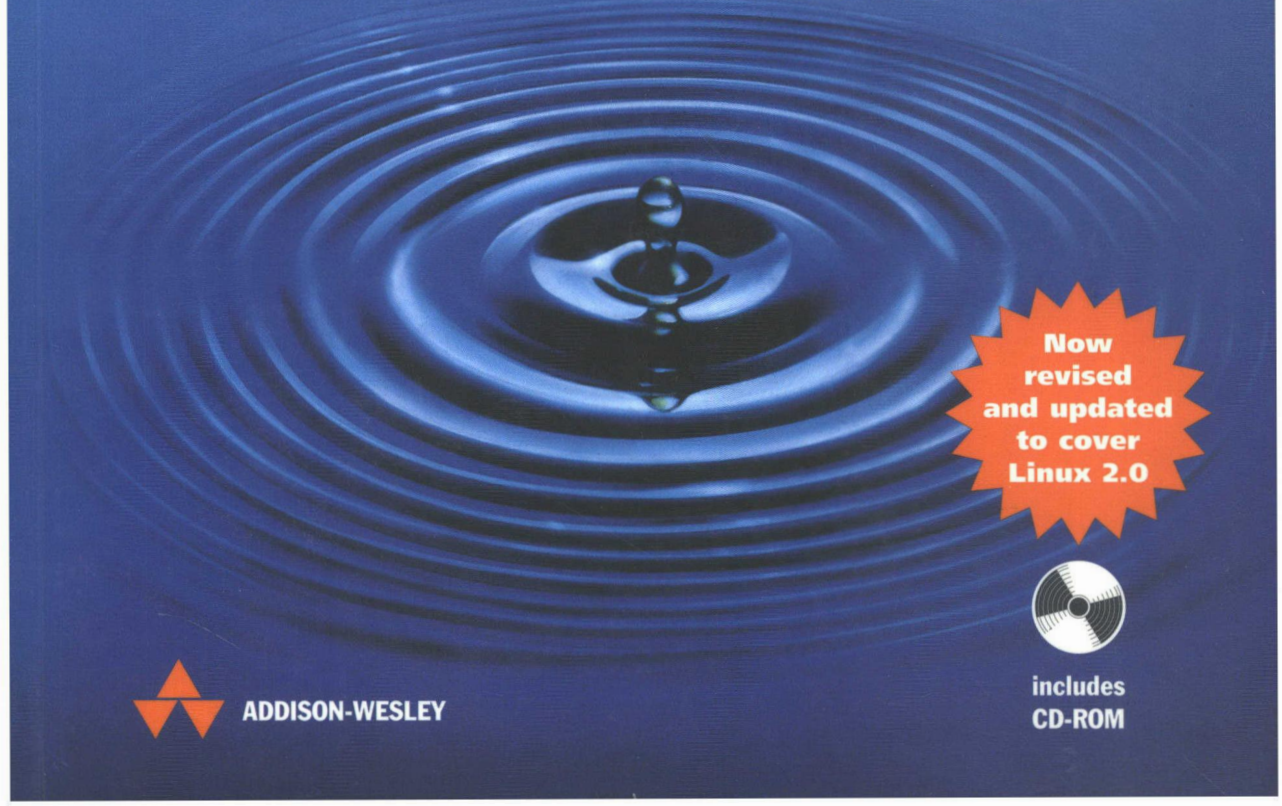
M BECK, H BÖHME, M DZIADZKA,
U KUNITZ, R MAGNUS, D VERWORNER

LINUX KERNEL INTERNALS

SECOND EDITION



Now revised and updated to cover Linux 2.0

ADDISON-WESLEY

includes CD-ROM

BECK
BÖHME
DZIADZKA
KUNITZ
MAGNUS
VERWORNER

LINUX KERNEL INTERNALS

SECONI
EDITIO

ADDISON
WESLEY

33143

© Addison Wesley Longman 1998

Translated from the German edition *Linux-Kernel-Programmierung* published by Addison-Wesley (Deutschland) GmbH.

Addison Wesley Longman Limited
Edinburgh Gate
Harlow
Essex
CM20 2JE
England
and Associated Companies throughout the World

The rights of Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus and Dirk Verworner to be identified as authors of this Work have been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a licence permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1P 9HE.

The programs in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations nor does it accept any liabilities with respect to the programs.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Addison Wesley Longman Limited has made every attempt to supply trademark information about manufacturers and their products mentioned in this book. A list of the trademark designations and their owners appears below.

Translated and typeset by 46
Cover designed by Senate
Printed and bound in the United States of America.

First printed 1997

ISBN 0–201–33143–8

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

Trademark Notice
Alpha™ and VAX™ are trademarks of Digital Equipment Corporation
Intel 386™, 486™, Pentium™ and Pentium Pro™ are trademarks of Intel Corporation
Logitech™ and MouseMan™ are trademarks of Logitech
NFS™ (Network File System) and Solaris™ are trademarks of Sun Microsystems, Inc.
OS/2™ and PowerPC™ are trademarks of International Business Machines Corporation
SCO UNIX™ is a trademark of SCO
SPARC™ is a trademark of Sparc International Inc.
UNIX™ and UNIX System V™ are trademarks, licensed through X/Open™ Company Ltd
Windows™, Windows NT™, Windows 95™ and MS-DOS™ are trademarks of Microsoft Corporation

6 The LINUX file system

"My dear Watson, coming by good information is not difficult.
What is far more difficult is finding it again."

Conan Doyle, The Adventures of Sherlock Holmes

6.1 Basic principles
6.2 The representation of file systems in the kernel
6.3 The *Proc* file system
6.4 The *Ext2* file system

In the PC field, variety in file systems is common; practically every operating system has its own file system, and each of these naturally claims to be 'faster, better and more secure' than its predecessors.

The large number of file systems supported by LINUX is undoubtedly one of the main reasons why LINUX has gained acceptance so quickly in its short history. Not every user is in a position to put in the time and effort to convert his/her old data to a new file system.

The range of file systems supported is made possible by the unified interface to the LINUX kernel. This is the *Virtual File System Switch* (VFS), which will be referred to below simply as the 'Virtual File System', although we are not dealing with a file system so much as an interface providing a clearly defined link between the operating system kernel and the different file systems (as illustrated in Figure 6.1).

The Virtual File System supplies the applications with the system calls for file management (*see* Section A.2), maintains internal structures and passes tasks on to the appropriate actual file system. Another important job of the VFS is performing standard actions. As a rule, for instance, no file system implementation will actually provide an `lseek()` function, as the functions of `lseek()` are provided by a standard action of the VFS. We are therefore justified in calling VFS a file system.

