). The two service routines differ only in how the six parameters of the system call are passed. Both of them end up invoking the `do_mmap_pgoff( )` function described in the section "Allocating a Linear Address Interval" in Chapter 9. We now complete that description by detailing the steps performed only when creating a memory region that maps a file. We thus describe the case where the `file` parameter (pointer to a file object) of `do_mmap_pgoff( )` is non-null. For the sake of clarity, we refer to the enumeration used to describe `do_mmap_pgoff( )` and point out the additional steps performed under the new condition.

Step 1

Checks whether the `mmap` file operation for the file to be mapped is defined; if not, it returns an error code. A `NULL` value for `mmap` in the file operation table indicates that the corresponding file cannot be mapped (for instance, because it is a directory).

Step 2

The `get_unmapped_area( )` function invokes the `get_unmapped_area` method of the file object, if it is defined, so as to allocate an interval of linear addresses suitable for the memory mapping of the file. The disk-based filesystems do not define this method; in this case, as explained in the section "Memory Region Handling" in Chapter 9, the `get_unmapped_area( )` function ends up invoking the `get_unmapped_area` method of the memory descriptor.

Step 3

In addition to the usual consistency checks, it compares the kind of memory mapping requested (stored in the `flags` parameter of the `mmap( )` system call) and the flags specified when the file was opened (stored in the `file->f_mode` field). In particular:

- If a shared writable memory mapping is required, it checks that the file was opened for writing and that it was not opened in append mode (`O_APPEND` flag of the `open( )` system call).
- If a shared memory mapping is required, it checks that there is no mandatory lock on the file (see the section "File Locking" in Chapter 12).
- For every kind of memory mapping, it checks that the file was opened for reading.

If any of these conditions is not fulfilled, an error code is returned.

Moreover, when initializing the value of the `vm_flags` field of the new memory region descriptor, it sets the `VM_READ`, `VM_WRITE`, `VM_EXEC`, `VM_SHARED`, `VM_MAYREAD`, `VM_MAYWRITE`, `VM_MAYEXEC`, and `VM_MAYSHARE` flags according to the access rights of the file and the kind of requested memory mapping (see the section "Memory Region Access Rights" in Chapter 9). As an optimization, the `VM_SHARED` and `VM_MAYWRITE` flags are cleared for nonwritable shared memory mapping. This can be done because the process is not allowed to write into the pages of the memory region, so the mapping is treated the same as a private mapping; however, the kernel actually allows other processes that share the file to read the pages in this memory region.

Step 10

Initializes the vm_file field of the memory region descriptor with the address of the file object and increases the file's usage counter. Invokes the mmap method for the file being mapped, passing as parameters the address of the file object and the address of the memory region descriptor. For most filesystems, this method is implemented by the generic_file_mmap( ) function, which performs the following operations:

a. Stores the current time in the i_atime field of the file's inode and marks the inode as dirty.
b. Initializes the vm_ops field of the memory region descriptor with the address of the generic_file_vm_ops table. All methods in this table are null, except the nopage method, which is implemented by the filemap_nopage( ) function, and the populate method, which is implemented by the filemap_populate( ) function (see "Non-Linear Memory Mappings" later in this chapter).

Step 11

Increases the i_writecount field of the file's inode, that is, the usage counter for writing processes.

16.2.3. Destroying a Memory Mapping

When a process is ready to destroy a memory mapping, it invokes munmap( ); this system call can also be used to reduce the size of each kind of memory region. The parameters used are:

- The address of the first location in the linear address interval to be removed.
- The length of the linear address interval to be removed.

The sys_munmap( ) service routine of the system call essentially invokes the do_munmap( ) function already described in the section "Releasing a Linear Address Interval" in Chapter 9. Notice that there is no need to flush to disk the contents of the pages included in a writable shared memory mapping to be destroyed. In fact, these pages continue to act as a disk cache because they are still included in the page cache.

16.2.4. Demand Paging for Memory Mapping

For reasons of efficiency, page frames are not assigned to a memory mapping right after it has been created, but at the last possible momentthat is, when the process attempts to address one of its pages, thus causing a Page Fault exception.

We saw in the section "Page Fault Exception Handler" in Chapter 9 how the kernel verifies whether the faulty address is included in some memory region of the process; if so, the kernel checks the Page Table entry corresponding to the faulty address and invokes the do_no_page( ) function if the entry is null (see the section "Demand Paging" in Chapter 9).

The do_no_page( ) function performs all the operations that are common to all types of demand paging, such as allocating a page frame and updating the Page Tables. It also checks whether the nopage method of the memory region involved is defined. In the section "Demand Paging" in Chapter 9, we described the case in which the method is undefined (anonymous memory region); now we complete the description by discussing the main actions performed by the function when the method is defined:

1. Invokes the nopage method, which returns the address of a page frame that contains the requested page.
2. If the process is trying to write into the page and the memory mapping is private, it avoids a future Copy On Write fault by making a copy of the page just read and inserting it into the inactive list of pages (see Chapter 17). If the private memory mapping region does not already have a slave anonymous memory region that includes the new page, it either adds a new slave anonymous memory region or extends an existing one (see the section "Memory Regions" in Chapter 9). In the following steps, the function uses the new page instead of the page returned by the nopage method, so that the latter is not modified by the User Mode process.
3. If some other process has truncated or invalidated the page (the truncate_count field of the address_space descriptor is used for this kind of check), the function retries getting the page by jumping back to step 1.
4. Increases the rss field of the process memory descriptor to indicate that a new page frame has been assigned to the process.
5. Sets up the Page Table entry corresponding to the faulty address with the address of the page frame and the page access rights included in the memory region vm_page_prot field.
6. If the process is trying to write into the page, it forces the Read/Write and Dirty bits of the Page Table entry to 1. In this case, either the page frame is exclusively assigned to the process, or the page is shared; in both cases, writing to it should be allowed.

The core of the demand paging algorithm consists of the memory region's nopage method. Generally speaking, it must return the address of a page frame that contains the page accessed by the process. Its implementation depends on the kind of memory region in which the page is included.

When handling memory regions that map files on disk, the nopage method must first search for the requested page in the page cache. If the page is not found, the method must read it from disk. Most filesystems implement the nopage method by means of the filemap_nopage( ) function, which receives three parameters:

area

Descriptor address of the memory region, including the required page `address`

Linear address of the required page

`type`

Pointer to a variable in which the function writes the type of page fault detected by the function (`VM_FAULT_MAJOR` or `VM_FAULT_MINOR`)

The `filemap_nopage( )` function executes the following steps:

1. Gets the file object address `file` from the `area->vm_file` field. Derives the `address_space` object address from `file->f_mapping`. Derives the inode object address from the `host` field of the `address_space` object.
2. Uses the `vm_start` and `vm_pgoff` fields of `area` to determine the offset within the file of the data corresponding to the page starting from `address`.
3. Checks whether the file offset exceeds the file size. When this happens, it returns `NULL`, which means failure in allocating the new page, unless the Page Fault was caused by a debugger tracing another process through the `ptrace( )` system call. We are not going to discuss this special case.
4. If the `VM_RAND_READ` flag of the memory region is set (see below), we may assume that the process is reading the pages of the memory mapping in a random way. In this case, it ignores read-ahead by jumping to step 10.
5. If the `VM_SEQ_READ` flag of the memory region is set (see below), we may assume that the process is reading the pages of the memory mapping in a strictly sequential way. In this case, it invokes `page_cache_readahead( )` to perform read-ahead starting from the faulty page (see the section "Read-Ahead of Files" earlier in this chapter).
6. Invokes `find_get_page( )` to look in the page cache for the page identified by the `address_space` object and the file offset. If the page is found, it jumps to step 11.
7. If the function has reached this point, the page has not been found in the page cache. Checks the `VM_SEQ_READ` flag of the memory region:
   - If the flag is set, the kernel is aggressively reading in advance the pages of the memory region, hence the read-ahead algorithm has failed: it invokes `handle_ra_miss( )` to tune up the read-ahead parameters (see the section "Read-Ahead of Files" earlier in this chapter), then jumps to step 10.
   - Otherwise, if the flag is clear, it increases by one the `mmap_miss` counter in the `file_ra_state` descriptor of the file. If the number of misses is much larger than the number of hits (stored in the `mmap_hit` counter), it ignores read-ahead by jumping to step 10.
8. If read-ahead is not permanently disabled (`ra_pages` field in the `file_ra_state` descriptor greater than zero), it invokes `do_page_cache_readahead( )` to read a set of pages surrounding the requested page.
9. Invokes `find_get_page( )` to check whether the requested page is in the page cache; if it is there, jumps to step 11.

10. Invokes page_cache_read( ). This function checks whether the requested page is already in the page cache and, if it is not there, allocates a new page frame, adds it to the page cache, and executes the mapping->a_ops->readpage method to schedule an I/O operation that reads the page's contents from disk.
11. Invokes the grab_swap_token( ) function to possibly assign the swap token to the current process (see the section "The Swap Token" in Chapter 17).
12. The requested page is now in the page cache. Increases by one the mmap_hit counter of the file_ra_state descriptor of the file.
13. If the page is not up-to-date (PG_uptodate flag clear), it invokes lock_page( ) to lock up the page, executes the mapping->a_ops->readpage method to trigger the I/O data transfer, and invokes wait_on_page_bit( ) to sleep until the page is unlockedthat is, until the data transfer completes.
14. Invokes mark_page_accessed( ) to mark the requested page as accessed (see next chapter).
15. If an up-to-date version of the page was found in the page cache, it sets *type to VM_FAULT_MINOR; otherwise sets it to VM_FAULT_MAJOR.
16. Returns the address of the requested page.

A User Mode process can tailor the read-ahead behavior of the filemap_nopage( ) function by using the madvise( ) system call. The MADV_RANDOM command sets the VM_RAND_READ flag of the memory region to specify that the pages of the memory region will be accessed in random order; the MADV_SEQUENTIAL command sets the VM_SEQ_READ flag to specify that the pages will be accessed in strictly sequential order; finally, the MADV_NORMAL command resets both the VM_RAND_READ and VM_SEQ_READ flags to specify that the pages will be accessed in a unspecified order.

16.2.5. Flushing Dirty Memory Mapping Pages to Disk

The msync( ) system call can be used by a process to flush to disk dirty pages belonging to a shared memory mapping. It receives as its parameters the starting address of an interval of linear addresses, the length of the interval, and a set of flags that have the following meanings:

MS_SYNC

Asks the system call to suspend the process until the I/O operation completes. In this way, the calling process can assume that when the system call terminates, all pages of its memory mapping have been flushed to disk.

MS_ASYNC (complement of MS_SYNC)

Asks the system call to return immediately without suspending the calling process.

`MS_INVALIDATE`

Asks the system call to invalidate other memory mappings of the same file (not really implemented, because useless in Linux).

The `sys_msync( )` service routine invokes `msync_interval( )` on each memory region included in the interval of linear addresses. In turn, the latter function performs the following operations:

1. If the `vm_file` field of the memory region descriptor is `NULL`, or if the `VM_SHARED` flag is clear, it returns 0 (the memory region is not a writable shared memory mapping of a file).
2. Invokes the `filemap_sync( )` function, which scans the Page Table entries corresponding to the linear address intervals included in the memory region. For each page found, it resets the `Dirty` flag in the corresponding page table entry and invokes `flush_tlb_page( )` to flush the corresponding translation lookaside buffers; then, it sets the `PG_dirty` flag in the page descriptor to mark the page as dirty.
3. If the `MS_ASYNC` flag is set, it returns. Therefore, the practical effect of the `MS_ASYNC` flag consists of setting the `PG_dirty` flags of the pages in the memory region; the system call does not actually start the I/O data transfers.
4. If the function has reached this point, the `MS_SYNC` flag is set, hence the function must flush the pages in the memory region to disk and put the current process to sleep until all I/O data transfers terminate. In order to do this, the function acquires the `i_sem` semaphore of the file's inode.
5. Invokes the `filemap_fdatawrite( )` function, which receives the address of the file's `address_space` object. This function essentially sets up a `writeback_control` descriptor with the `WB_SYNC_ALL` synchronization mode, and checks whether the address space has a built-in `writepages` method. If so, it invokes the corresponding function and returns. In the opposite case, it executes the `mpage_writepages( )` function. (See the section "Writing Dirty Pages to Disk" earlier in this chapter.)
6. Checks whether the `fsync` method of the file object is defined; if so, executes it. For regular files, this method usually limits itself to flushing the inode object of the file to disk. For block device files, however, the method invokes `sync_blockdev( )`, which activates the I/O data transfer of all dirty buffers of the device.
7. Executes the `filemap_fdatawait( )` function. We recall from the section "The Tags of the Radix Tree" in Chapter 15 that a radix tree in the page cache identifies all pages that are currently being written to disk by means of the `PAGECACHE_TAG_WRITEBACK` tag. The function quickly scans the portion of the radix tree that covers the given interval of linear addresses looking for pages having the `PG_writeback` flag set; for each such page, the function invokes `wait_on_page_bit( )` to sleep until the `PG_writeback` flag is cleared that is, until the ongoing I/O data transfer on the page terminates.
8. Releases the `i_sem` semaphore of the file and returns.

16.2.6. Non-Linear Memory Mappings

The Linux 2.6 kernel offers yet another kind of access method for regular files: the *non-linear memory mappings*. Basically, a non-linear memory mapping is a file memory mapping as described previously, but its memory pages are not mapped to sequential pages on the file; rather, each memory page maps a random (arbitrary) page of file's data.

Of course, a User Mode application might achieve the same result by invoking the `mmap( )` system call repeatedly, each time on a different 4096-byte-long portion of the file. However, this approach is not very efficient for non-linear mapping of large files, because each mapping page requires its own memory region.

In order to support non-linear memory mapping, the kernel makes use of a few additional data structures. First of all, the `VM_NONLINEAR` flag of the memory region descriptor specifies that the memory region contains a non-linear mapping. All descriptors of non-linear mapping memory regions for a given file are collected in a doubly linked circular list rooted at the `i_mmap_nonlinear` field of the `address_space` object.