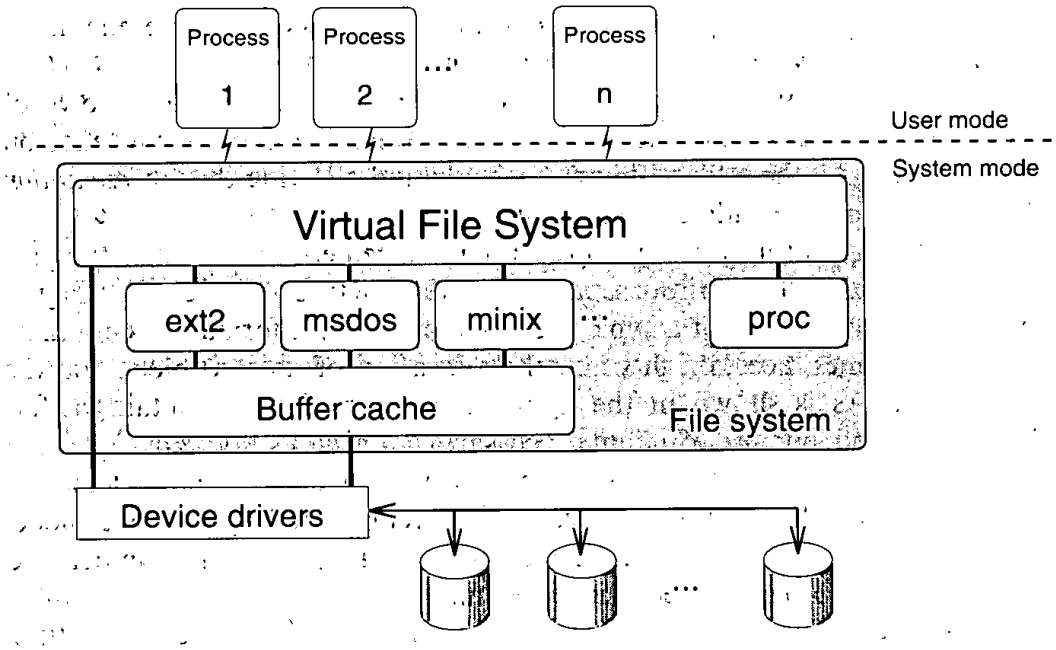


Figure 6.1 The layers in the file system.

In this chapter, we take a closer look at how VFS works and how it interacts with specific file system implementations. As a simple example, the implementation of the *Proc* file system will be considered. In addition, we examine the design and structure of the *Ext2* file system serving as the standard LINUX file system.

6.1 Basic principles

The importance of a good file management system is often underestimated. Where human beings can use their memory or a notebook, a computer has to resort to other means.

A central demand made of a file system is the *purposeful structuring* of data. When selecting a purposeful structure, however, two factors not to be neglected are the *speed* of access to data and a facility for *random access*.

Random access is made possible by block-oriented devices, which are divided into a specific number of equal-sized blocks. When using these, LINUX also has at its disposal the buffer cache described in Section 4.3. Using the functions of the buffer cache, it is possible to access any of the sequentially numbered blocks in a given device. The file system itself must be capable of ensuring unique allocation of the data to the hardware blocks.

In UNIX, the data are stored in a hierarchical file system containing files of different types. These comprise not only regular files and directories but also

device files, FIFOs (*named pipes*), symbolic links and sockets. These enable all the resources of the system to be accessed via files.

From a programming point of view, files are simply data flows of unspecified content containing no further structuring. The file system takes on the task of managing these 'data flows' efficiently and allowing the *representation of different file types* (including pseudo-files).

In UNIX, the information required for management is kept strictly apart from the data and collected in a separate inode structure for each file. Figure 6.2 shows the arrangement of a typical UNIX inode. The information contained includes access times, access rights and the allocation of data to blocks on the physical media. As is shown in the figure, the inode already contains a few block numbers to ensure efficient access to small files (which are often encountered under UNIX). Access to larger files is provided via indirect blocks, which also contain block numbers. Every file is represented by just one inode, which means that, within a file system, each inode has a unique number and the file itself can also be accessed using this number.

Directories allow the file system to be given a hierarchical structure. These are also implemented as files, but the kernel assumes them to contain pairs consisting of a filename and its inode number. There is no reason why a file cannot be accessed via a number of names, which can even be held in

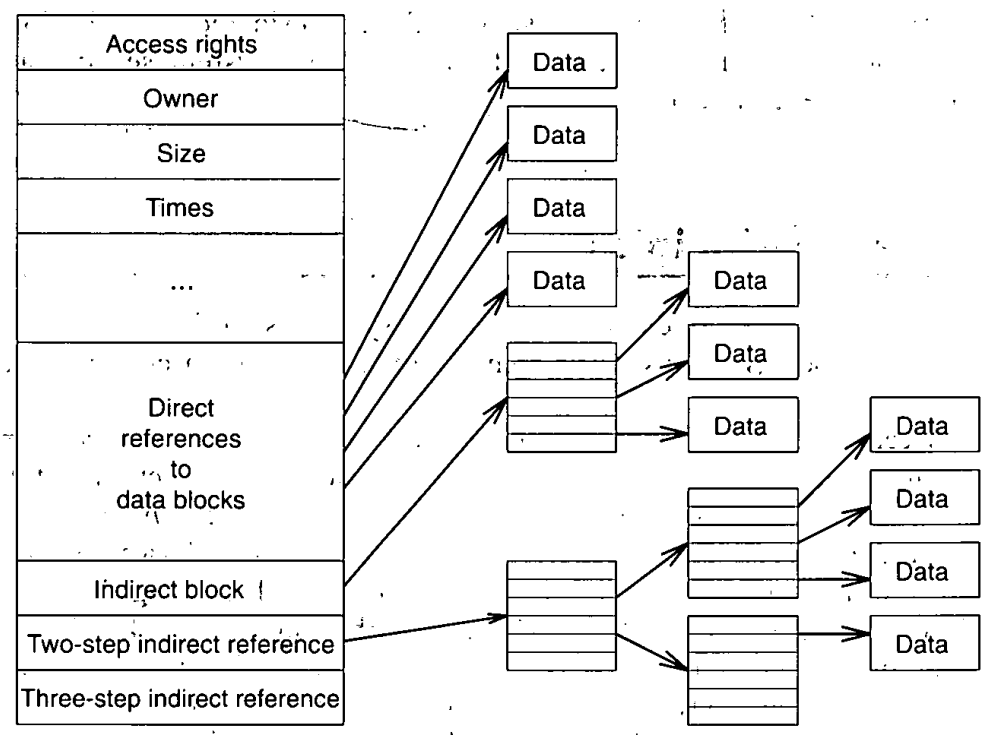


Figure 6.2 Structure of a UNIX inode.

different directories (in the form of a *hard link*). In older versions of UNIX it was still possible to modify directory files using a simple editor, but to ensure consistency this is no longer permitted in recent versions. LINUX file systems will not even allow these to be read with normal system calls.

The basic structure is the same for all the different UNIX file systems (*see* Figure 6.3). Each file system starts with a *boot block*. This block is reserved for the code required to boot the operating system (*see* Appendix D). As file systems should usually be able to exist on any block-oriented device, and on each device, in principle, they will always have the same structure (to ensure uniformity), the boot block will be present whether or not the computer is booted from the device in question.

All the information which is essential for managing the file system is held in the *superblock*. This is followed by a number of *inode blocks* containing the inode structures for the file system. The remaining blocks for the device provide the space for the data. These *data blocks* thus contain ordinary files along with the directory entries and the indirect blocks.

As file systems must be able to be implemented on different devices, the implementation of the file system must also adapt to different device-level characteristics, such as block size, and so on. At the same time, all operating systems aim for *device independence*, which will make it immaterial what media the data have been stored on. In LINUX, this task is handled by the respective file system implementation, enabling the Virtual File System to work with device-independent structures.