To create a non-linear memory mapping, the User Mode application first creates a normal shared memory mapping with the `mmap( )` system call. Then, the application remaps some of the pages in the memory mapping region by invoking `remap_file_pages( )`. The `sys_remap_file_pages( )` service routine of the system call receives four parameters:

`start`

A linear address inside a shared file memory mapping region of the calling process

`size`

Size of the remapped portion of the file in bytes

`prot`

Unused (must be zero)

`pgoff`

Page index of the initial file's page to be remapped

`flags`

Flags controlling the non-linear memory mapping

The service routine remaps the portion of the file's data identified by the `pgoff` and `size` parameters starting from the `start` linear address. If either the memory region is not shared or it is not large enough to include all the pages requested for the mapping, the system call fails and an error code is returned. Essentially, the service routine inserts the memory region in the `i_mmap_nonlinear` list of the file and invokes the `populate` method of the memory region.

For all regular files, the `populate` method is implemented by the `filemap_populate( )` function, which executes the following steps:

1. Checks whether the `MAP_NONBLOCK` flag in the flags parameter of the `remap_file_pages( )` system call is clear; if so, it invokes `do_page_cache_readahead( )` to read in advance the pages of the file to be remapped.
2. For each page to be remapped, performs the following substeps:
    a. Checks whether the page descriptor is already included in the page cache; if it is not there and the `MAP_NONBLOCK` flag is cleared, it reads the page from disk.
    b. If the page descriptor is in the page cache, it updates the Page Table entry of the corresponding linear address so that it points to the page frame, and updates the counter of pages in the memory region descriptor.
    c. Otherwise, if the page descriptor has not been found in the page cache, it stores the offset of the file's page in the 32 highest-order bits of the Page Table entry for the corresponding linear address; also, clears the `Present` bit of the Page Table entry and sets the `Dirty` bit.

As explained in the section "Demand Paging" in Chapter 9, when handling a demand-paging fault the `handle_ pte_fault( )` function checks the `Present` and `Dirty` bits in the Page Table entry; if they have the values corresponding to a non-linear memory mapping, `handle_pte_fault( )` invokes the `do_file_page( )` function, which extracts the index of the requested file's page from the high-order bits of the Page Table entry; then, `do_file_page( )` invokes the `populate` method of the memory region to read the page from disk and update the Page Table entry itself.

Because the memory pages of a non-linear memory mapping are included in the page cache according to the page index relative to the beginning of the filerather than the index relative to the beginning of the memory regionnon-linear memory mappings are flushed to disk exactly like linear memory mappings (see the section "Flushing Dirty Memory Mapping Pages to Disk" earlier in this chapter).

16.3. Direct I/O Transfers

As we have seen, in Version 2.6 of Linux, there is no substantial difference between accessing a regular file through the filesystem, accessing it by referencing its blocks on the underlying block device file, or even establishing a file memory mapping. There are, however, some highly sophisticated programs (*self-caching applications*) that would like to have full control of the whole I/O data transfer mechanism.

Consider, for example, high-performance database servers: most of them implement their own caching mechanisms that exploit the peculiar nature of the queries to the database. For these kinds of programs, the kernel page cache doesn't help; on the contrary, it is detrimental for the following reasons:

- Lots of page frames are wasted to duplicate disk data already in RAM (in the user-level disk cache).
- The `read( )` and `write( )` system calls are slowed down by the redundant instructions that handle the page cache and the read-ahead; ditto for the paging operations related to the file memory mappings.
- Rather than transferring the data directly between the disk and the user memory, the `read( )` and `write( )` system calls make two transfers: between the disk and a kernel buffer and between the kernel buffer and the user memory.

Because block hardware devices *must* be handled through interrupts and Direct Memory Access (DMA), and this can be done only in Kernel Mode, some sort of kernel support is definitely required to implement self-caching applications.

Linux offers a simple way to bypass the page cache: *direct I/O transfers*. In each I/O direct transfer, the kernel programs the disk controller to transfer the data directly from/to pages belonging to the User Mode address space of a self-caching application.

As we know, each data transfer proceeds asynchronously. While it is in progress, the kernel may switch the current process, the CPU may return to User Mode, the pages of the process that raised the data transfer might be swapped out, and so on. This works just fine for ordinary I/O data transfers because they involve pages of the disk caches . Disk caches are owned by the kernel, cannot be swapped out, and are visible to all processes in Kernel Mode.

On the other hand, direct I/O transfers should move data within pages that belong to the User Mode address space of a given process. The kernel must take care that these pages are accessible by every process in Kernel Mode and that they are not swapped out while the data transfer is in progress. Let us see how this is achieved.

When a self-caching application wishes to directly access a file, it opens the file specifying the `O_DIRECT` flag (see the section "The open( ) System Call" in Chapter 12). While servicing the `open( )` system call, the `dentry_open( )` function checks whether the `direct_IO` method is implemented for the `address_space` object of the file being opened, and returns an error code in the opposite case. The `O_DIRECT` flag can also be set for a file already opened by using the `F_SETFL` command of the `fcntl( )` system call.

Let us consider first the case where the self-caching application issues a `read( )` system call on a file with `O_DIRECT`. As mentioned in the section "Reading from a File" earlier in this chapter, the `read` file method is usually implemented by the `generic_file_read( )` function, which initializes the `iovec` and `kiocb` descriptors and invokes `_ _generic_file_aio_read( )`. The latter function verifies that the User Mode buffer described by the `iovec` descriptor is valid, then checks whether the `O_DIRECT` flag of the file is set. When invoked by a `read( )` system call, the function executes a code fragment essentially equivalent to the following:

```
if (filp->f_flags & O_DIRECT) {
    if (count == 0 || *ppos > filp->f_mapping->host->i_size)
        return 0;
    retval = generic_file_direct_IO(READ, iocb, iov, *ppos, 1);
    if (retval > 0)
        *ppos += retval;
    file_accessed(filp);
    return retval;
}
```

The function checks the current values of the file pointer, the file size, and the number of requested characters, and then invokes the `generic_file_direct_IO( )` function, passing to it the `READ` operation type, the `iocb` descriptor, the `iovec` descriptor, the current value of the file pointer, and the number of User Mode buffers specified in the `io_vec` descriptor (one). When `generic_file_direct_IO( )` terminates, `_ _generic_file_aio_read( )` updates the file pointer, sets the access timestamp on the file's inode, and returns.

Something similar happens when a `write( )` system call is issued on a file having the `O_DIRECT` flag set. As mentioned in the section "Writing to a File" earlier in this chapter, the `write` method of the file ends up invoking `generic_file_aio_write_nolock( )`: this function checks whether the `O_DIRECT` flag is set and, if so, invokes the `generic_file_direct_IO( )` function, this time specifying the `WRITE` operation type.

The `generic_file_direct_IO( )` function acts on the following parameters:

`rw`

Type of operation: `READ` or `WRITE`

`iocb`

Pointer to a `kiocb` descriptor (see Table 16-1)

`iov`

Pointer to an array of `iovec` descriptors (see the section "Reading from a File" earlier in this chapter) `offset`

File offset

`nr_segs`

Number of `iovec` descriptors in the `iov` array

The steps performed by `generic_file_direct_IO( )` are the following:

1. Gets the address `file` of the file object from the `ki_filp` field of the `kiocb` descriptor, and the address `mapping` of the `address_space` object from the `file->f_mapping` field.
2. If the type of operation is `WRITE` and if one or more processes have created a memory mapping associated with a portion of the file, it invokes `unmap_mapping_range( )` to unmap all pages of the file. This function also ensures that if any Page Table entry corresponding to a page to be unmapped has the `Dirty` bit set, then the corresponding page is marked as dirty in the page cache.
3. If the radix tree rooted at `mapping` is not empty (`mapping->nrpages` greater than zero), it invokes the `filemap_fdatawrite( )` and `filemap_fdatawait( )` functions to flush all dirty pages to disk and to wait until the I/O operations complete (see the section "Flushing Dirty Memory Mapping Pages to Disk" earlier in this chapter). (Even if the self-caching application is accessing the file directly, there could be other applications in the system that access the file through the page cache. To avoid data loss, the disk image is synchronized with the page cache before starting the direct I/O transfer.)
4. Invokes the `direct_IO` method of the `mapping` address space (see the following paragraphs).
5. If the operation type was `WRITE`, it invokes `invalidate_inode_pages2( )` to scan all pages in the radix tree of `mapping` and to release them. The function also clears the User Mode Page Table entries that refer to those pages.

In most cases, the `direct_IO` method is a wrapper for the `_ _blockdev_direct_IO( )` function. This function is quite complex and invokes a large number of auxiliary data structures and functions; however, it executes essentially the same kind of operations already described in this chapter: it splits the data to be read or written in suitable blocks, locates the data on disk, and fills up one or more bio descriptors that describe the I/O operations to be performed. Of course, the data will be read or written directly in the User Mode buffers specified by the `iovec` descriptors in the `iov` array. The bio descriptors are submitted to the generic block layer by invoking the `submit_bio( )` function (see the section "Submitting Buffer Heads to the Generic Block Layer" in Chapter 15). Usually, the `_ _blockdev_direct_IO( )` function does not return until all direct I/O transfers have been completed; thus, once the `read( )` or `write( )` system call returns, the self-caching application can safely access the buffers containing the file data.

16.4. Asynchronous I/O

The POSIX 1003.1 standard defines a set of library functionslisted in Table 16-4for accessing the files in an asynchronous way. "Asynchronous" essentially means that when a User Mode process invokes a library function to read or write a file, the function terminates as soon as the read or write operation has been enqueued, possibly even before the actual I/O data transfer takes place. The calling process can thus continue its execution while the data is being transferred.

Table 16-4. The POSIX library functions for asynchronous I/O

Function	Description
`aio_read( )`	Asynchronously reads some data from a file
`aio_write( )`	Asynchronously writes some data into a file
`aio_fsync( )`	Requests a flush operation for all outstanding asynchronous I/O operations (does not block)
`aio_error( )`	Gets the error code for an outstanding asynchronous I/O operation
`aio_return( )`	Gets the return code for a completed asynchronous I/O operation
`aio_cancel( )`	Cancels an outstanding asynchronous I/O operation
`aio_suspend( )`	Suspends the process until at least one of several outstanding I/O operations completes

Using asynchronous I/O is quite simple. The application opens the file by means of the usual `open( )` system call. Then, it fills up a control block of type `struct aiocb` with the information describing the requested operation. The most commonly used fields of the `struct aiocb` control block are:

`aio_fildes`

The file descriptor of the file (as returned by the `open( )` system call)

`aio_buf`

The User Mode buffer for the file's data

`aio_nbytes`

How many bytes should be transferred

`aio_offset`

> Position in the file where the read or write operation will start (it is independent of the "synchronous" file pointer)

Finally, the application passes the address of the control block to either `aio_read( )` or `aio_write( )` ; both functions terminate as soon as the requested I/O data transfer has been enqueued by the system library or kernel. The application can later check the status of the outstanding I/O operation by invoking `aio_error( )`, which returns `EINPROGRESS` if the data transfer is still in progress, 0 if it is successfully completed, or an error code in case of failure. The `aio_return( )` function returns the number of bytes effectively read or written by a completed asynchronous I/O operation, or `-1` in case of failure.

16.4.1. Asynchronous I/O in Linux 2.6

Asynchronous I/O can be implemented by a system library without any kernel support at all. Essentially, the `aio_read( )` or `aio_write( )` library function clones the current process and lets the child invoke the synchronous `read( )` or `write( )` system calls; then, the parent terminates the `aio_read( )` or `aio_write( )` function and continues the execution of the program, hence it does not wait for the synchronous operation started by the child to finish. However, this "poor man's" version of the POSIX functions is significantly slower than a version that uses a kernel-level implementation of asynchronous I/O.

The Linux 2.6 kernel version sports a set of system calls for asynchronous I/O. However, in Linux 2.6.11 this feature is a work in progress, and asyncronous I/O works properly only for files opened with the `O_DIRECT` flag set (see the previous section). The system calls for asynchronous I/O are listed in Table 16-5.

Table 16-5. Linux system calls for asynchronous I/O

System call	Description
`io_setup( )`	Initializes an asynchronous context for the current process
`io_submit( )`	Submits one or more asynchronous I/O operations
`io_getevents( )`	Gets the completion status of some outstanding asynchronous I/O operations
`io_cancel( )`	Cancels an outstanding I/O operation
`io_destroy( )`	Removes an asynchronous context for the current process

16.4.1.1. The asynchronous I/O context

If a User Mode process wants to make use of the `io_submit( )` system call to start an asynchronous I/O operation, it must create beforehand an *asynchronous I/O context*.

Basically, an asynchronous I/O context (in short, AIO context) is a set of data structures that keep track of the on-going progresses of the asynchronous I/O operations requested by the process. Each AIO context is associated with a `kioctx` object, which stores all information relevant for the context. An application might create several AIO contexts; all `kioctx` descriptors of a given process are collected in a singly linked list rooted at the `ioctx_list` field of the memory descriptor (see Table 9-2 in Chapter 9).

We are not going to discuss in detail the `kioctx` object; however, we should pinpoint an important data structure referenced by the `kioctx` object: the AIO ring.

The *AIO ring* is a memory buffer in the address space of the User Mode process that is also accessible by all processes in Kernel Mode. The User Mode starting address and length of the AIO ring are stored in the `ring_info.mmap_base` and `ring_info.mmap_size` fields of the `kioctx` object, respectively. The descriptors of all page frames composing the AIO ring are stored in an array pointed to by the `ring_info.ring_pages` field.

The AIO ring is essentially a circular buffer where the kernel writes the completion reports of the outstanding asynchronous I/O operations. The first bytes of the AIO ring contain an header (a `struct aio_ring` data structure); the remaining bytes store `io_event` data structures, each of which describes a completed asynchronous I/O operation. Because the pages of the AIO ring are mapped in the User Mode address space of the process, the application can check directly the progress of the outstanding asynchronous I/O operations, thus avoiding using a relatively slow system call.

The `io_setup( )` system call creates a new AIO context for the calling process. It expects two parameters: the maximum number of outstanding asynchronous I/O operations, which ultimately determines the size of the AIO ring, and a pointer to a variable that will store a handle to the context; this handle is also the base address of the AIO ring. The `sys_io_setup( )` service routine essentially invokes `do_mmap( )` to allocate a new anonymous memory region for the process that will contain the AIO ring (see the section "Allocating a Linear Address Interval" in Chapter 9), and creates and initializes a `kioctx` object describing the AIO context.

Conversely, the `io_destroy( )` system call removes an AIO context; it also destroys the anonymous memory region containing the corresponding AIO ring. The system call blocks the current process until all outstanding asynchronous I/O operations are complete.