In UNIX, the separate file systems are not accessed via device identifiers (such as drive numbers), as is the case for other operating systems, but combined into a hierarchical directory tree.

This arrangement is built up by the action of *mounting* the file system, which adds another file system (of whatever type) to an existing directory tree. A new file system can be mounted onto any directory. This original directory is then known as the *mount point* and is covered up by the root directory of the new file system along with its subdirectories and files. *Unmounting* the file system releases the hidden directory structure again.

A further aspect of major importance to the quality of a file system is *data security*. On the one hand, this comprises facilities to maintain consistency and mechanisms ensuring data protection. On the other hand, the file system should behave robustly in the event of system faults, corruption of data or program crashes.

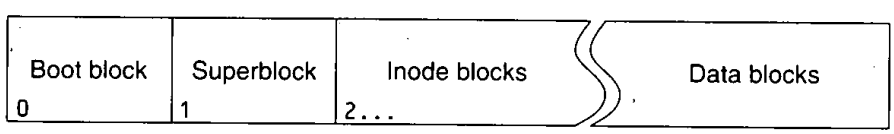


Figure 6.3 Schematic structure of a UNIX file system.

6.2 The representation of file systems in the kernel

The representation of data on a floppy disk or hard disk may differ considerably from case to case. In the end, however, the actual representation of data in LINUX's memory works out the same. Here, once again, LINUX sticks closely to its 'model', UNIX, because the management structures for the file systems are very similar to the logical structure of a UNIX file system.

These are the responsibility of the VFS, which calls the file-system-specific functions for the various implementations to fill up the structures. These functions are provided by every actual implementation and made known to the VFS via the function `register_filesystem()`.

```
#ifdef CONFIG_MINIX_FS
   register_filesystem(&(struct file_system_type)
      {minix_read_super, "minix", 1, NULL});
#endif
```

By this means the VFS is given the name of the file system ('`minix`'), a function of the implementation and a flag indicating whether a device is strictly necessary to mount the file system from. The function passed, `read_super()`, forms the mount interface: it is only via this function that further functions of the file system implementation will be made known to the VFS.

The function sets up the `file_system_type` structure it has been passed in a singly linked list whose beginning is pointed to by `file_systems`.

```
struct file_system_type {
   struct super_block *(*read_super) (struct super_block *,
                                      void *, int);
   char *name;
   int requires_dev;
   struct file_system_type *next;
}     *file_systems = NULL;
```

In older LINUX kernels (before version 1.1.8) the structures were still managed in a static table, as all the file system implementations were known when the kernel was compiled. With the introduction of modules it became desirable to load new file systems after the LINUX system had started running.

Once a file system implementation has been registered with the VFS, file systems of this type can be administered.

6.2.1 Mounting

Before a file can be accessed, the file system containing the file must be mounted. This can be done using either the system call *mount* or the function `mount_root()`.

The `mount_root()` function takes care of mounting the first file system (the *root file system*). It is called by the system call *setup* (*see* page 378) after all the file system implementations permanently included in the kernel have been registered. The *setup* call itself is called just once,[1] immediately after the init process is created by the kernel function `init()` (file `init/main.c`). This system call is necessary because access to kernel structures is not allowed from user mode (which is the status of the init process).

Every mounted file system is represented by a `super_block` structure. These structures are held in the static table `super_blocks[]` and limited in number to `NR_SUPER`.

The superblock is initialized by the function `read_super()` in the Virtual File System. It interrogates floppy disks and CD-ROM for a change of media, tests whether the superblock is already present and, if so, returns it. If it finds no superblock in existence it searches for a new entry in the superblock table and calls the function to create a superblock which is provided by every file system implementation. This file-system-specific function will have been made known on registering the implementation with the VFS. When called, it will contain:

- a superblock structure in which the elements `s_dev` and `s_flags` are filled in accordance with Table 6.1,
- a character string (in this case `void *`) containing further mount options for the file system, and
- a `silent` flag indicating whether unsuccessful mounting should be reported. This flag is used only by the kernel function `mount_root()`, as this calls all the `read_super()` functions present in the various file system implementations in turn when mounting the root file system, and constant error messages during start-up would be disruptive.

The file-system-specific function `read_super()` reads its data if necessary from the appropriate block device using the LINUX cache functions introduced in Section 4.3. This also provides the reason why a process is required to mount file systems: the process can be halted by the device driver (using the sleep/

[1] Nothing would be achieved by calling it a second time, as the system call *setup* returns an error after it has been used once.

Table 6.1 The file-system-independent mount flags in the superblock.

Macro	Value	Remarks
MS_RDONLY	1	File system is read only
MS_NOSUID	2	Ignores S bits
MS_NODEV	4	Inhibits access to device files
MS_NOEXEC	8	Inhibits execution of program
MS_SYNCHRONOUS	16	Immediate write to disk
MS_REMOUNT	32	Changes flags

wake-up mechanism; *see* Section 3.1.5), since it requires time to access the device. The LINUX superblock is structured as follows:

```
struct super_block {
   dev_t s_dev;                    /* device for file system */
   unsigned long s_blocksize;      /* block size             */
   unsigned char s_blocksize_bits; /* ld (block size)        */
   unsigned char s_lock;           /* superblock lock        */
   unsigned char s_rd_only;        /* not used (= 0)         */
   unsigned char s_dirt;           /* superblock changed     */
   struct file_system_type *s_type;/* file system type       */
   struct super_operations *s_op;  /* superblock operations  */
   unsigned long s_flags;          /* flags                  */
   unsigned long s_magic;          /* file system identifier */
   unsigned long s_time;           /* time of change         */
   struct inode * s_covered;       /* mount point            */
   struct inode * s_mounted;       /* root inode             */
   struct wait_queue * s_wait;     /* s_lock wait queue      */
   union {
      struct minix_sb_info minix_sb;
                  ...
      void *generic_sdp;
   } u;                    /* file-system-specific information */
};
```

The superblock contains information on the entire file system, such as block size, access rights and time of the last change. In addition, the union `u` at the end of the structure holds special information on the relevant file systems. For file system modules mounted later, there is a pointer `generic_sdp`.

The components `s_lock` and `s_wait` ensure access to the superblock is synchronized. This uses the functions `lock_super()` and `unlock_super()`, which are defined in the file `<linux/locks.h>`.

```
extern inline void lock_super(struct super_block * sb)
{
   if (sb->s_lock)
      __wait_on_super(sb);
   sb->s_lock = 1;
}

extern inline void unlock_super(struct super_block * sb)
{
   sb->s_lock = 0;
   wake_up(&sb->s_wait);
}
```

The superblock also holds references to the file system's *root inode* `s_mounted` and the mount point `s_covered`.

Another task performed by the `read_super()` function in the actual file system implementation involves making the file system's root inode available and entering it in the superblock. This can be carried out using the functions of the VFS, such as the `iget()` function, which will be described later, provided the components `s_dev` and `s_op` are set correctly.

6.2.2 Superblock operations

The superblock structure provides, in the function vector `s_op`, functions for accessing the file system, and these form the basis for further work on the file system.

```
struct super_operations {
   void (*read_inode) (struct inode *);
   int  (*notify_change) (struct inode *, struct iattr *);
   void (*write_inode) (struct inode *);
   void (*put_inode) (struct inode *);
   void (*put_super) (struct super_block *);
   void (*write_super) (struct super_block *);
   void (*statfs) (struct super_block *, struct statfs *);
   int  (*remount_fs) (struct super_block *, int *, char *);
};
```

The functions in the `super_operations` structure serve to read and write an individual inode, to write the superblock and to read file system information. This means that the superblock operations contain functions to transfer the specific representation of the superblock and inode on the data media to their general form in memory and vice versa. As a result, this layer completely hides the actual representations. Strictly speaking, the inodes and the superblock do

not even have to exist. An example of this is the MS-DOS file system, in which the FAT and the information in the boot block are transferred to the UNIX-internal view consisting of the superblock and inodes. If a superblock operation is not implemented – that is, if the pointer to the operation is NULL – no further action will take place.

- **write_super(sb)**
  The write_super(sb) function is used to save the information of the superblock. This need not necessarily guarantee the consistency of the file system.[2] If the current file system supports a flag indicating inconsistency (*valid flag*), this should be set. In normal cases the function will cause the cache to write back the buffer for the superblock: this is ensured by setting the buffer's b_dirt flag. The function is used in synchronizing the device and is ignored by read-only file systems such as *Isofs*.
- **put_super(sb)**
  The Virtual File System calls this function when unmounting file systems, when it should also release the superblock and other information buffers (*see* brelse() in Section 4.3) and/or restore the consistency of the file system, leaving the valid flag correctly set. In addition, the s_dev entry in the superblock structure must be set to 0 to ensure that the superblock is once again available after unmounting.
- **statfs(sb, statfsbuf)**
  The two system calls *statfs* and *fstatfs* (*see* page 364) call the superblock operation which in fact does no more than fill in the statfs structure. This structure provides information on the file system, the number of free blocks and the preferred block size. Note that the structure is located in the user address space. If the operation fails, the VFS returns the error ENOSYS.
- **remount_fs(sb, flags, options)**
  The remount_fs() function changes the status of a file system (*see* Table 6.1). This generally only involves entering the new attributes for the file system in the superblock and restoring the consistency of the file system.
- **read_inode(inode)**
  This function is responsible for filling in the inode structure it has been passed, in a similar way to read_super(). It is called by the function __iget(), which will already have given the entries i_dev, i_ino, i_sb and i_flags their contents. The main purpose of the read_inode() function is to mark the different file types by entering inode operations in the inode according to the file type. Almost every read_inode function

[2] The data and inode blocks need not be written back, nor the lists and bitmaps for the free blocks, which means that the file system may not be consistent.

(the example is taken from the *Ext2* file system) will therefore contain the following lines:

```
if (S_ISREG(inode->i_mode))
   inode->i_op = &ext2_file_inode_operations;
else if (S_ISDIR(inode->i_mode))
   inode->i_op = &ext2_dir_inode_operations;
else if (S_ISLNK(inode->i_mode))
   inode->i_op = &ext2_symlink_inode_operations;
else if (S_ISCHR(inode->i_mode))
   inode->i_op = &chrdev_inode_operations;
else if (S_ISBLK(inode->i_mode))
   inode->i_op = &blkdev_inode_operations;
else if (S_ISFIFO(inode->i_mode))
   init_fifo(inode);
```

- **`notify_change(inode, attr)`**
  The changes made to the inode via system calls are acknowledged by `notify_change()`. This operation is missing from a number of file system implementations, but is of interest for the NFS, for example, as this file system has, so to speak, a local and an external inode. All inode changes are carried out on the local inode structure only, which means that the computer exporting the file system needs to be informed. This is done using the structure `iattr`:

```
struct iattr {
   unsigned int   ia_valid; /* flags for changed components */
   umode_t        ia_mode;  /* new access rights            */
   uid_t          ia_uid;   /* new user                     */
   gid_t          ia_gid;   /* new group                    */
   off_t          ia_size;  /* new size                     */
   time_t         ia_atime; /* time of last access          */
   time_t         ia_mtime; /* time of last modification    */
   time_t         ia_ctime; /* time of creation             */
};
```

  The functions calling `notify_change` and the flags passed in the component `ia_valid` are listed in Table 6.2.

- **`write_inode(inode)`**
  This function saves the inode structure, analogous to `write_super()`.

- **`put_inode(inode)`**
  This function is called by `iput()` if the inode is no longer required. Its main task is to delete the file physically and release its blocks if `i_nlink` is zero.

Table 6.2 The flags for `notify_change`.

Kernel function	ATTR_MODE	ATTR_UID	ATTR_GID	ATTR_SIZE	ATTR_ATIME	ATTR_MTIME	ATTR_CTIME	ATTR_ATIME_SET	ATTR_MTIME_SET
sys_chmod()	×						×		
sys_fchmod()	×						×		
sys_chown()	×	×	×				×		
sys_fchown()	×	×	×				×		
sys_truncate()				×		×	×		
sys_ftruncate()				×		×	×		
sys_write()	×								
open_namei()				×					
sys_utime()					×	×	×	×	×

6.2.3 The inode

When a file system is mounted, the superblock is generated and the root inode for the file system is entered in the component `i_mount` at the appropriate mount point, that is, in its inode structure. The definition of the `inode` structure is as follows:

```
struct inode {
    dev_t          i_dev;     /* file device number          */
    unsigned long  i_ino;     /* inode number                */
    umode_t        i_mode;    /* file type and access rights */
    nlink_t        i_nlink;   /* number of hard links        */
    uid_t          i_uid;     /* owner                       */
    gid_t          i_gid;     /* owner                       */
    dev_t          i_rdev;    /* device, if device file      */
    off_t          i_size;    /* size                        */
    time_t         i_atime;   /* time of last access         */
    time_t         i_mtime;   /* time of last modification   */
    time_t         i_ctime;   /* time of creation            */
    unsigned long  i_blksize; /* block size                  */
    unsigned long  i_blocks;  /* number of blocks            */
    unsigned long  i_version; /* DCache version management   */
    struct semaphore i_sem;          /* access control       */
    struct inode_operations * i_op;  /* inode operations     */
    struct super_block * i_sb;       /* superblock           */
```

```
    struct wait_queue * i_wait;      /* wait queue              */
    struct file_lock * i_flock;      /* file locks              */
    struct vm_area_struct * i_mmap;  /* memory areas            */
    struct inode * i_next, * i_prev; /* inode linking           */
    struct inode * i_hash_next, * i_hash_prev;
    struct inode * i_bound_to, * i_bound_by;
    struct inode * i_mount;          /* mounted inode           */
    struct socket * i_socket;   /* socket management            */
    unsigned short i_count;     /* reference counter            */
    unsigned short i_wcount;    /* number authorized to write */
    unsigned short i_flags;     /* flags (= i_sb->s_flags)      */
    unsigned char i_lock;       /* lock                         */
    unsigned char i_dirt;       /* inode has been modified      */
    unsigned char i_pipe;       /* inode represents pipe        */
    unsigned char i_sock;       /* inode represents socket      */
    unsigned char i_seek;       /* not used                     */
    unsigned char i_update;     /* inode is current             */
    union {
       struct pipe_inode_info pipe_i;
       struct minix_inode_info minix_i;
                ...
       void *generic_ip;
    } u;                  /* file-system-specific information */
};
```

In the first section, this holds information on the file. The remainder contains management information and the file-system-dependent union u.