16.4.1.2. Submitting the asynchronous I/O operations

To start some asynchronous I/O operations, the application invokes the `io_submit( )` system call. The system call has three parameters:

`ctx_id`

The handle returned by io_setup( ), which identifies the AIO context

iocbpp

The address of an array of pointers to descriptors of type iocb, each of which describes one asynchronous I/O operation

nr

The length of the array pointed to by iocbpp

The iocb data structure includes the same fields as the POSIX aiocb descriptor (aio_fildes, aio_buf, aio_nbytes, aio_offset) plus the aio_lio_opcode field that stores the type of the requested operation (typically read, write, or sync).

The service routine sys_io_submit( ) performs essentially the following steps:

1. Verifies that the array of iocb descriptors is valid.
2. Searches the kioctx object corresponding to the ctx_id handle in the list rooted at the ioctx_list field of the memory descriptor.
3. For each iocb descriptor in the array, it executes the following substeps:
    a. Gets the address of the file object corresponding to the file descriptor stored in the aio_fildes field.
    b. Allocates and initializes a new kiocb descriptor for the I/O operation.
    c. Checks that there is a free slot in the AIO ring to store the completion result of the operation.
    d. Sets the ki_retry method of the kiocb descriptor according to the type of the operation (see below).
    e. Executes the aio_run_iocb( ) function, which essentially invokes the ki_retry method to start the I/O data transfer for the corresponding asynchronous I/O operation. If the ki_retry method returns the value -EIOCBRETRY, the asynchronous I/O operation has been submitted but not yet fully satisfied: the aio_run_iocb( ) function will be invoked again on this kiocb at a later time (see below). Otherwise, it invokes aio_complete( ) to add a completion event for the asynchronous I/O operation in the ring of the AIO context.

If the asynchronous I/O operation is a read request, the ki_retry method of the corresponding kiocb descriptor is implemented by aio_pread( ). This function essentially executes the aio_read method of the file object, then updates the ki_buf and ki_left fields of the kiocb descriptor (see Table 16-1 earlier in this chapter) according to the value returned by the aio_read method. Finally, aio_pread( ) returns the number of bytes effectively read from the file, or the value -EIOCBRETRY if the function determines that not all requested bytes have been transferred. For most filesystems, the aio_read method of the file object ends up invoking the _ _generic_file_aio_read( ) function. Assuming that the O_DIRECT flag of the file is set, this function ends up invoking the `generic_file_direct_IO( )` function, as described in the previous section. In this case, however, the `_ _blockdev_direct_IO( )` function does not block the current process waiting for the I/O data transfer to complete; instead, the function returns immediately. Because the asynchronous I/O operation is still outstanding, the `aio_run_iocb( )` will be invoked again, this time by the *aio* kernel thread of the `aio_wq` work queue. The `kiocb` descriptor keeps track of the progress of the I/O data transfer; eventually all requested data will be transferred and the completion result will be added to the AIO ring.

Similarly, if the asynchronous I/O operation is a write request, the `ki_retry` method of the `kiocb` descriptor is implemented by `aio_pwrite( )`. This function essentially executes the `aio_write` method of the file object, then updates the `ki_buf` and `ki_left` fields of the `kiocb` descriptor (see Table 16-1 earlier in this chapter) according to the value returned by the `aio_write` method. Finally, `aio_pwrite( )` returns the number of bytes effectively written to the file, or the value `-EIOCBRETRY` if the function determines that not all requested bytes have been transferred. For most filesystems, the `aio_write` method of the file object ends up invoking the `generic_file_aio_write_nolock( )` function. Assuming that the `O_DIRECT` flag of the file is set, this function ends up invoking the `generic_file_direct_IO( )` function, as above.

Chapter 18. The Ext2 and Ext3 Filesystems

In this chapter, we finish our extensive discussion of I/O and filesystems by taking a look at the details the kernel has to take care of when interacting with a specific filesystem. Because the Second Extended Filesystem (Ext2) is native to Linux and is used on virtually every Linux system, it is a natural choice for this discussion. Furthermore, Ext2 illustrates a lot of good practices in its support for modern filesystem features with fast performance. To be sure, other filesystems supported by Linux include many interesting features, but we have no room to examine all of them.

After introducing Ext2 in the section "General Characteristics of Ext2," we describe the data structures needed, just as in other chapters. Because we are looking at a specific way to store data on disk, we have to consider two versions of the same data structures. The section "Ext2 Disk Data Structures" shows the data structures stored by Ext2 on disk, while "Ext2 Memory Data Structures" shows the corresponding versions in memory.

Then we get to the operations performed on the filesystem. In the section "Creating the Ext2 Filesystem," we discuss how Ext2 is created in a disk partition. The next sections describe the kernel activities performed whenever the disk is used. Most of these are relatively low-level activities dealing with the allocation of disk space to inodes and data blocks.

In the last section, we give a short description of the Ext3 filesystem, which is the next step in the evolution of the Ext2 filesystem .

18.1. General Characteristics of Ext2

Unix-like operating systems use several types of filesystems. Although the files of all such filesystems have a common subset of attributes required by a few POSIX APIs such as `stat( )`, each filesystem is implemented in a different way.

The first versions of Linux were based on the MINIX filesystem. As Linux matured, the *Extended Filesystem (Ext FS)* was introduced; it included several significant extensions, but offered unsatisfactory performance. The *Second Extended Filesystem (Ext2)* was introduced in 1994; besides including several new features , it is quite efficient and robust and is, together with its offspring Ext3, the most widely used Linux filesystem.

The following features contribute to the efficiency of Ext2:

- When creating an Ext2 filesystem, the system administrator may choose the optimal block size (from 1,024 to 4,096 bytes), depending on the expected average file length. For instance, a 1,024-block size is preferable when the average file length is smaller than a few thousand bytes because this leads to less internal fragmentationthat is, less of a mismatch between the file length and the portion of the disk that stores it (see the section "Memory Area Management" in Chapter 8, where internal fragmentation for dynamic memory was discussed). On the other hand, larger block sizes are usually preferable for files greater than a few thousand bytes because this leads to fewer disk transfers, thus reducing system overhead.
- When creating an Ext2 filesystem, the system administrator may choose how many inodes to allow for a partition of a given size, depending on the expected number of files to be stored on it. This maximizes the effectively usable disk space.
- The filesystem partitions disk blocks into groups. Each group includes data blocks and inodes stored in adjacent tracks. Thanks to this structure, files stored in a single block group can be accessed with a lower average disk seek time.
- The filesystem *preallocates* disk data blocks to regular files before they are actually used. Thus, when the file increases in size, several blocks are already reserved at physically adjacent positions, reducing file fragmentation.
- Fast symbolic links (see the section "Hard and Soft Links" in Chapter 1) are supported. If the symbolic link represents a short pathname (at most 60 characters), it can be stored in the inode and can thus be translated without reading a data block.

Moreover, the Second Extended Filesystem includes other features that make it both robust and flexible:

- A careful implementation of file-updating that minimizes the impact of system crashes. For instance, when creating a new hard link for a file, the counter of hard links in the disk inode is increased first, and the new name is added into the proper directory next. In this way, if a hardware failure occurs after the inode update but before the directory can be changed, the directory is consistent, even if the inode's hard link counter is wrong. Deleting the file does not lead to catastrophic results, although the file's data blocks cannot be automatically reclaimed. If the reverse were done (changing the directory before updating the inode), the same hardware failure would produce a dangerous inconsistency: deleting the original hard link would remove its data blocks from disk, yet the new directory entry would refer to an inode that no longer exists. If that inode number were used later for another file, writing into the stale directory entry would corrupt the new file.

- Support for automatic consistency checks on the filesystem status at boot time. The checks are performed by the *e2fsck* external program, which may be activated not only after a system crash, but also after a predefined number of filesystem mounts (a counter is increased after each mount operation) or after a predefined amount of time has elapsed since the most recent check.
- Support for *immutable* files (they cannot be modified, deleted, or renamed) and for *append-only* files (data can be added only to the end of them).
- Compatibility with both the Unix System V Release 4 and the BSD semantics of the user group ID for a new file. In SVR4, the new file assumes the user group ID of the process that creates it; in BSD, the new file inherits the user group ID of the directory containing it. Ext2 includes a mount option that specifies which semantic to use.

Even if the Ext2 filesystem is a mature, stable program, several additional features have been considered for inclusion. Some of them have already been coded and are available as external patches. Others are just planned, but in some cases, fields have already been introduced in the Ext2 inode for them. The most significant features being considered are:

Block fragmentation

System administrators usually choose large block sizes for accessing disks, because computer applications often deal with large files. As a result, small files stored in large blocks waste a lot of disk space. This problem can be solved by allowing several files to be stored in different fragments of the same block.

Handling of transparently compressed and encrypted files

These new options, which must be specified when creating a file, allow users to transparently store compressed and/or encrypted versions of their files on disk.

Logical deletion

An *undelete* option allows users to easily recover, if needed, the contents of a previously removed file.

Journaling

> Journaling avoids the time-consuming check that is automatically performed on a filesystem when it is abruptly unmounted for instance, as a consequence of a system crash.

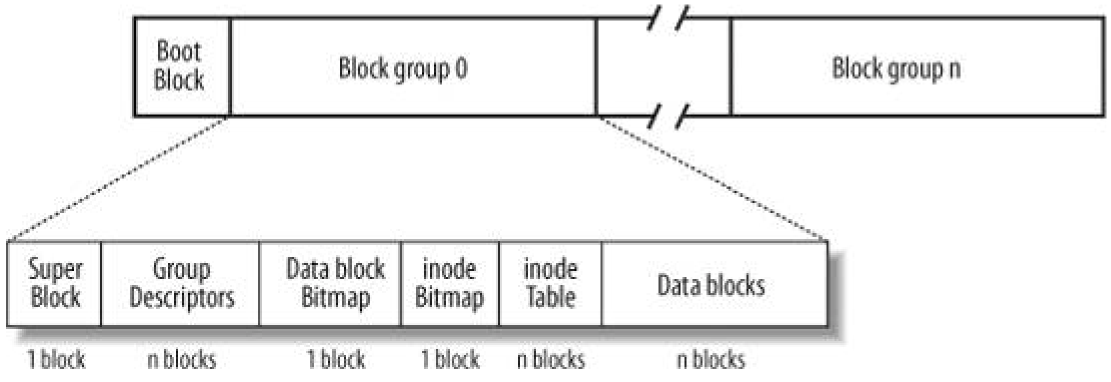
In practice, none of these features has been officially included in the Ext2 filesystem. One might say that Ext2 is victim of its success; it has been the preferred filesystem adopted by most Linux distribution companies until a few years ago, and the millions of users who relied on it every day would have looked suspiciously at any attempt to replace Ext2 with some other filesystem.

The most compelling feature missing from Ext2 is journaling, which is required by high-availability servers. To provide for a smooth transition, journaling has not been introduced in the Ext2 filesystem; rather, as we'll discuss in the later section "The Ext3 Filesystem," a more recent filesystem that is fully compatible with Ext2 has been created, which also offers journaling. Users who do not really require journaling may continue to use the good old Ext2 filesystem, while the others will likely adopt the new filesystem. Nowadays, most distributions adopt Ext3 as the standard filesystem.

18.2. Ext2 Disk Data Structures

The first block in each Ext2 partition is never managed by the Ext2 filesystem, because it is reserved for the partition boot sector (see Appendix A). The rest of the Ext2 partition is split into *block groups* , each of which has the layout shown in Figure 18-1. As you will notice from the figure, some data structures must fit in exactly one block, while others may require more than one block. All the block groups in the filesystem have the same size and are stored sequentially, thus the kernel can derive the location of a block group in a disk simply from its integer index.

Figure 18-1. Layouts of an Ext2 partition and of an Ext2 block group



Block groups reduce file fragmentation, because the kernel tries to keep the data blocks belonging to a file in the same block group, if possible. Each block in a block group contains one of the following pieces of information:

- A copy of the filesystem's superblock
- A copy of the group of block group descriptors
- A data block bitmap
- An inode bitmap
- A table of inodes
- A chunk of data that belongs to a file; i.e., data blocks

If a block does not contain any meaningful information, it is said to be free.

As you can see from Figure 18-1, both the superblock and the group descriptors are duplicated in each block group. Only the superblock and the group descriptors included in block group 0 are used by the kernel, while the remaining superblocks and group descriptors are left unchanged; in fact, the kernel doesn't even look at them. When the *e2fsck* program executes a consistency check on the filesystem status, it refers to the superblock and the group descriptors stored in block group 0, and then copies them into all other block groups. If data corruption occurs and the main superblock or the main group descriptors in block group 0 become invalid, the system administrator can instruct *e2fsck* to refer to the old copies of the superblock and the group descriptors stored in a block groups other than the first. Usually, the redundant copies store enough information to allow *e2fsck* to bring the Ext2 partition back to a consistent state.

How many block groups are there? Well, that depends both on the partition size and the block size. The main constraint is that the block bitmap, which is used to identify the blocks that are used and free inside a group, must be stored in a single block. Therefore, in each block group, there can be at most 8x*b* blocks, where *b* is the block size in bytes. Thus, the total number of block groups is roughly *s*/(8x*b*), where *s* is the partition size in blocks.

For example, let's consider a 32-GB Ext2 partition with a 4-KB block size. In this case, each 4-KB block bitmap describes 32K data blocks that is, 128 MB. Therefore, at most 256 block groups are needed. Clearly, the smaller the block size, the larger the number of block groups.

18.2.1. Superblock

An Ext2 disk superblock is stored in an `ext2_super_block` structure, whose fields are listed in Table 18-1.[*] The `_ _u8`, `_ _u16`, and `_ _u32` data types denote unsigned numbers of length 8, 16, and 32 bits respectively, while the `_ _s8`, `_ _s16`, `_ _s32` data types denote signed numbers of length 8, 16, and 32 bits. To explicitly specify the order in which the bytes of a word or double-word are stored on disk, the kernel also makes use of the `_ _le16`, `_ _le32`, `_ _be16`, and `_ _be32` data types; the former two types denote the *little-endian ordering* for words and double-words (the least significant byte is stored at the highest address), respectively, while the latter two types denote the *big-endian ordering* (the most significant byte is stored at the highest address).

[*] To ensure compatibility between the Ext2 and Ext3 filesystems, the `ext2_super_block` data structure includes some Ext3-specific fields, which are not shown in Table 18-1.