In memory, the inodes are managed in two ways. First, they are managed in a doubly linked circular list starting with `first_inode`, which is accessed via the entries `i_next` and `i_prev`. The complete list of inodes is scanned through in the following way:

```
struct inode * inode, * next;

next = first_inode;
for(i = nr_inodes ; i > 0 ; i--) {
   inode = next;
   next = inode->i_next;
     ...
}
```

This approach is not particularly efficient, as the complete list of inodes also includes the 'free', unused inodes, for which the components `i_count`, `i_dirt` and `i_lock` should all be zero. The unused inodes are generated via the

grow_inodes() function, which is called every time that less than a quarter of all the inodes are free but not more than NR_INODE are in existence. The number of unused inodes and the count of all available inodes are held in the static variables nr_free and nr_inode respectively.

For fast access, inodes are also stored in an open hash table hash_table[], where collisions are dealt with via a doubly linked list using the components i_hash_next and i_hash_prev. Access to any of the NR_IHASH entries is made through the device and inode numbers.

The functions for working with inodes are iget(), namei() and iput().

```
inline struct inode *iget(struct super_block *sb, int nr)
{
   return __iget(sb, nr, 1);
}

struct inode *__iget(struct super_block * sb, int nr,
                     int crossmntp);

void iput(struct inode * inode);
```

The iget() function supplies the inode specified by the superblock sb and the inode number nr. In its turn it calls the __iget() function, which is instructed via a further parameter, crossmntp, to resolve mount points as well – that is, if the requested inode is a mount point it supplies the corresponding root inode for the mounted file system.

If the required inode is included in the hash table, the i_count reference counter is simply incremented. If it is not found, a 'free' inode is selected (get_empty_inode()) and the implementation of the relevant file system calls the superblock operation read_inode() to fill it with information. The resulting inode is then added to the hash table.

An inode obtained using iget() has to be released using the function iput(). This decrements the reference counter by 1 and marks the inode structure as 'free' if the former is 0.

Other functions to supply inodes are:

```
int namei(const char * pathname, struct inode ** res_inode);
int lnamei(const char * pathname, struct inode ** res_inode);
```

The filename pathname that has been passed is resolved and the address of the inode structure is stored in res_inode. The lnamei() function differs from namei() in that lnamei() does not resolve a symbolic link and will continue to supply the inode for the symbolic link. Both functions call _namei(). Additional parameters passed to this function are the inode of the base directory for the resolving procedures and a flag indicating whether symbolic links are to be resolved using follow_link.

The actual functioning of `_namei()` derives from `dir_namei()`. This function supplies the inode for the directory that contains the file with the name specified. All functions return an error code smaller than 0 if they are not successful.

6.2.4 Inode operations

The inode structure also has its own operations, which are held in the `inode_operations` structure and mainly provide for file management. These functions are usually called directly from the implementations of the appropriate system calls. Note that in all cases the functions must use `iput()` to release the inodes that have been passed to them, since the inode reference counter is incremented by 1 before the functions are called, to indicate that they are in use. If an inode operation fails, the calling function performs default actions; however, this often comprises the mere generation of an error.

```
struct inode_operations {
   struct file_operations * default_file_ops;
   int (*create) (struct inode *,const char *,int,int,
                  struct inode **);
   int (*lookup) (struct inode *,const char *,int,
                  struct inode **);
   int (*link) (struct inode *,struct inode *,const char *,int);
   int (*unlink) (struct inode *,const char *,int);
   int (*symlink) (struct inode *,const char *,int,const char *);
   int (*mkdir) (struct inode *,const char *,int,int);
   int (*rmdir) (struct inode *,const char *,int);
   int (*mknod) (struct inode *,const char *,int,int,int);
   int (*rename) (struct inode *,const char *,int,struct inode *,
                  const char *,int);
   int (*readlink) (struct inode *,char *,int);
   int (*follow_link) (struct inode *,struct inode *,int,int,
                       struct inode **);
   int (*bmap) (struct inode *,int);
   void (*truncate) (struct inode *);
   int (*permission) (struct inode *, int);
   int (*smap) (struct inode *, int);
};
```

- **`create(dir, name, len, mode, res_inode)`**
  This function is called from within the VFS function `open_namei()`. It performs a number of tasks. First, it extracts a free inode from the complete list of inodes with the aid of the `get_empty_inode()` function.

The inode structure now needs to be filled with file-system-specific data, for which, for example, a free inode on the media is sought out. After this, `create()` enters the filename `name` of length `len` in the directory specified by the inode `dir`. If `create()` is not present in a file system implementation, the VFS returns the error `EACCESS`.

- **`lookup(dir, name, len, res_inode)`**
  This function is supplied with a filename and its length and returns the inode for the file in the argument `res_inode`. This is carried out by scanning the directory specified by the inode `dir`. The `lookup()` function must be defined for directories, otherwise the VFS will return the error `ENOTDIR`.

  The calling VFS function `lookup()` performs a special procedure for the name '..'. If the process is already in its root directory, the root inode is returned. However, if the root inode for a mounted file system is overstepped by '..', the VFS function uses the 'hidden' inode to call the inode operation.

- **`link(oldinode, dir, name, len)`**
  This function sets up a hard link. The file `oldinode` will be linked under the stated name and the associated length in the directory specified by the inode `dir`. Before `link()` is called, a check is made that the inodes `dir` and `oldinode` are on the same device and that the current process is authorized to write to `dir`. If this function is missing, the calling function in the VFS returns the error `EPERM`.

- **`unlink(dir, name, len)`**
  This function deletes the specified file in the directory specified by the inode `dir`. The calling function first confirms that this operation possesses the relevant permissions. If `unlink()` is not implemented, the VFS returns the error `EPERM`.

- **`symlink(dir, name, len, symname)`**
  This function sets up the symbolic link `name` in the directory `dir`, with `len` giving the length of the name `name`. The symbolic link points to the path `symname`. Before this function is called by the VFS, the access permissions will have been checked by a call to `permission()`. If `symlink()` is not present in a specific implementation, the VFS returns the error `EPERM`.

- **`mkdir(dir, name, len, mode)`**
  This function sets up a subdirectory with the name `name` and the access rights `mode` in the directory `dir`. The `mkdir()` function first has to check whether further subdirectories are permitted in the directory, then allocate a free inode on the data media and a free block, to which the directory is then written together with its default entries '.' and '..'. The access rights will already have been checked in the calling VFS function. If the `mkdir()` function is not implemented, the error `EPERM` is returned.

- **rmdir(dir, name, len)**
  This function deletes the subdirectory name from the directory dir. The function first checks that the directory to be deleted is empty and whether it is currently being used by a process, as well as whether the process is the owner of the subdirectory if the sticky bit is set in the directory dir. As with the functions already described, the access rights are checked beforehand by a VFS function. If rmdir() is not available, the VFS returns the error EPERM.

- **mknod(dir, name, len, mode, rdev)**
  This function sets up a new inode in the mode mode. This inode will be given the name name in the directory dir. If the inode is a device file (in which case either S_ISBLK(mode) or S_ISCHR(mode) applies), the parameter rdev gives the number of the device. If this function is not implemented, the error EPERM is returned.

- **rename(odir, oname, olen, ndir, nname, nlen)**
  This function changes the name of a file. This involves removing the old name oname from the odir directory and entering the new name nname in ndir. The calling function checks the relevant access permissions in the directories beforehand, and a further check is made to ensure that the directories '.' and '..' do not appear as the source or destination of an operation. If the function is missing, an EPERM error is generated by the VFS.

- **readlink(inode, buf, size)**
  This function reads symbolic links and should copy into the buffer in the user address space the pathname for the file to which the link points. If the buffer is too small, the pathname should simply be truncated. If the inode is not a symbolic link, EINVAL should be returned. This function is called directly from sys_readlink() once the write access permission to the buffer buf has been checked and the inode has been found using lnamei(). If the implementation does not exist, the system call returns the error EINVAL.

- **follow_link(dir, inode, flag, mode, res_inode)**
  This function is used to resolve symbolic links. For the inode assigned to a symbolic link, this function returns the inode to which the link points in the argument res_inode. To avoid endless loops,[3] the maximum number of links to be resolved is set at 5 in LINUX. In the implementations, this count is 'hard-wired', and any implementations written should follow this example.

  If follow_link() is missing, the calling function of the same name in the VFS simply returns inode, as if the link were pointing to itself. This behaviour means that the VFS function can always be called

[3] A symbolic link can, after all, point to another symbolic link.

without testing whether the current inode describes a file or a symbolic link.

- **bmap(inode, block)**
  This function is called to enable memory mapping of files. In the argument `block` it is given the number of a logical data block in the file. This number must be converted by `bmap()` into the logical number of the block on the media. To do this, `bmap()` searches for the block in the actual implementation of the specified inode and returns its number. This may in some cases involve reading other blocks from the media. A return value of 0 signifies an error and is returned by the Virtual File System's calling function `bmap()` if a file-system-specific `bmap` function is not defined.

  This function is used by `generic_mmap()` to map a block from the file to an address in the user address space. If it cannot be found, executable files must first be loaded into memory completely, as the more efficient demand paging is not then available.

- **truncate(inode)**
  This function is mainly intended to shorten a file, but can also lengthen a file to any length if this is supported by the specific implementation. The only parameter required by `truncate()` is the inode of the file to be amended, with the `i_size` field set to the new length before the function is called. The `truncate()` function is used at a number of places in the kernel, both by the system call `sys_truncate()` and when a file is opened. It will also release the blocks no longer required by a file.

  Thus, the `truncate()` function can be used to delete a file physically if the inode on the media is cleared afterwards. Although the functioning of this function is very simple to describe, it can be very complicated to implement, as problems can arise with synchronization. If this function is not implemented, no error message is generated. In this case, as the `i_size` component has been set beforehand, the length of the file will only appear to have changed.

- **permission(inode, flag)**
  This function checks the inode to confirm the access rights to the file given by the mask. The possible values for the mask are `MAY_READ`, `MAY_WRITE` and `MAY_EXEC`. If the function is not available, the calling function in the Virtual File System checks the standard UNIX permissions, which means that implementation is actually unnecessary, unless additional access mechanisms are to be implemented.

- **smap(inode, sector)**
  This function is intended principally to allow swap files to be created on a UMSDOS file system. Like `bmap()`, this inode operation supplies the logical sector number (not block or cluster) on the media for the sector of the file specified. This means that the operation is only of interest for

the UMSDOS file system implementation, but is already provided by the MS-DOS file system in the form of the `msdos_smap()` function, where it is, however, only used internally. In the memory management function `rw_swap_page()`, the `smap()` function is required to prepare to work with a swap file if `bmap()` is not available.

6.2.5 The file structure

In a multi-tasking system the problem often arises that a number of processes wish to access a file at the same time, both to read and to write. Even a single process may be reading *and* writing at different points in the file. To avoid synchronization problems and allow shared access to files by different processes, UNIX has simply introduced an extra structure.

This relatively simple structure, `file`, contains information on a specific file's access rights `f_mode`, the current file position `f_pos`, the type of access `f_flags` and the number of accesses `f_count`.

```
struct file {
   mode_t          f_mode;          /* access type                 */
   loff_t          f_pos;           /* file position               */
   unsigned short  f_flags;         /* open() flags                */
   unsigned short  f_count;         /* reference counter           */
   off_t           f_reada;         /* read ahead flag             */
   struct file     *f_next,*f_prev;/* links                       */
   int             f_owner;         /* PID or -PGRP for SIGIO      */
   struct inode    *f_inode;        /* related inode               */
   struct file_operations * f_op; /* file operations             */
   unsigned long   f_version;       /* Dcache version management */
   void            *private_data;   /* needed for tty driver       */
};
```

The file structures are managed in a doubly linked circular list via the pointers `f_next` and `f_prev`. This *file table* can be accessed via the pointer `first_file`.

6.2.6 File operations

The `file_operations` structure is the general interface for work on files, and contains the functions to open, close, read and write files. The reason why these functions are not held in `inode_operations` but in a separate structure is that they need to make changes to the `file` structure.

The inode's `inode_operations` structure also includes the component `default_file_ops`, in which the standard file operations are already specified.

```
struct file_operations {
   int (*lseek) (struct inode *, struct file *, off_t, int);
   int (*read) (struct inode *, struct file *, char *, int);
   int (*write) (struct inode *, struct file *, char *, int);
   int (*readdir) (struct inode *, struct file *,
                   struct dirent *, int);
   int (*select) (struct inode *, struct file *, int,
                  select_table *);
   int (*ioctl) (struct inode *, struct file *,
                 unsigned int, unsigned long);
   int (*mmap) (struct inode *, struct file *,
                struct vm_area_struct *);
   int (*open) (struct inode *, struct file *);
   void (*release) (struct inode *, struct file *);
   int (*fsync) (struct inode *, struct file *);
   int (*fasync) (struct inode *, struct file *, int);
   int (*check_media_change) (dev_t);
   int (*revalidate) (dev_t);
};
```

These functions are also useful for sockets and device drivers, as they contain the actual functions for sockets and devices. The inode operations, on the other hand, only use the representation of the socket or device in the related file system or its copy in memory.

- **`lseek(inode, filp, offset, origin)`**
  The job of the `lseek` function is to deal with positioning within the file. If this function is not implemented, the default action simply converts the file position `f_pos` for the file structure if the positioning is to be carried out from the start or from the current position. If the file is represented by an inode, the default function can also be positioned from the end of the file. If the function is missing, the file position in the file structure is updated by the VFS.
- **`read(inode, filp, buf, count)`**
  This function copies `count` bytes from the file into the buffer `buf` in the user address space. Before calling the function, the Virtual File System first confirms that the entire buffer is located in the user address space and can be written to, and also that the file pointer is valid and the file has been opened to read. If no read function is implemented, the error `EINVAL` is returned.
- **`write(inode, filp, buf, count)`**
  The `write` function operates in an analogous manner to `read()` and copies data from the user address space to the file.

- **readdir(inode, filp, dirent, count)**
  This function returns the next directory entry in the dirent structure or an ENOTDIR or EBADF error. If this function is not implemented, the Virtual File System returns ENOTDIR.

- **select(inode, filp, type, wait)**
  This function checks whether data can be read from a file or written to one. An additional test for exception conditions can also be made. This function only serves a useful purpose for device drivers and sockets. The main task of the function is taken care of by the Virtual File System; thus, when interrogating files the VFS always returns the value 1 if it is a normal file, otherwise 0. Further consideration will be given to the select function in Section 7.4.6.