Table 18-1. The fields of the Ext2 superblock

Type	Field	Description
`_ _le32`	`s_inodes_count`	Total number of inodes
`_ _le32`	`s_blocks_count`	Filesystem size in blocks
`_ _le32`	`s_r_blocks_count`	Number of reserved blocks
`_ _le32`	`s_free_blocks_count`	Free blocks counter
`_ _le32`	`s_free_inodes_count`	Free inodes counter
`_ _le32`	`s_first_data_block`	Number of first useful block (always 1)
`_ _le32`	`s_log_block_size`	Block size
`_ _le32`	`s_log_frag_size`	Fragment size
`_ _le32`	`s_blocks_per_group`	Number of blocks per group
`_ _le32`	`s_frags_per_group`	Number of fragments per group
`_ _le32`	`s_inodes_per_group`	Number of inodes per group
`_ _le32`	`s_mtime`	Time of last mount operation
`_ _le32`	`s_wtime`	Time of last write operation
`_ _le16`	`s_mnt_count`	Mount operations counter
`_ _le16`	`s_max_mnt_count`	Number of mount operations before check
`_ _le16`	`s_magic`	Magic signature
`_ _le16`	`s_state`	Status flag
`_ _le16`	`s_errors`	Behavior when detecting errors
`_ _le16`	`s_minor_rev_level`	Minor revision level
`_ _le32`	`s_lastcheck`	Time of last check
`_ _le32`	`s_checkinterval`	Time between checks
`_ _le32`	`s_creator_os`	OS where filesystem was created
`_ _le32`	`s_rev_level`	Revision level of the filesystem
`_ _le16`	`s_def_resuid`	Default UID for reserved blocks
`_ _le16`	`s_def_resgid`	Default user group ID for reserved blocks
`_ _le32`	`s_first_ino`	Number of first nonreserved inode
`_ _le16`	`s_inode_size`	Size of on-disk inode structure
`_ _le16`	`s_block_group_nr`	Block group number of this superblock
`_ _le32`	`s_feature_compat`	Compatible features bitmap
`_ _le32`	`s_feature_incompat`	Incompatible features bitmap

Table 18-1. The fields of the Ext2 superblock

Type	Field	Description
_ _le32	s_feature_ro_compat	Read-only compatible features bitmap
_ _u8 [16]	s_uuid	128-bit filesystem identifier
char [16]	s_volume_name	Volume name
char [64]	s_last_mounted	Pathname of last mount point
_ _le32	s_algorithm_usage_bitmap	Used for compression
_ _u8	s_prealloc_blocks	Number of blocks to preallocate
_ _u8	s_prealloc_dir_blocks	Number of blocks to preallocate for directories
_ _u16	s_padding1	Alignment to word
_ _u32 [204]	s_reserved	Nulls to pad out 1,024 bytes

The s_inodes_count field stores the number of inodes, while the s_blocks_count field stores the number of blocks in the Ext2 filesystem.

The s_log_block_size field expresses the block size as a power of 2, using 1,024 bytes as the unit. Thus, 0 denotes 1,024-byte blocks, 1 denotes 2,048-byte blocks, and so on. The s_log_frag_size field is currently equal to s_log_block_size, because block fragmentation is not yet implemented.

The s_blocks_per_group, s_frags_per_group, and s_inodes_per_group fields store the number of blocks, fragments, and inodes in each block group, respectively.

Some disk blocks are reserved to the superuser (or to some other user or group of users selected by the s_def_resuid and s_def_resgid fields). These blocks allow the system administrator to continue to use the filesystem even when no more free blocks are available for normal users.

The s_mnt_count, s_max_mnt_count, s_lastcheck, and s_checkinterval fields set up the Ext2 filesystem to be checked automatically at boot time. These fields cause *e2fsck* to run after a predefined number of mount operations has been performed, or when a predefined amount of time has elapsed since the last consistency check. (Both kinds of checks can be used together.) The consistency check is also enforced at boot time if the filesystem has not been cleanly unmounted (for instance, after a system crash) or when the kernel discovers some errors in it. The s_state field stores the value 0 if the filesystem is mounted or was not cleanly unmounted, 1 if it was cleanly unmounted, and 2 if it contains errors.

18.2.2. Group Descriptor and Bitmap

Each block group has its own group descriptor, an `ext2_group_desc` structure whose fields are illustrated in Table 18-2.

Table 18-2. The fields of the Ext2 group descriptor

Type	Field	Description
`_ _le32`	`bg_block_bitmap`	Block number of block bitmap
`_ _le32`	`bg_inode_bitmap`	Block number of inode bitmap
`_ _le32`	`bg_inode_table`	Block number of first inode table block
`_ _le16`	`bg_free_blocks_count`	Number of free blocks in the group
`_ _le16`	`bg_free_inodes_count`	Number of free inodes in the group
`_ _le16`	`bg_used_dirs_count`	Number of directories in the group
`_ _le16`	`bg_pad`	Alignment to word
`_ _le32 [3]`	`bg_reserved`	Nulls to pad out 24 bytes

The `bg_free_blocks_count`, `bg_free_inodes_count`, and `bg_used_dirs_count` fields are used when allocating new inodes and data blocks. These fields determine the most suitable block in which to allocate each data structure. The bitmaps are sequences of bits, where the value 0 specifies that the corresponding inode or data block is free and the value 1 specifies that it is used. Because each bitmap must be stored inside a single block and because the block size can be 1,024, 2,048, or 4,096 bytes, a single bitmap describes the state of 8,192, 16,384, or 32,768 blocks.

18.2.3. Inode Table

The inode table consists of a series of consecutive blocks, each of which contains a predefined number of inodes. The block number of the first block of the inode table is stored in the `bg_inode_table` field of the group descriptor.

All inodes have the same size: 128 bytes. A 1,024-byte block contains 8 inodes, while a 4,096-byte block contains 32 inodes. To figure out how many blocks are occupied by the inode table, divide the total number of inodes in a group (stored in the `s_inodes_per_group` field of the superblock) by the number of inodes per block.

Each Ext2 inode is an `ext2_inode` structure whose fields are illustrated in Table 18-3.

Table 18-3. The fields of an Ext2 disk inode

Type	Field	Description
`_ _le16`	`i_mode`	File type and access rights

Table 18-3. The fields of an Ext2 disk inode

Type	Field	Description
`_ _le16`	`i_uid`	Owner identifier
`_ _le32`	`i_size`	File length in bytes
`_ _le32`	`i_atime`	Time of last file access
`_ _le32`	`i_ctime`	Time that inode last changed
`_ _le32`	`i_mtime`	Time that file contents last changed
`_ _le32`	`i_dtime`	Time of file deletion
`_ _le16`	`i_gid`	User group identifier
`_ _le16`	`i_links_count`	Hard links counter
`_ _le32`	`i_blocks`	Number of data blocks of the file
`_ _le32`	`i_flags`	File flags
`union`	`osd1`	Specific operating system information
`_ _le32 [EXT2_N_BLOCKS]`	`i_block`	Pointers to data blocks
`_ _le32`	`i_generation`	File version (used when the file is accessed by a network filesystem)
`_ _le32`	`i_file_acl`	File access control list
`_ _le32`	`i_dir_acl`	Directory access control list
`_ _le32`	`i_faddr`	Fragment address
`union`	`osd2`	Specific operating system information

Many fields related to POSIX specifications are similar to the corresponding fields of the VFS's inode object and have already been discussed in the section "Inode Objects" in Chapter 12. The remaining ones refer to the Ext2-specific implementation and deal mostly with block allocation.

In particular, the `i_size` field stores the effective length of the file in bytes, while the `i_blocks` field stores the number of data blocks (in units of 512 bytes) that have been allocated to the file.

The values of `i_size` and `i_blocks` are not necessarily related. Because a file is always stored in an integer number of blocks, a nonempty file receives at least one data block (since fragmentation is not yet implemented) and `i_size` may be smaller than 512 x`i_blocks`. On the other hand, as we'll see in the section "File Holes" later in this chapter, a file may contain holes. In that case, `i_size` may be greater than 512 x`i_blocks`.

The `i_block` field is an array of `EXT2_N_BLOCKS` (usually 15) pointers to blocks used to identify the data blocks allocated to the file (see the section "Data Blocks Addressing" later in this chapter).

The 32 bits reserved for the `i_size` field limit the file size to 4 GB. Actually, the highest-order bit of the `i_size` field is not used, so the maximum file size is limited to 2 GB. However, the Ext2 filesystem includes a "dirty trick" that allows larger files on systems that sport a 64-bit processor such as AMD's Opteron or IBM's PowerPC G5. Essentially, the `i_dir_acl` field of the inode, which is not used for regular files, represents a 32-bit extension of the `i_size` field. Therefore, the file size is stored in the inode as a 64-bit integer. The 64-bit version of the Ext2 filesystem is somewhat compatible with the 32-bit version because an Ext2 filesystem created on a 64-bit architecture may be mounted on a 32-bit architecture, and vice versa. On a 32-bit architecture, a large file cannot be accessed, unless opening the file with the `O_LARGEFILE` flag set (see the section "The open( ) System Call" in Chapter 12).

Recall that the VFS model requires each file to have a different inode number. In Ext2, there is no need to store on disk a mapping between an inode number and the corresponding block number because the latter value can be derived from the block group number and the relative position inside the inode table. For example, suppose that each block group contains 4,096 inodes and that we want to know the address on disk of inode 13,021. In this case, the inode belongs to the third block group and its disk address is stored in the 733rd entry of the corresponding inode table. As you can see, the inode number is just a key used by the Ext2 routines to retrieve the proper inode descriptor on disk quickly.

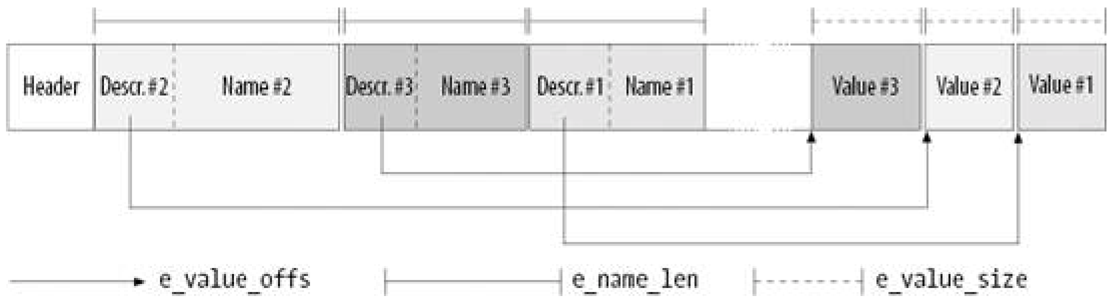
18.2.4. Extended Attributes of an Inode

The Ext2 inode format is a kind of straitjacket for filesystem designers. The length of an inode must be a power of 2 to avoid internal fragmentation in the blocks that store the inode table. Actually, most of the 128 characters of an Ext2 inode are currently packed with information, and there is little room left for additional fields. On the other hand, expanding the inode length to 256 would be quite wasteful, besides introducing compatibility problems between Ext2 filesystems that use different inode lengths.

Extended attributes have been introduced to overcome the above limitation. These attributes are stored on a disk block allocated outside of any inode. The `i_file_acl` field of an inode points to the block containing the extended attributes . Different inodes that have the same set of extended attributes may share the same block.

Each extended attribute has a name and a value. Both of them are encoded as variable length arrays of characters, as specified by the `ext2_xattr_entry` descriptor. Figure 18-2 shows the layout in Ext2 of the extended attributes inside a block. Each attribute is split in two parts: the `ext2_xattr_entry` descriptor together with the name of the attribute are placed at the beginning of the block, while the value of the attribute is placed at the end of the block. The entries at the beginning of the block are ordered according to the attribute names, while the positions of the values are fixed, because they are determined by the allocation order of the attributes.

Figure 18-2. Layout of a block containing extended attributes



There are many system calls used to set, retrieve, list, and remove the extended attributes of a file. The `setxattr( )` , `lsetxattr( )` , and `fsetxattr( )` system calls set an extended attribute of a file; essentially, they differ in how symbolic links are handled, and in how the file is specified (either passing a pathname or a file descriptor). Similarly, the `getxattr( )` , `lgetxattr( )` , and `fgetxattr( )` system calls return the value of an extended attribute. The `listxattr( )`, `llistxattr( )` , and `flistxattr( )` list all extended attributes of a file. Finally, the `removexattr( )` , `lremovexattr( )` , and `fremovexattr( )` system calls remove an extended attribute from a file.

18.2.5. Access Control Lists

Access control lists were proposed a long time ago to improve the file protection mechanism in Unix filesystems. Instead of classifying the users of a file under three classesowner, group, and othersan *access control list* (*ACL*) can be associated with each file. Thanks to this kind of list, a user may specify for each of his files the names of specific users (or groups of users) and the privileges to be given to these users.

Linux 2.6 fully supports ACLs by making use of inode extended attributes. As a matter of fact, extended attributes have been introduced mainly to support ACLs. Therefore, the `chacl( )` , `setfacl( )` , and `getfacl( )` library functions, which allow you to manipulate the ACLs of a file, rely essentially upon the `setxattr( )` and `getxattr( )` system calls introduced in the previous section.

Unfortunately, the outcome of a working group that defined security extensions within the POSIX 1003.1 family of standards has never been formalized as a new POSIX standard. As a result, ACLs are supported nowadays on different filesystem types on many UNIX-like systems, albeit with a number of subtle differences among the different implementations.

18.2.6. How Various File Types Use Disk Blocks

The different types of files recognized by Ext2 (regular files, pipes, etc.) use data blocks in different ways. Some files store no data and therefore need no data blocks at all. This section discusses the storage requirements for each type, which are listed in Table 18-4.

Table 18-4. Ext2 file types

File_type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

18.2.6.1. Regular file

Regular files are the most common case and receive almost all the attention in this chapter. But a regular file needs data blocks only when it starts to have data. When first created, a regular file is empty and needs no data blocks; it can also be emptied by the `truncate( )` or `open( )` system calls. Both situations are common; for instance, when you issue a shell command that includes the string >*filename*, the shell creates an empty file or truncates an existing one.