- **ioctl(inode, filp, cmd, arg)**
  Strictly speaking, the ioctl() function sets device-specific parameters. However, before the Virtual File System calls the ioctl operation, it tests the following default arguments:

  FIONCLEX Clears the close-on-exec bit.

  FIOCLEX Sets the close-on-exec bit.

  FIONBIO If the additional argument arg refers to a value not equal to zero, the O_NONBLOCK flag is set; otherwise it is cleared.

  FIOASYNC Sets or clears the O_SYNC flag as for FIONBIO. This flag is not at present evaluated.

  If cmd is not among these values, a check is performed on whether filp refers to a normal file. If so, the function file_ioctl() is called and the system call terminates. For other files, the VFS tests for the presence of an ioctl function. If there is none, the EINVAL error is returned, otherwise the file-specific ioctl function is called.
  The following commands are available to the file_ioctl() function:

  FIBMAP Expects in the argument arg a pointer to a block number and returns the logical number of this block in the file on the device if the inode relating to the file has a bmap function. This logical number is written back to the address arg. Absence of the inode operations or bmap() generates an EBADF or EINVAL error respectively.

  FIGETBSZ Returns the block size of the file system in which the file is located. It is written to the address arg if a superblock is assigned to the file. Otherwise, an EBADF error is generated.

  FIONREAD Writes the number of bytes within the file not yet read to the address arg.

As all of these commands write to the user address area, permission for this is always obtained via the function `verify_area()`, and an access error may be returned. If the command `cmd` is not among the values described, `file_ioctl()`, too, calls an existing file-specific `ioctl` function; otherwise the `EINVAL` error is returned.

- **`map(inode, filp, vm_area)`**
  This function maps part of a file to the user address space of the current process. The structure `vm_area` specified describes all the characteristics of the memory area to be mapped: the components `vm_start` and `vm_end` give the start and end addresses of the memory area to which the file is to be mapped and `vm_offset` the position in the file from which mapping is to be carried out. For a more comprehensive description of the `mmap` mechanism *see* Section 4.2.2.

- **`open(inode, filp)`**
  This function only serves a useful purpose for device drivers, as the standard function in the Virtual File System will already have taken care of all the necessary actions on regular files, such as allocating the file structure.

- **`release(inode, filp)`**
  This function is called when the file structure is released, that is, when its reference counter `f_count` is zero. This function is primarily intended for device drivers, and its absence will be ignored by the Virtual File System. Updating of the inode is also taken care of automatically by the Virtual File System.

- **`fsync(inode, filp)`**
  The `fsync()` function ensures that all buffers for the file have been updated and written back to the device, which means that the function is only relevant for file systems. If a file system has not implemented an `fsync()` function, `EINVAL` is returned.

- **`fasync(inode, filp, on)`**
  This function is called by the VFS when a process uses the system call *`fcntl`* to log on or off for asynchronous messaging by sending a `SIGIO` signal. The messaging will take place when data are received and the `on` flag is set. If `on` is not set, the process unregisters the file structure from asynchronous messaging. Absence of this function is ignored by the VFS. At present, only terminal drivers and socket handling implement a `fasync()` function.

- **`check_media_change(dev)`**
  This function is only relevant to block devices supporting changeable media. It tests whether there has been a change of media since the last operation on it. If so, the function will return a 1, otherwise a zero. The `check_media_change()` function is called by the VFS function `check_disk_change()`; if a change of media has taken place, it calls

put_super() to remove any superblock belonging to the device, discards all the buffers belonging to the device dev which are still in the buffer cache, along with all the inodes on this device, and then calls revalidate(). As check_disk_change() requires a considerable amount of time, it is only called when mounting a device. Its return values are the same as for check_media_change(). If it is not available, zero (that is, no change) is always returned.

- **revalidate(dev)**
  This function is called by the VFS after a media change has been recognized, to restore the consistency of a block device. It should establish and record all the necessary parameters of the media, such as the number of blocks, number of tracks and so on. If this function is missing, the VFS takes no further action.

6.2.7 Opening a file

One of the most important operations when accessing data is opening a file with the system call *open*. For this, the system not only has to make the appropriate preparations to ensure access to data without problems, but also has to check the authorizations for the process. This is also where the actual switching function of the Virtual File System is implemented, passing data between the specific file system implementations and the various devices.

Once it has been confirmed that the calling process is entitled to open files in the first place, a new file structure is requested via the function get_empty_filp() and entered in the file descriptor table for the process. In this structure, appropriate contents are entered in the fields f_flags and f_mode, and the open_namei() function is called to obtain the inode for the file to be opened.

Before this function is called, the open() flags are modified, leaving the two lowest bits holding the access permissions – bit 0 for read and bit 1 for write operations. The advantage of representing access to the file in this way is clear: it allows the access authorizations to be checked using a simple bit test.

The open_namei() function calls the function dir_namei() (mentioned earlier) and resolves the filename except for the base name of the file, obtaining the inode for the directory in which the file is located. The open_namei() function then performs a number of tests.

- If a filename ends in a slash (/), the dir_namei() function has already supplied the inode for the directory to be opened. If the process does *not* wish to write, the inode for the directory will be returned after the access authorization test.
- If the O_CREAT flag is set, open_namei() not only calls the VFS function lookup() to obtain the inode for the file, but in the event of an error

indicating that the file does not exist, it will generate the file by calling the inode operation `create()`, provided the directory has the appropriate access permissions. The `O_EXCL` flag is also evaluated at this point.

- Once the inode has been obtained using `lookup()`, symbolic links are resolved by calling the `follow_link()` function. As both `lookup()` and `follow_link()` make use of the `iget()` function and therefore also of the `read_inode()` function, the type of file (normal file, device file and so on) is known from this point onwards. Thus, the inode operations contain the file operations specific to the file type in the `default_file_ops` component.
- If the file is a directory and if the process is seeking write permission, `open_namei()` returns the error `EISDIR`.
- The access rights of the inode are now checked. This uses the function `permission()`, which completes its task by reference to those flags shown in Table 6.3 which have been changed. If the file system implementation defines an inode operation `permission()`, it is used instead at this point.
- If access to devices has been prohibited by the mount option `MS_NODEV` and if the file is a device file, the error `EACCES` is returned.
- If the file is not a device file and if the process is attempting to gain write access to a read-only file system, the attempt is aborted with the `EROFS` error.
- If the process is requesting write permission for an 'append only' file for which the `open()` flag `O_APPEND` is not set, `open_namei()` terminates with an `EPERM` error.
- If the `O_TRUNC` flag is set, `open_namei()` calls the function `get_write_access()` to check that the file has not been mapped with `VM_DENYWRITE` set. If this returns successfully, the `notify_change()` function is called to propagate the change in size. Then `open_namei()`

Table 6.3 Conversion of the `open()` flags.

`open()` flag	Value	Bits 1 and 0 of the `open_namei()` flag	Permissions required
		00	None (symbolic links)
`O_RDONLY`	0	01	Read
`O_WRONLY`	1	10	Write
`O_RDWR`	2	11	Read and write
`O_CREAT`	64	1*	Write
`O_TRUNC`	512	1*	Write

sets the i_size component of the inode to zero and calls the inode operation truncate() and the function put_write_access().