18.2.6.2. Directory

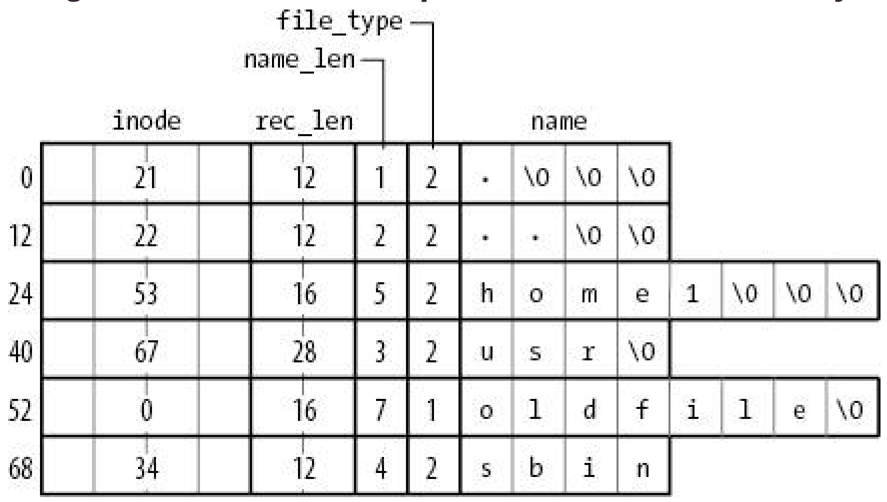
Ext2 implements directories as a special kind of file whose data blocks store filenames together with the corresponding inode numbers. In particular, such data blocks contain structures of type `ext2_dir_entry_2`. The fields of that structure are shown in Table 18-5. The structure has a variable length, because the last `name` field is a variable length array of up to `EXT2_NAME_LEN` characters (usually 255). Moreover, for reasons of efficiency, the length of a directory entry is always a multiple of 4 and, therefore, null characters (`\0`) are added for padding at the end of the filename, if necessary. The `name_len` field stores the actual filename length (see Figure 18-3).

Table 18-5. The fields of an Ext2 directory entry

Type	Field	Description
`_ _le32`	`inode`	Inode number
`_ _le16`	`rec_len`	Directory entry length
`_ _u8`	`name_len`	Filename length
`_ _u8`	`file_type`	File type
`char [EXT2_NAME_LEN]`	`name`	Filename

The file_type field stores a value that specifies the file type (see Table 18-4). The rec_len field may be interpreted as a pointer to the next valid directory entry: it is the offset to be added to the starting address of the directory entry to get the starting address of the next valid directory entry. To delete a directory entry, it is sufficient to set its inode field to 0 and suitably increment the value of the rec_len field of the previous valid entry. Read the rec_len field of Figure 18-3 carefully; you'll see that the *oldfile* entry was deleted because the rec_len field of *usr* is set to 12+16 (the lengths of the *usr* and *oldfile* entries).

Figure 18-3. An example of the Ext2 directory



18.2.6.3. Symbolic link

As stated before, if the pathname of a symbolic link has up to 60 characters, it is stored in the i_block field of the inode, which consists of an array of 15 4-byte integers; no data block is therefore required. If the pathname is longer than 60 characters, however, a single data block is required.

18.2.6.4. Device file, pipe, and socket

No data blocks are required for these kinds of files. All the necessary information is stored in the inode.

18.3. Ext2 Memory Data Structures

For the sake of efficiency, most information stored in the disk data structures of an Ext2 partition are copied into RAM when the filesystem is mounted, thus allowing the kernel to avoid many subsequent disk read operations. To get an idea of how often some data structures change, consider some fundamental operations:

- When a new file is created, the values of the s_free_inodes_count field in the Ext2 superblock and of the bg_free_inodes_count field in the proper group descriptor must be decreased.

- If the kernel appends some data to an existing file so that the number of data blocks allocated for it increases, the values of the `s_free_blocks_count` field in the Ext2 superblock and of the `bg_free_blocks_count` field in the group descriptor must be modified.
- Even just rewriting a portion of an existing file involves an update of the `s_wtime` field of the Ext2 superblock.

Because all Ext2 disk data structures are stored in blocks of the Ext2 partition, the kernel uses the page cache to keep them up-to-date (see the section "Writing Dirty Pages to Disk" in Chapter 15).

Table 18-6 specifies, for each type of data related to Ext2 filesystems and files, the data structure used on the disk to represent its data, the data structure used by the kernel in memory, and a rule of thumb used to determine how much caching is used. Data that is updated very frequently is always cached; that is, the data is permanently stored in memory and included in the page cache until the corresponding Ext2 partition is unmounted. The kernel gets this result by keeping the page's usage counter greater than 0 at all times.

Table 18-6. VFS images of Ext2 data structures

Type	Disk data structure	Memory data structure	Caching mode
Superblock	`ext2_super_block`	`ext2_sb_info`	Always cached
Group descriptor	`ext2_group_desc`	`ext2_group_desc`	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	`ext2_inode`	`ext2_inode_info`	Dynamic
Data block	Array of bytes	VFS buffer	Dynamic
Free inode	`ext2_inode`	None	Never
Free block	Array of bytes	None	Never

The never-cached data is not kept in any cache because it does not represent meaningful information. Conversely, the always-cached data is always present in RAM, thus it is never necessary to read the data from disk (periodically, however, the data must be written back to disk). In between these extremes lies the *dynamic* mode. In this mode, the data is kept in a cache as long as the associated object (inode, data block, or bitmap) is in use; when the file is closed or the data block is deleted, the page frame reclaiming algorithm may remove the associated data from the cache.

It is interesting to observe that inode and block bitmaps are not kept permanently in memory; rather, they are read from disk when needed. Actually, many disk reads are avoided thanks to the page cache, which keeps in memory the most recently used disk blocks (see the section "Storing Blocks in the Page Cache" in Chapter 15).[*]

[*] In Linux 2.4 and earlier versions, the most recently used inode and block bitmaps were stored in ad-hoc caches of bounded size.

18.3.1. The Ext2 Superblock Object

As stated in the section "Superblock Objects" in Chapter 12, the `s_fs_info` field of the VFS superblock points to a structure containing filesystem-specific data. In the case of Ext2, this field points to a structure of type `ext2_sb_info`, which includes the following information:

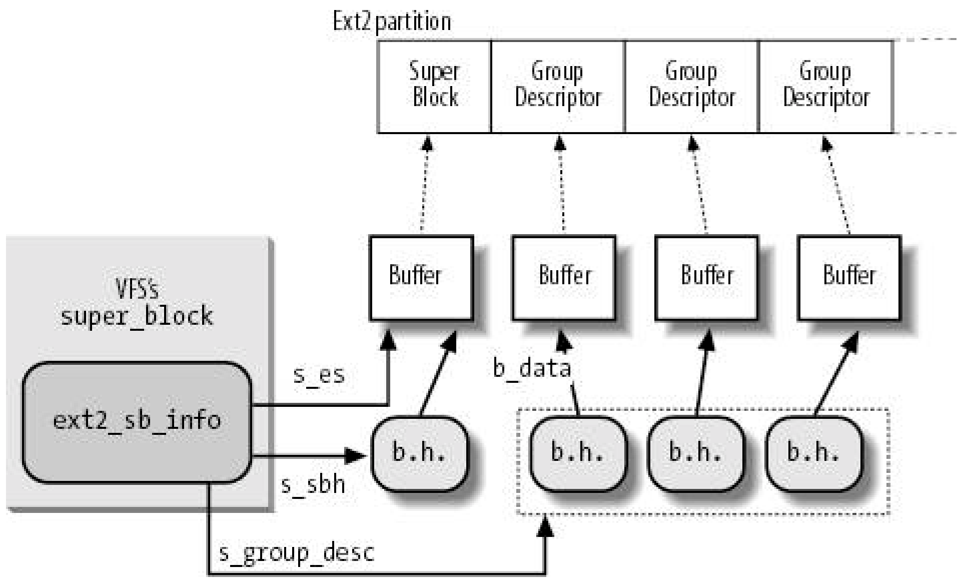
- Most of the disk superblock fields
- An `s_sbh` pointer to the buffer head of the buffer containing the disk superblock
- An `s_es` pointer to the buffer containing the disk superblock
- The number of group descriptors, `s_desc_ per_block`, that can be packed in a block
- An `s_group_desc` pointer to an array of buffer heads of buffers containing the group descriptors (usually, a single entry is sufficient)
- Other data related to mount state, mount options, and so on

Figure 18-4 shows the links between the `ext2_sb_info` data structures and the buffers and buffer heads relative to the Ext2 superblock and to the group descriptors.

When the kernel mounts an Ext2 filesystem, it invokes the `ext2_fill_super( )` function to allocate space for the data structures and to fill them with data read from disk (see the section "Mounting a Generic Filesystem" in Chapter 12). This is a simplified description of the function, which emphasizes the memory allocations for buffers and descriptors:

1. Allocates an `ext2_sb_info` descriptor and stores its address in the `s_fs_info` field of the superblock object passed as the parameter.

Figure 18-4. The ext2_sb_info data structure



2. Invokes _ _bread( ) to allocate a buffer in a buffer page together with the corresponding buffer head, and to read the superblock from disk into the buffer; as discussed in the section "Searching Blocks in the Page Cache" in Chapter 15, no allocation is performed if the block is already stored in a buffer page in the page cache and it is up-to-date. Stores the buffer head address in the s_sbh field of the Ext2 superblock object.
3. Allocates an array of bytesone byte for each groupand stores its address in the s_debts field of the ext2_sb_info descriptor (see the section "Creating inodes" later in this chapter).
4. Allocates an array of pointers to buffer heads, one for each group descriptor, and stores the address of the array in the s_group_desc field of the ext2_sb_info descriptor.
5. Invokes repeatedly _ _bread( ) to allocate buffers and to read from disk the blocks containing the Ext2 group descriptors; stores the addresses of the buffer heads in the s_group_desc array allocated in the previous step.
6. Allocates an inode and a dentry object for the root directory, and sets up a few fields of the superblock object so that it will be possible to read the root inode from disk.

Clearly, all the data structures allocated by ext2_fill_super( ) are kept in memory after the function returns; they will be released only when the Ext2 filesystem will be unmounted. When the kernel must modify a field in the Ext2 superblock, it simply writes the new value in the proper position of the corresponding buffer and then marks the buffer as dirty.

18.3.2. The Ext2 inode Object

When opening a file, a pathname lookup is performed. For each component of the pathname that is not already in the dentry cache , a new dentry object and a new inode object are created (see the section "Standard Pathname Lookup" in Chapter 12). When the VFS accesses an Ext2 disk inode, it creates a corresponding *inode descriptor* of type ext2_inode_info. This descriptor includes the following information:

- The whole VFS inode object (see Table 12-3 in Chapter 12) stored in the field vfs_inode
- Most of the fields found in the disk's inode structure that are not kept in the VFS inode
- The i_block_group block group index at which the inode belongs (see the section "Ext2 Disk Data Structures" earlier in this chapter)
- The i_next_alloc_block and i_next_alloc_goal fields, which store the logical block number and the physical block number of the disk block that was most recently allocated to the file, respectively
- The i_prealloc_block and i_prealloc_count fields, which are used for data block preallocation (see the section "Allocating a Data Block" later in this chapter)
- The xattr_sem field, a read/write semaphore that allows extended attributes to be read concurrently with the file data
- The i_acl and i_default_acl fields, which point to the ACLs of the file

When dealing with Ext2 files, the `alloc_inode` superblock method is implemented by means of the `ext2_alloc_inode( )` function. It gets first an `ext2_inode_info` descriptor from the `ext2_inode_cachep` slab allocator cache, then it returns the address of the inode object embedded in the new `ext2_inode_info` descriptor.

18.4. Creating the Ext2 Filesystem

There are generally two stages to creating a filesystem on a disk. The first step is to format it so that the disk driver can read and write blocks on it. Modern hard disks come preformatted from the factory and need not be reformatted; floppy disks may be formatted on Linux using a utility program such as *superformat* or *fdformat*. The second step involves creating a filesystem, which means setting up the structures described in detail earlier in this chapter.

Ext2 filesystems are created by the *mke2fs* utility program; it assumes the following default options, which may be modified by the user with flags on the command line:

- Block size: 1,024 bytes (default value for a small filesystem)
- Fragment size: block size (block fragmentation is not implemented)
- Number of allocated inodes: 1 inode for each 8,192 bytes
- Percentage of reserved blocks: 5 percent

The program performs the following actions:

1. Initializes the superblock and the group descriptors.
2. Optionally, checks whether the partition contains defective blocks; if so, it creates a list of defective blocks.
3. For each block group, reserves all the disk blocks needed to store the superblock, the group descriptors, the inode table, and the two bitmaps.
4. Initializes the inode bitmap and the data map bitmap of each block group to 0.
5. Initializes the inode table of each block group.
6. Creates the */root* directory.
7. Creates the *lost+found* directory, which is used by *e2fsck* to link the lost and found defective blocks.
8. Updates the inode bitmap and the data block bitmap of the block group in which the two previous directories have been created.
9. Groups the defective blocks (if any) in the *lost+found* directory.

Let's consider how an Ext2 1.44 MB floppy disk is initialized by *mke2fs* with the default options.

Once mounted, it appears to the VFS as a volume consisting of 1,412 blocks; each one is 1,024 bytes in length. To examine the disk's contents, we can execute the Unix command:

```
$ dd if=/dev/fd0 bs=1k count=1440 | od -tx1 -Ax > /tmp/dump_hex
```

to get a file containing the hexadecimal dump of the floppy disk contents in the */tmp* directory.[*]

[*] Most information on an Ext2 filesystem could also be obtained by using the *dumpe2fs* and *debugfs* utility programs.

By looking at that file, we can see that, due to the limited capacity of the disk, a single group descriptor is sufficient. We also notice that the number of reserved blocks is set to 72 (5 percent of 1,440) and, according to the default option, the inode table must include 1 inode for each 8,192 bytes that is, 184 inodes stored in 23 blocks.

Table 18-7 summarizes how the Ext2 filesystem is created on a floppy disk when the default options are selected.

Table 18-7. Ext2 block allocation for a floppy disk

Block	Content
0	Boot block
1	Superblock
2	Block containing a single block group descriptor
3	Data block bitmap
4	inode bitmap
5-27	inode table: inodes up to 10: reserved (inode 2 is the root); inode 11: *lost+found*; inodes 12-184: free
28	Root directory (includes `.`, `..`, and *lost+found*)
29	*lost+found* directory (includes `.` and `..`)
30-40	Reserved blocks preallocated for *lost+found* directory
41-1439	Free blocks

18.5. Ext2 Methods

Many of the VFS methods described in Chapter 12 have a corresponding Ext2 implementation. Because it would take a whole book to describe all of them, we limit ourselves to briefly reviewing the methods implemented in Ext2. Once the disk and the memory data structures are clearly understood, the reader should be able to follow the code of the Ext2 functions that implement them.

18.5.1. Ext2 Superblock Operations

Many VFS superblock operations have a specific implementation in Ext2, namely `alloc_inode`, `destroy_inode`, `read_inode`, `write_inode`, `delete_inode`, `put_super`, `write_super`, `statfs`, `remount_fs`, and `clear_inode`. The addresses of the superblock methods are stored in the `ext2_sops` array of pointers.

18.5.2. Ext2 inode Operations

Some of the VFS inode operations have a specific implementation in Ext2, which depends on the type of the file to which the inode refers.

The inode operations for Ext2 regular files and Ext2 directories are shown in Table 18-8; the purpose of each method is described in the section "Inode Objects" in Chapter 12. The table does not show the methods that are undefined (a `NULL` pointer) for both regular files and directories; recall that if a method is undefined, the VFS either invokes a generic function or does nothing at all. The addresses of the Ext2 methods for regular files and directories are stored in the `ext2_file_inode_operations` and `ext2_dir_inode_operations` tables, respectively.

Table 18-8. Ext2 inode operations for regular files and directories

VFS inode operation	Regular file	Directory
`create`	`NULL`	`ext2_create( )`
`lookup`	`NULL`	`ext2_lookup( )`
`link`	`NULL`	`ext2_link( )`
`unlink`	`NULL`	`ext2_unlink( )`
`symlink`	`NULL`	`ext2_symlink( )`
`mkdir`	`NULL`	`ext2_mkdir( )`
`rmdir`	`NULL`	`ext2_rmdir( )`
`mknod`	`NULL`	`ext2_mknod( )`
`rename`	`NULL`	`ext2_rename( )`
`truncate`	`ext2_TRuncate( )`	`NULL`
`permission`	`ext2_permission( )`	`ext2_permission( )`
`setattr`	`ext2_setattr( )`	`ext2_setattr( )`
`setxattr`	`generic_setxattr( )`	`generic_setxattr( )`
`getxattr`	`generic_getxattr( )`	`generic_getxattr( )`
`listxattr`	`ext2_listxattr( )`	`ext2_listxattr( )`
`removexattr`	`generic_removexattr( )`	`generic_removexattr( )`

The inode operations for Ext2 symbolic links are shown in Table 18-9 (undefined methods have been omitted). Actually, there are two types of symbolic links: the fast symbolic links represent pathnames that can be fully stored inside the inodes, while the regular symbolic links represent longer pathnames. Accordingly, there are two sets of inode operations, which are stored in the `ext2_fast_symlink_inode_operations` and `ext2_symlink_inode_operations` tables, respectively.

Table 18-9. Ext2 inode operations for fast and regular symbolic links

VFS inode operation	Fast symbolic link	Regular symbolic link
`readlink`	`generic_readlink( )`	`generic_readlink( )`
`follow_link`	`ext2_follow_link( )`	`page_follow_link_light( )`
`put_link`	`NULL`	`page_put_link( )`
`setxattr`	`generic_setxattr( )`	`generic_setxattr( )`
`getxattr`	`generic_getxattr( )`	`generic_getxattr( )`
`listxattr`	`ext2_listxattr( )`	`ext2_listxattr( )`
`removexattr`	`generic_removexattr( )`	`generic_removexattr( )`

If the inode refers to a character device file, to a block device file, or to a named pipe (see "FIFOs" in Chapter 19), the inode operations do not depend on the filesystem. They are specified in the `chrdev_inode_operations`, `blkdev_inode_operations`, and `fifo_inode_operations` tables, respectively.

18.5.3. Ext2 File Operations

The file operations specific to the Ext2 filesystem are listed in Table 18-10. As you can see, several VFS methods are implemented by generic functions that are common to many filesystems. The addresses of these methods are stored in the `ext2_file_operations` table.

Table 18-10. Ext2 file operations

VFS file operation	Ext2 method
`llseek`	`generic_file_llseek( )`
`read`	`generic_file_read( )`
`write`	`generic_file_write( )`
aio_read	generic_file_aio_read( )
aio_write	generic_file_aio_write( )
`ioctl`	`ext2_ioctl( )`
`mmap`	`generic_file_mmap( )`
`open`	`generic_file_open( )`
`release`	`ext2_release_file( )`
`fsync`	`ext2_sync_file( )`
readv	generic_file_readv( )
writev	generic_file_writev( )
sendfile	generic_file_sendfile( )

Notice that the Ext2's `read` and `write` methods are implemented by the `generic_file_read( )` and `generic_file_write( )` functions, respectively. These are described in the sections "Reading from a File" and "Writing to a File" in Chapter 16.

18.6. Managing Ext2 Disk Space

The storage of a file on disk differs from the view the programmer has of the file in two ways: blocks can be scattered around the disk (although the filesystem tries hard to keep blocks sequential to improve access time), and files may appear to a programmer to be bigger than they really are because a program can introduce holes into them (through the `lseek( )` system call).

In this section, we explain how the Ext2 filesystem manages the disk space how it allocates and deallocates inodes and data blocks. Two main problems must be addressed:

- Space management must make every effort to avoid *file fragmentation* the physical storage of a file in several, small pieces located in non-adjacent disk blocks. File fragmentation increases the average time of sequential read operations on the files, because the disk heads must be frequently repositioned during the read operation.[*] This problem is similar to the external fragmentation of RAM discussed in the section "The Buddy System Algorithm" in Chapter 8.

  [*] Please note that fragmenting a file across block groups (A Bad Thing) is quite different from the not-yet-implemented fragmentation of blocks to store many files in one block (A Good Thing).

- Space management must be time-efficient; that is, the kernel should be able to quickly derive from a file offset the corresponding logical block number in the Ext2 partition. In doing so, the kernel should limit as much as possible the number of accesses to addressing tables stored on disk, because each such intermediate access considerably increases the average file access time.

18.6.1. Creating inodes

The `ext2_new_inode( )` function creates an Ext2 disk inode, returning the address of the corresponding inode object (or `NULL`, in case of failure). The function carefully selects the block group that contains the new inode; this is done to spread unrelated directories among different groups and, at the same time, to put files into the same group as their parent directories. To balance the number of regular files and directories in a block group, Ext2 introduces a "debt" parameter for every block group.

The function acts on two parameters: the address `dir` of the inode object that refers to the directory into which the new inode must be inserted and a `mode` that indicates the type of inode being created. The latter argument also includes the `MS_SYNCHRONOUS` mount flag (see the section "Mounting a Generic Filesystem" in

Chapter 12) that requires the current process to be suspended until the inode is allocated. The function performs the following actions:

1. Invokes `new_inode( )` to allocate a new VFS inode object; initializes its `i_sb` field to the superblock address stored in `dir->i_sb`, and adds it to the in-use inode list and to the superblock's list (see the section "Inode Objects" in Chapter 12).
2. If the new inode is a directory, the function invokes `find_group_orlov( )` to find a suitable block group for the directory.[*] This function implements the following heuristics:

   [*] The Ext2 filesystem may also be mounted with an option flag that forces the kernel to make use of a simpler, older allocation strategy, which is implemented by the `find_group_dir( )` function.

   a. Directories having as parent the filesystem root should be spread among all block groups. Thus, the function searches the block groups looking for a group having a number of free inodes and a number of free blocks above the average. If there is no such group, it jumps to step 2c.
   b. Nested directoriesnot having the filesystem root as parentshould be put in the group of the parent if it satisfies the following rules:
      - The group does not contain too many directories
      - The group has a sufficient number of free inodes left
      - The group has a small "debt" (the debt of a block group is stored in the array of counters pointed to by the `s_debts` field of the `ext2_sb_info` descriptor; the debt is increased each time a new directory is added and decreased each time another type of file is added)

      If the parent's group does not satisfy these rules, it picks the first group that satisfies them. If no such group exists, it jumps to step 2c.

   c. This is the "fallback" rule, to be used if no good group has been found. The function starts with the block group containing the parent directory and selects the first block group that has more free inodes than the average number of free inodes per block group.
3. If the new inode is not a directory, it invokes `find_group_other( )` to allocate it in a block group having a free inode. This function selects the group by starting from the one that contains the parent directory and moving farther away from it; to be precise:
   a. Performs a quick logarithmic search starting from the block group that includes the parent directory `dir`. The algorithm searches log(*n*) block groups, where *n* is the total number of block groups. The algorithm jumps further ahead until it finds an available block group for example, if we call the number of the starting block group *i*, the algorithm considers block groups *i mod*(*n*), *i*+1 *mod*(*n*), *i*+1+2 *mod*(*n*), *i*+1+2+4 *mod*(*n*), etc.
   b. If the logarithmic search failed in finding a block group with a free inode, the function performs an exhaustive linear search starting from the block group that includes the parent directory `dir`.

4. Invokes read_inode_bitmap( ) to get the inode bitmap of the selected block group and searches for the first null bit into it, thus obtaining the number of the first free disk inode.
5. Allocates the disk inode: sets the corresponding bit in the inode bitmap and marks the buffer containing the bitmap as dirty. Moreover, if the filesystem has been mounted specifying the MS_SYNCHRONOUS flag (see the section "Mounting a Generic Filesystem" in Chapter 12), the function invokes sync_dirty_buffer( ) to start the I/O write operation and waits until the operation terminates.
6. Decreases the bg_free_inodes_count field of the group descriptor. If the new inode is a directory, the function increases the bg_used_dirs_count field and marks the buffer containing the group descriptor as dirty.
7. Increases or decreases the group's counter in the s_debts array of the superblock, according to whether the inode refers to a regular file or a directory.
8. Decreases the s_freeinodes_counter field of the ext2_sb_info data structure; moreover, if the new inode is a directory, it increases the s_dirs_counter field in the ext2_sb_info data structure.
9. Sets the s_dirt flag of the superblock to 1, and marks the buffer that contains it to as dirty.
10. Sets the s_dirt field of the VFS's superblock object to 1.
11. Initializes the fields of the inode object. In particular, it sets the inode number i_no and copies the value of xtime.tv_sec into i_atime, i_mtime, and i_ctime. Also loads the i_block_group field in the ext2_inode_info structure with the block group index. Refer to Table 18-3 for the meaning of these fields.
12. Initializes the ACLs of the inode.
13. Inserts the new inode object into the hash table inode_hashtable and invokes mark_inode_dirty( ) to move the inode object into the superblock's dirty inode list (see the section "Inode Objects" in Chapter 12).
14. Invokes ext2_preread_inode( ) to read from disk the block containing the inode and to put the block in the page cache. This type of read-ahead is done because it is likely that a recently created inode will be written back soon.
15. Returns the address of the new inode object.

18.6.2. Deleting inodes

The ext2_free_inode( ) function deletes a disk inode, which is identified by an inode object whose address inode is passed as the parameter. The kernel should invoke the function after a series of cleanup operations involving internal data structures and the data in the file itself. It should come after the inode object has been removed from the inode hash table, after the last hard link referring to that inode has been deleted from the proper directory and after the file is truncated to 0 length to reclaim all its data blocks (see the section "Releasing a Data Block" later in this chapter). It performs the following actions:

1. Invokes clear_inode( ), which in turn executes the following operations:
    a. Removes any dirty "indirect" buffer associated with the inode (see the later section "Data Blocks Addressing"); they are collected in the list headed at the private_list field of the address_space object inode->i_data (see the section "The address_space Object" in Chapter 15).

b. If the I_LOCK flag of the inode is set, some of the inode's buffers are involved in I/O data transfers; the function suspends the current process until these I/O data transfers terminate.
c. Invokes the clear_inode method of the superblock object, if defined; the Ext2 filesystem does not define it.
d. If the inode refers to a device file, it removes the inode object from the device's list of inodes; this list is rooted either in the list field of the cdev character device descriptor (see the section "Character Device Drivers" in Chapter 13) or in the bd_inodes field of the block_device block device descriptor (see the section "Block Devices" in Chapter 14).
e. Sets the state of the inode to I_CLEAR (the inode object contents are no longer meaningful).

2. Computes the index of the block group containing the disk inode from the inode number and the number of inodes in each block group.
3. Invokes read_inode_bitmap( ) to get the inode bitmap.
4. Increases the bg_free_inodes_count( ) field of the group descriptor. If the deleted inode is a directory, it decreases the bg_used_dirs_count field. Marks the buffer that contains the group descriptor as dirty.
5. If the deleted inode is a directory, it decreases the s_dirs_counter field in the ext2_sb_info data structure, sets the s_dirt flag of the superblock to 1, and marks the buffer that contains it as dirty.
6. Clears the bit corresponding to the disk inode in the inode bitmap and marks the buffer that contains the bitmap as dirty. Moreover, if the filesystem has been mounted with the MS_SYNCHRONIZE flag, it invokes sync_dirty_buffer( ) to wait until the write operation on the bitmap's buffer terminates.

18.6.3. Data Blocks Addressing

Each nonempty regular file consists of a group of data blocks . Such blocks may be referred to either by their relative position inside the file their file block numberor by their position inside the disk partitiontheir logical block number (see the section "Block Devices Handling" in Chapter 14).

Deriving the logical block number of the corresponding data block from an offset *f* inside a file is a two-step process:

1. Derive from the offset *f* the file block number the index of the block that contains the character at offset *f*.
2. Translate the file block number to the corresponding logical block number.

Because Unix files do not include any control characters, it is quite easy to derive the file block number containing the *f* th character of a file: simply take the quotient of *f* and the filesystem's block size and round down to the nearest integer.

For instance, let's assume a block size of 4 KB. If *f* is smaller than 4,096, the character is contained in the first data block of the file, which has file block number 0. If *f* is equal to or greater than 4,096 and less than 8,192, the character is contained in the data block that has file block number 1, and so on.

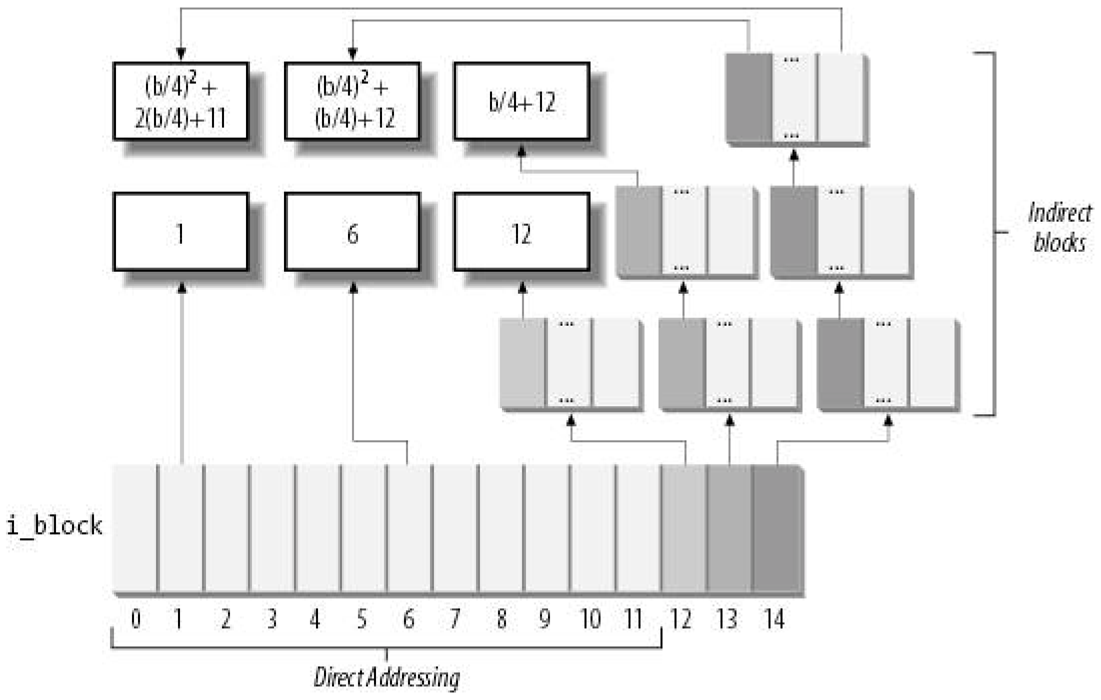
This is fine as far as file block numbers are concerned. However, translating a file block number into the corresponding logical block number is not nearly as straightforward, because the data blocks of an Ext2 file are not necessarily adjacent on disk.

The Ext2 filesystem must therefore provide a method to store the connection between each file block number and the corresponding logical block number on disk. This mapping, which goes back to early versions of Unix from AT&T, is implemented partly inside the inode. It also involves some specialized blocks that contain extra pointers, which are an inode extension used to handle large files.

The `i_block` field in the disk inode is an array of `EXT2_N_BLOCKS` components that contain logical block numbers. In the following discussion, we assume that `EXT2_N_BLOCKS` has the default value, namely 15. The array represents the initial part of a larger data structure, which is illustrated in Figure 18-5. As can be seen in the figure, the 15 components of the array are of 4 different types:

- The first 12 components yield the logical block numbers corresponding to the first 12 blocks of the fileto the blocks that have file block numbers from 0 to 11.
- The component at index 12 contains the logical block number of a block, called *indirect block*, that represents a second-order array of logical block numbers. They correspond to the file block numbers ranging from 12 to $b/4+11$, where b is the filesystem's block size (each logical block number is stored in 4 bytes, so we divide by 4 in the formula). Therefore, the kernel must look in this component for a pointer to a block, and then look in that block for another pointer to the ultimate block that contains the file contents.
- The component at index 13 contains the logical block number of an indirect block containing a second-order array of logical block numbers; in turn, the entries of this second-order array point to third-order arrays, which store the logical block numbers that correspond to the file block numbers ranging from $b/4+12$ to $(b/4)^2+(b/4)+11$.
- Finally, the component at index 14 uses triple indirection: the fourth-order arrays store the logical block numbers corresponding to the file block numbers ranging from $(b/4)^2+(b/4)+12$ to $(b/4)^3+(b/4)^2+(b/4)+11$.

Figure 18-5. Data structures used to address the file's data blocks



In Figure 18-5, the number inside a block represents the corresponding file block number. The arrows, which represent logical block numbers stored in array components, show how the kernel finds its way through indirect blocks to reach the block that contains the actual contents of the file.

Notice how this mechanism favors small files. If the file does not require more than 12 data blocks, every data can be retrieved in two disk accesses: one to read a component in the `i_block` array of the disk inode and the other to read the requested data block. For larger files, however, three or even four consecutive disk accesses may be needed to access the required block. In practice, this is a worst-case estimate, because dentry, inode, and page caches contribute significantly to reduce the number of real disk accesses.

Notice also how the block size of the filesystem affects the addressing mechanism, because a larger block size allows the Ext2 to store more logical block numbers inside a single block. Table 18-11 shows the upper limit placed on a file's size for each block size and each addressing mode. For instance, if the block size is 1,024 bytes and the file contains up to 268 kilobytes of data, the first 12 KB of a file can be accessed through direct mapping and the remaining 13-268 KB can be addressed through simple indirection. Files larger than 2 GB must be opened on 32-bit architectures by specifying the `O_LARGEFILE` opening flag.

Table 18-11. File-size upper limits for data block addressing

Block size	Direct	1-Indirect	2-Indirect	3-Indirect
1,024	12 KB	268 KB	64.26 MB	16.06 GB

Table 18-11. File-size upper limits for data block addressing

Block size	Direct	1-Indirect	2-Indirect	3-Indirect
2,048	24 KB	1.02 MB	513.02 MB	256.5 GB
4,096	48 KB	4.04 MB	4 GB	~ 4 TB

18.6.4. File Holes

A *file hole* is a portion of a regular file that contains null characters and is not stored in any data block on disk. Holes are a long-standing feature of Unix files. For instance, the following Unix command creates a file in which the first bytes are a hole:

```
$ echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

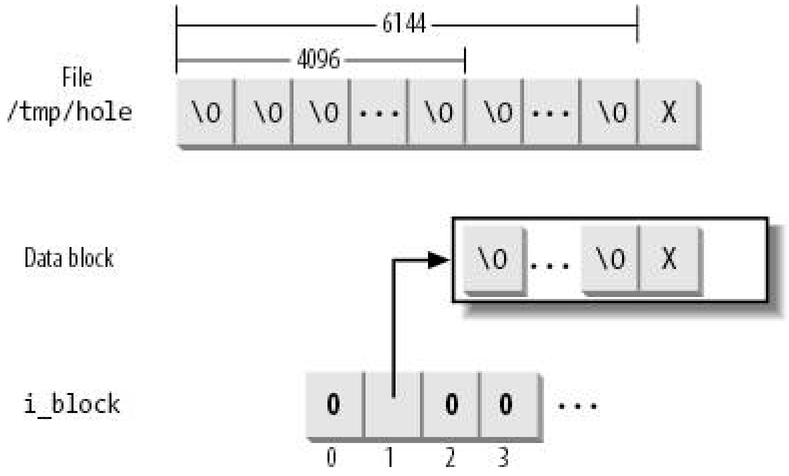
Now */tmp/hole* has 6,145 characters (6,144 null characters plus an X character), yet the file occupies just one data block on disk.

File holes were introduced to avoid wasting disk space. They are used extensively by database applications and, more generally, by all applications that perform hashing on files.

The Ext2 implementation of file holes is based on dynamic data block allocation: a block is actually assigned to a file only when the process needs to write data into it. The `i_size` field of each inode defines the size of the file as seen by the program, including the holes, while the `i_blocks` field stores the number of data blocks effectively assigned to the file (in units of 512 bytes).

In our earlier example of the `dd` command, suppose the */tmp/hole* file was created on an Ext2 partition that has blocks of size 4,096. The `i_size` field of the corresponding disk inode stores the number 6,145, while the `i_blocks` field stores the number 8 (because each 4,096-byte block includes eight 512-byte blocks). The second element of the `i_block` array (corresponding to the block having file block number 1) stores the logical block number of the allocated block, while all other elements in the array are null (see Figure 18-6).

Figure 18-6. A file with an initial hole



18.6.5. Allocating a Data Block

When the kernel has to locate a block holding data for an Ext2 regular file, it invokes the `ext2_get_block( )` function. If the block does not exist, the function automatically allocates the block to the file. Remember that this function may be invoked every time the kernel issues a read or write operation on an Ext2 regular file (see the sections "Reading from a File" and "Writing to a File" in Chapter 16); clearly, this function is invoked only if the affected block is not included in the page cache.

The `ext2_get_block( )` function handles the data structures already described in the section "Data Blocks Addressing," and when necessary, invokes the `ext2_alloc_block( )` function to actually search for a free block in the Ext2 partition. If necessary, the function also allocates the blocks used for indirect addressing (see Figure 18-5).

To reduce file fragmentation, the Ext2 filesystem tries to get a new block for a file near the last block already allocated for the file. Failing that, the filesystem searches for a new block in the block group that includes the file's inode. As a last resort, the free block is taken from one of the other block groups.

The Ext2 filesystem uses preallocation of data blocks. The file does not get only the requested block, but rather a group of up to eight adjacent blocks. The `i_prealloc_count` field in the `ext2_inode_info` structure stores the number of data blocks preallocated to a file that are still unused, and the `i_prealloc_block` field stores the logical block number of the next preallocated block to be used. All preallocated blocks that remain unused are freed when the file is closed, when it is truncated, or when a write operation is not sequential with respect to the write operation that triggered the block preallocation.

The `ext2_alloc_block( )` function receives as its parameters a pointer to an inode object, a *goal* , and the address of a variable that will store an error code. The goal is a logical block number that represents the preferred position of the new block. The `ext2_get_block( )` function sets the goal parameter according to the following heuristic:

1. If the block that is being allocated and the previously allocated block have consecutive file block numbers, the goal is the logical block number of the previous block plus 1; it makes sense that consecutive blocks as seen by a program should be adjacent on disk.
2. If the first rule does not apply and at least one block has been previously allocated to the file, the goal is one of these blocks' logical block numbers. More precisely, it is the logical block number of the already allocated block that precedes the block to be allocated in the file.
3. If the preceding rules do not apply, the goal is the logical block number of the first block (not necessarily free) in the block group that contains the file's inode.

The `ext2_alloc_block( )` function checks whether the goal refers to one of the preallocated blocks of the file. If so, it allocates the corresponding block and returns its logical block number; otherwise, the function discards all remaining preallocated blocks and invokes `ext2_new_block( )`.

This latter function searches for a free block inside the Ext2 partition with the following strategy:

1. If the preferred block passed to `ext2_alloc_block( )`the block that is the goalis free, the function allocates the block.
2. If the goal is busy, the function checks whether one of the next blocks after the preferred block is free.
3. If no free block is found in the near vicinity of the preferred block, the function considers all block groups, starting from the one including the goal. For each block group, the function does the following:
    a. Looks for a group of at least eight adjacent free blocks.
    b. If no such group is found, looks for a single free block.

The search ends as soon as a free block is found. Before terminating, the `ext2_new_block( )` function also tries to preallocate up to eight free blocks adjacent to the free block found and sets the `i_prealloc_block` and `i_prealloc_count` fields of the disk inode to the proper block location and number of blocks.

18.6.6. Releasing a Data Block

When a process deletes a file or truncates it to 0 length, all its data blocks must be reclaimed. This is done by `ext2_truncate( )`, which receives the address of the file's inode object as its parameter. The function essentially scans the disk inode's `i_block` array to locate all data blocks and all blocks used for the indirect addressing. These blocks are then released by repeatedly invoking `ext2_free_blocks( )`.

The `ext2_free_blocks( )` function releases a group of one or more adjacent data blocks. Besides its use by `ext2_truncate( )`, the function is invoked mainly when discarding the preallocated blocks of a file (see the earlier section "Allocating a Data Block"). Its parameters are:

`inode`

The address of the inode object that describes the file

`block`

The logical block number of the first block to be released

`count`

The number of adjacent blocks to be released

The function performs the following actions for each block to be released:

1. Gets the block bitmap of the block group that includes the block to be released
2. Clears the bit in the block bitmap that corresponds to the block to be released and marks the buffer that contains the bitmap as dirty.
3. Increases the `bg_free_blocks_count` field in the block group descriptor and marks the corresponding buffer as dirty.
4. Increases the `s_free_blocks_count` field of the disk superblock, marks the corresponding buffer as dirty, and sets the `s_dirt` flag of the superblock object.
5. If the filesystem has been mounted with the `MS_SYNCHRONOUS` flag set, it invokes `sync_dirty_buffer( )` and waits until the write operation on the bitmap's buffer terminates.

18.7. The Ext3 Filesystem

In this section we'll briefly describe the enhanced filesystem that has evolved from Ext2, named *Ext3*. The new filesystem has been designed with two simple concepts in mind:

- To be a journaling filesystem (see the next section)
- To be, as much as possible, compatible with the old Ext2 filesystem

Ext3 achieves both the goals very well. In particular, it is largely based on Ext2, so its data structures on disk are essentially identical to those of an Ext2 filesystem. As a matter of fact, if an Ext3 filesystem has been cleanly unmounted, it can be remounted as an Ext2 filesystem; conversely, creating a journal of an Ext2 filesystem and remounting it as an Ext3 filesystem is a simple, fast operation.

Thanks to the compatibility between Ext3 and Ext2, most descriptions in the previous sections of this chapter apply to Ext3 as well. Therefore, in this section, we focus on the new feature offered by Ext3 "the journal."

18.7.1. Journaling Filesystems

As disks became larger, one design choice of traditional Unix filesystems (such as Ext2) turns out to be inappropriate. As we know from Chapter 14, updates to filesystem blocks might be kept in dynamic memory for long period of time before being flushed to disk. A dramatic event such as a power-down failure or a system crash might thus leave the filesystem in an inconsistent state. To overcome this problem, each traditional Unix filesystem is checked before being mounted; if it has not been properly unmounted, then a specific program executes an exhaustive, time-consuming check and fixes all the filesystem's data structures on disk.

For instance, the Ext2 filesystem status is stored in the `s_mount_state` field of the superblock on disk. The *e2fsck* utility program is invoked by the boot script to check the value stored in this field; if it is not equal to `EXT2_VALID_FS`, the filesystem was not properly unmounted, and therefore *e2fsck* starts checking all disk data structures of the filesystem.

Clearly, the time spent checking the consistency of a filesystem depends mainly on the number of files and directories to be examined; therefore, it also depends on the disk size. Nowadays, with filesystems reaching hundreds of gigabytes, a single consistency check may take hours. The involved downtime is unacceptable for every production environment or high-availability server.

The goal of a *journaling filesystem* is to avoid running time-consuming consistency checks on the whole filesystem by looking instead in a special disk area that contains the most recent disk write operations named *journal*. Remounting a journaling filesystem after a system failure is a matter of a few seconds.

18.7.2. The Ext3 Journaling Filesystem

The idea behind Ext3 journaling is to perform each high-level change to the filesystem in two steps. First, a copy of the blocks to be written is stored in the journal; then, when the I/O data transfer to the journal is completed (in short, data is *committed to the journal*), the blocks are written in the filesystem. When the I/O data transfer to the filesystem terminates (data is *committed to the filesystem*), the copies of the blocks in the journal are discarded.

While recovering after a system failure, the *e2fsck* program distinguishes the following two cases:

The system failure occurred before a commit to the journal

Either the copies of the blocks relative to the high-level change are missing from the journal or they are incomplete; in both cases, *e2fsck* ignores them.

The system failure occurred after a commit to the journal

The copies of the blocks are valid, and *e2fsck* writes them into the filesystem.

In the first case, the high-level change to the filesystem is lost, but the filesystem state is still consistent. In the second case, *e2fsck* applies the whole high-level change, thus fixing every inconsistency due to unfinished I/O data transfers into the filesystem.

Don't expect too much from a journaling filesystem; it ensures consistency only at the system call level. For instance, a system failure that occurs while you are copying a large file by issuing several `write( )` system calls will interrupt the copy operation, thus the duplicated file will be shorter than the original one.

Furthermore, journaling filesystems do not usually copy all blocks into the journal. In fact, each filesystem consists of two kinds of blocks: those containing the so-called *metadata* and those containing regular data. In the case of Ext2 and Ext3, there are six kinds of metadata: superblocks, group block descriptors, inodes, blocks used for indirect addressing (indirection blocks), data bitmap blocks, and inode bitmap blocks. Other filesystems may use different metadata.

Several journaling filesystems, such as SGI's XFS and IBM's JFS , limit themselves to logging the operations affecting metadata. In fact, metadata's log records are sufficient to restore the consistency of the on-disk filesystem data structures. However, since operations on blocks of file data are not logged, nothing prevents a system failure from corrupting the contents of the files.

The Ext3 filesystem, however, can be configured to log the operations affecting both the filesystem metadata and the data blocks of the files. Because logging every kind of write operation leads to a significant performance penalty, Ext3 lets the system administrator decide what has to be logged; in particular, it offers three different journaling modes :

Journal

All filesystem data and metadata changes are logged into the journal. This mode minimizes the chance of losing the updates made to each file, but it requires many additional disk accesses. For example, when a new file is created, all its data blocks must be duplicated as log records. This is the safest and slowest Ext3 journaling mode.

Ordered

Only changes to filesystem metadata are logged into the journal. However, the Ext3 filesystem groups metadata and relative data blocks so that data blocks are written to disk *before* the metadata. This way, the chance to have data corruption inside the files is reduced; for instance, each write access that enlarges a file is guaranteed to be fully protected by the journal. This is the default Ext3 journaling mode.

Writeback

Only changes to filesystem metadata are logged; this is the method found on the other journaling filesystems and is the fastest mode.

The journaling mode of the Ext3 filesystem is specified by an option of the *mount* system command. For instance, to mount an Ext3 filesystem stored in the */dev/sda2* partition on the */jdisk* mount point with the "writeback" mode, the system administrator can type the command:

```
# mount -t ext3 -o data=writeback /dev/sda2 /jdisk
```

18.7.3. The Journaling Block Device Layer

The Ext3 journal is usually stored in a hidden file named *.journal* located in the root directory of the filesystem.

The Ext3 filesystem does not handle the journal on its own; rather, it uses a general kernel layer named *Journaling Block Device*, or *JBD*. Right now, only Ext3 uses the JBD layer, but other filesystems might use it in the future.

The JBD layer is a rather complex piece of software. The Ext3 filesystem invokes the JBD routines to ensure that its subsequent operations don't corrupt the disk data structures in case of system failure. However, JBD typically uses the same disk to log the changes performed by the Ext3 filesystem, and it is therefore vulnerable to system failures as much as Ext3. In other words, JBD must also protect itself from system failures that could corrupt the journal.

Therefore, the interaction between Ext3 and JBD is essentially based on three fundamental units:

Log record

Describes a single update of a disk block of the journaling filesystem.

Atomic operation handle

Includes log records relative to a single high-level change of the filesystem; typically, each system call modifying the filesystem gives rise to a single atomic operation handle.

Transaction

Includes several atomic operation handles whose log records are marked valid for *e2fsck* at the same time.

18.7.3.1. Log records

A *log record* is essentially the description of a low-level operation that is going to be issued by the filesystem. In some journaling filesystems, the log record consists of exactly the span of bytes modified by the operation, together with the starting position of the bytes inside the filesystem. The JBD layer, however, uses log records consisting of the whole buffer modified by the low-level operation. This approach may waste a lot of journal space (for instance, when the low-level operation just changes the value of a bit in a bitmap), but it is also much faster because the JBD layer can work directly with buffers and their buffer heads.

Log records are thus represented inside the journal as normal blocks of data (or metadata). Each such block, however, is associated with a small tag of type `journal_block_tag_t`, which stores the logical block number of the block inside the filesystem and a few status flags.

Later, whenever a buffer is being considered by the JBD, either because it belongs to a log record or because it is a data block that should be flushed to disk before the corresponding metadata block (in the "ordered" journaling mode), the kernel attaches a `journal_head` data structure to the buffer head. In this case, the `b_private` field of the buffer head stores the address of the `journal_head` data structure and the `BH_JBD` flag is set (see the section "Block Buffers and Buffer Heads" in Chapter 15).

18.7.3.2. Atomic operation handles

Every system call modifying the filesystem is usually split into a series of low-level operations that manipulate disk data structures.

For instance, suppose that Ext3 must satisfy a user request to append a block of data to a regular file. The filesystem layer must determine the last block of the file, locate a free block in the filesystem, update the data block bitmap inside the proper block group, store the logical number of the new block either in the file's inode or in an indirect addressing block, write the contents of the new block, and finally, update several fields of the inode. As you see, the append operation translates into many lower-level operations on the data and metadata blocks of the filesystem.

Now, just imagine what could happen if a system failure occurred in the middle of an append operation, when some of the lower-level manipulations have already been executed while others have not. Of course, the scenario could be even worse, with high-level operations affecting two or more files (for example, moving a file from one directory to another).

To prevent data corruption, the Ext3 filesystem must ensure that each system call is handled in an atomic way. An *atomic operation handle* is a set of low-level operations on the disk data structures that correspond to a single high-level operation. When recovering from a system failure, the filesystem ensures that either the whole high-level operation is applied or none of its low-level operations is.

Each atomic operation handle is represented by a descriptor of type `handle_t`. To start an atomic operation, the Ext3 filesystem invokes the `journal_start( )` JBD function, which allocates, if necessary, a new atomic operation handle and inserts it into the current transactions (see the next section). Because every low-level operation on the disk might suspend the process, the address of the active handle is stored in the `journal_info` field of the process descriptor. To notify that an atomic operation is completed, the Ext3 filesystem invokes the `journal_stop( )` function.

18.7.3.3. Transactions

For reasons of efficiency, the JBD layer manages the journal by grouping the log records that belong to several atomic operation handles into a single *transaction*. Furthermore, all log records relative to a handle must be included in the same transaction.

All log records of a transaction are stored in consecutive blocks of the journal. The JBD layer handles each transaction as a whole. For instance, it reclaims the blocks used by a transaction only after all data included in its log records is committed to the filesystem.

As soon as it is created, a transaction may accept log records of new handles. The transaction stops accepting new handles when either of the following occurs:

- A fixed amount of time has elapsed, typically 5 seconds.
- There are no free blocks in the journal left for a new handle.

A transaction is represented by a descriptor of type `TRansaction_t`. The most important field is `t_state`, which describes the current status of the transaction.

Essentially, a transaction can be:

Complete

All log records included in the transaction have been physically written onto the journal. When recovering from a system failure, *e2fsck* considers every complete transaction of the journal and writes the corresponding blocks into the filesystem. In this case, the `t_state` field stores the value `T_FINISHED`.

Incomplete

At least one log record included in the transaction has not yet been physically written to the journal, or new log records are still being added to the transaction. In case of system failure, the image of the transaction stored in the journal is likely not up-to-date. Therefore, when recovering from a system failure, *e2fsck* does not trust the incomplete transactions in the journal and skips them. In this case, the `t_state` field stores one of the following values:

`T_RUNNING`

Still accepting new atomic operation handles.

`T_LOCKED`

Not accepting new atomic operation handles, but some of them are still unfinished.

`T_FLUSH`

All atomic operation handles have finished, but some log records are still being written to the journal.

`T_COMMIT`

All log records of the atomic operation handles have been written to disk, but the transaction has yet to be marked as completed on the journal.

At any time the journal may include several transactions, but only one of them is in the `T_RUNNING` state it is the *active transaction* that is accepting the new atomic operation handle requests issued by the Ext3 filesystem.

Several transactions in the journal might be incomplete, because the buffers containing the relative log records have not yet been written to the journal.

If a transaction is complete, all its log records have been written to the journal but some of the corresponding buffers have yet to be written onto the filesystem. A complete transaction is deleted from the journal when the JBD layer verifies that all buffers described by the log records have been successfully written onto the Ext3 filesystem.

18.7.4. How Journaling Works

Let's try to explain how journaling works with an example: the Ext3 filesystem layer receives a request to write some data blocks of a regular file.

As you might easily guess, we are not going to describe in detail every single operation of the Ext3 filesystem layer and of the JBD layer. There would be far too many issues to be covered! However, we describe the essential actions:

1. The service routine of the `write( )` system call triggers the `write` method of the file object associated with the Ext3 regular file. For Ext3, this method is implemented by the `generic_file_write( )` function, already described in the section "Writing to a File" in Chapter 16.
2. The `generic_file_write( )` function invokes the `prepare_write` method of the `address_space` object several times, once for every page of data involved by the write operation. For Ext3, this method is implemented by the `ext3_prepare_write( )` function.
3. The `ext3_prepare_write( )` function starts a new atomic operation by invoking the `journal_start( )` JBD function. The handle is added to the active transaction. Actually, the atomic operation handle is created only when executing the first invocation of the `journal_start( )` function. Following invocations verify that the `journal_info` field of the process descriptor is already set and use the referenced handle.
4. The `ext3_prepare_write( )` function invokes the `block_prepare_write( )` function already described in Chapter 16, passing to it the address of the `ext3_get_block( )` function. Remember that `block_prepare_write( )` takes care of preparing the buffers and the buffer heads of the file's page.
5. When the kernel must determine the logical number of a block of the Ext3 filesystem, it executes the `ext3_get_block( )` function. This function is actually similar to `ext2_get_block( )`, which is described in the earlier section "Allocating a Data Block." A crucial difference, however, is that the Ext3 filesystem invokes functions of the JBD layer to ensure that the low-level operations are logged:
    - *Before* issuing a low-level write operation on a metadata block of the filesystem, the function invokes `journal_get_write_access( )`. Basically, this latter function adds the metadata buffer to a list of the active transaction. However, it must also check whether the metadata is included in an older incomplete transaction of the journal; in this case, it duplicates the buffer to make sure that the older transactions are committed with the old content.
    - *After* updating the buffer containing the metadata block, the Ext3 filesystem invokes `journal_dirty_metadata( )` to move the metadata buffer to the proper dirty list of the active transaction and to log the operation in the journal.

   Notice that metadata buffers handled by the JBD layer are not usually included in the dirty lists of buffers of the inode, so they are not written to disk by the normal disk cache flushing mechanisms described in Chapter 15.

6. If the Ext3 filesystem has been mounted in "journal" mode, the `ext3_prepare_write( )` function also invokes `journal_get_write_access( )` on every buffer touched by the write operation.
7. Control returns to the `generic_file_write( )` function, which updates the page with the data stored in the User Mode address space and then invokes the `commit_write` method of the `address_space` object. For Ext3, the function

that implements this method depends on how the Ext3 filesystem has been mounted:

   - If the Ext3 filesystem has been mounted in "journal" mode, the `commit_write` method is implemented by the `ext3_journalled_commit_write( )` function, which invokes `journal_dirty_metadata( )` on every buffer of data (not metadata) in the page. This way, the buffer is included in the proper dirty list of the active transaction and not in the dirty list of the owner inode; moreover, the corresponding log records are written to the journal. Finally, `ext3_journalled_commit_write( )` invokes `journal_stop( )` to notify the JBD layer that the atomic operation handle is closed.
   - If the Ext3 filesystem has been mounted in "ordered" mode, the `commit_write` method is implemented by the `ext3_ordered_commit_write( )` function, which invokes the `journal_dirty_data( )` function on every buffer of data in the page to insert the buffer in a proper list of the active transactions. The JBD layer ensures that all buffers in this list are written to disk before the metadata buffers of the transaction. No log record is written onto the journal. Next, `ext3_ordered_commit_write( )` executes the normal `generic_commit_write( )` function described in Chapter 15, which inserts the data buffers in the list of the dirty buffers of the owner inode. Finally, `ext3_ordered_commit_write( )` invokes `journal_stop( )` to notify the JBD layer that the atomic operation handle is closed.
   - If the Ext3 filesystem has been mounted in "writeback" mode, the `commit_write` method is implemented by the `ext3_writeback_commit_write( )` function, which executes the normal `generic_commit_write( )` function described in Chapter 15, which inserts the data buffers in the list of the dirty buffers of the owner inode. Then, `ext3_writeback_commit_write( )` invokes `journal_stop( )` to notify the JBD layer that the atomic operation handle is closed.

8. The service routine of the `write( )` system call terminates here. However, the JBD layer has not finished its work. Eventually, our transaction becomes complete when all its log records have been physically written to the journal. Then `journal_commit_transaction( )` is executed.
9. If the Ext3 filesystem has been mounted in "ordered" mode, the `journal_commit_transaction( )` function activates the I/O data transfers for all data buffers included in the list of the transaction and waits until all data transfers terminate.
10. The `journal_commit_transaction( )` function activates the I/O data transfers for all metadata buffers included in the transaction (and also for all data buffers, if Ext3 was mounted in "journal" mode).
11. Periodically, the kernel activates a checkpoint activity for every complete transaction in the journal. The checkpoint basically involves verifying whether the I/O data transfers triggered by `journal_commit_transaction( )` have successfully terminated. If so, the transaction can be deleted from the journal.

Of course, the log records in the journal never play an active role until a system failure occurs. Only during system reboot does the *e2fsck* utility program scan the journal stored in the filesystem and reschedule all write operations described by the log records of the complete transactions.