The get_write_access() function is used to announce a write access to a file represented by its inode. If the inode is used a number of times, a check is made for each process to confirm that the file has not been mapped with VM_DENYWRITE set. Also, the inode component i_wcount is incremented. The complementary function put_write_access() simply decrements the value of i_wcount. This makes it a simple matter to test whether processes are currently writing to a file which is about to be executed.

If the file survives all of this, open_namei() enters the inode for the newly opened file in res_inode and returns zero to do_open().

This function calls get_write_access() to request write permission for the file where necessary (if bit 1 is set). In addition, it fills the file structure with default values, so that the current file position is set to 0 and the file operations to the default file operations f_inode->i_op->default_file_ops for the inode. The operation open() is then called if it is defined.

This operation takes care of the actions specific to the file type. If the file that has been opened is a file for a character-oriented device, the function chrdev_open() is called at this point, which in its turn modifies the file operations according to the major number of the device.

```
int chrdev_open(struct inode * inode, struct file * filp)
{
   int i;

   i = MAJOR(inode->i_rdev);
   if (i >= MAX_CHRDEV || !chrdevs[i].fops)
      return -ENODEV;
   filp->f_op = chrdevs[i].fops;
   if (filp->f_op->open)
      return filp->f_op->open(inode,filp);
   return 0;
}
```

The file operations for the device drivers are held in the chrdevs[] table, where they were entered by the function register_chrdev() (*see* Chapter 7) when the driver was initialized. The device driver's open() function is certain to add further file operations according to the minor number of the device. This is described in the next chapter.

If no error is returned by any of these open functions, the file has been successfully opened, and the file descriptor is returned to the process by the functions do_open() or sys_open().

6.2.8 The directory cache

The directory cache originates in the *Ext2* file system. Since LINUX version 1.1.37 it has been part of the VFS and can be used by all file system implementations. This cache maintains directory entries which help to speed up access when reading directories, which is necessary when opening files. The entries in the cache have the following structure:

```
struct dir_cache_entry {
   struct hash_list h;            /* hash list management      */
   unsigned long dev;             /* device number             */
   unsigned long dir;             /* directory inode number    */
   unsigned long version;         /* directory version         */
   unsigned long ino;             /* inode number of file      */
   unsigned char name_len;        /* length of directory entry */
   char name[DCACHE_NAME_LEN];          /* directory entry     */
   struct dir_cache_entry ** lru_head; /* ptr to head of list  */
   struct dir_cache_entry * next_lru,
                          * prev_lru;  /* links in list        */
};
```

Only directory entries up to a length of `DCACHE_NAME_LEN` (that is, 15) are held in the cache. This is not a severe limitation, however, as the most frequently used files and directories will have short names.

The directory cache is a two-level cache, with both levels operating according to the LRU (*Least Recently Used*) algorithm. Each of the levels consists of a doubly linked circular list, which always contains `DCACHE_SIZE` entries. The pointers `level1_head` and `level2_head` point to the oldest element in each list, which will be the next to be overwritten. The component `lru_head` in the structure is also a pointer to this, enabling every cache entry to 'know' which level of the cache it is in.

For rapid location of an entry already in the cache, use is made, as regularly occurs in the LINUX kernel, of an open hash list. The hash key is formed from the device number `dev`, the inode number `dir` of the directory and a hash function of the name.

To access the directory cache, two functions are exported. The function

```
void dcache_add(struct inode * dir, const char * name,
                int len, unsigned long ino);
```

enters the directory entry `name` of length `len`, located in the directory with the inode `dir`, in the cache. The number `ino` is the inode number of the directory entry.

If the entry being made is already in the cache, it is revised to make it the youngest in its list, and the function then terminates. A new entry, however, is always inserted in level 1. This involves the oldest element, pointed to by the pointer `level1_head`, being first removed from the hash table and then overwritten with the data for the new directory entry. The pointer `level1_head` is moved on by one entry, so that the newly inserted entry becomes the youngest. Finally, the new entry is added to the hash table.

The function

```
int dcache_lookup(struct inode * dir, const char * name,
                  int len, unsigned long ino);
```

is used to interrogate the cache. If the entry name cannot be found, the function returns zero. If the entry is found in the cache, it is promoted to level 2, where it is entered (or updated, if it was already there). The inode number of the directory entry found is returned in the argument `ino`, and the function itself returns a 1.

Special importance is attached to the component `i_version` in the inode structure. This component is compared with the `version` component in the cache entry, and only if the two match is the cache entry still valid. Every file system implementation must give due attention to the fact that this version is incremented each time the inode of a directory is modified; otherwise it could happen that this function supplies a directory entry which is no longer in the cache. Fortunately, however, this version can be simply updated by the line:

```
dir->i_version = ++event;
```

The variable `event` is defined in `<linux/sched.h>` as an `unsigned long`. This range of values is wide enough to exclude any risk of overlap.

The directory cache thus serves to speed up the `lookup` function specific to a file system, although it is at present used only by the *Ext2* and ISO 9660 file systems. Finally, it should be mentioned that the directory cache is particularly effective in speeding up file access in systems with relatively little memory. On systems with larger memory, this is used for caching block devices anyway, and therefore also maintains directories in memory.

6.3 The *Proc* file system

As an example of how the Virtual File System interacts with a file system implementation, we now take a closer look at the *Proc* file system. The *Proc* file system in this form is peculiar to LINUX. It provides, in a portable way, information on the current status of the LINUX kernel and running processes.

In its general concepts, it resembles the process file system of System V Release 4 and, in some of its approaches, the experimental system Plan 9.[4] Each process in the system which is currently running is assigned a directory `/proc/`*pid*, where *pid* is the process identification number of the relevant process. This directory contains files holding information on certain characteristics of the process. A detailed breakdown of these files and their contents is given in Appendix C.

Let us now take a look at how this file system is implemented. As in so many other places in this book, we will have to manage without reproducing the algorithms in full and restrict ourselves instead to brief explanations of the most important fragments of the program. A full implementation can be found in the directory `fs/proc`.

When the *Proc* file system is mounted, the VFS function `read_super()` is called by `do_mount()`, and in turn calls the function `proc_read_super()` for the *Proc* file system in the `file_systems` list.

```
struct super_block *proc_read_super(struct super_block *s,
                                    void *data, int silent)
{
   lock_super(s);
   s->s_blocksize = 1024;
   s->s_blocksize_bits = 10;
   s->s_magic = PROC_SUPER_MAGIC;
   s->s_op = &proc_sops;
   unlock_super(s);
   if (!(s->s_mounted = iget(s,PROC_ROOT_INO))) {
      s->s_dev = 0;
      printk("get root inode failed\n");
      return NULL;
   }
   parse_options(data, &s->s_mounted->i_uid,
                       &s->s_mounted->i_gid);
   return s;
}
```

Among other things, this initializes the superblock operations (`s_op`) with the special structure `proc_sops`:

```
static struct super_operations proc_sops = {
   proc_read_inode,
   NULL,
```

[4] Plan 9 has been developed by such notable names as Rob Pike and Ken Thompson at AT&T's Bell Labs, and provides a perspective on what the developers of UNIX are currently doing. A good survey of Plan 9 is given in Pike *et al.* (1991).

```
    proc_write_inode,
    proc_put_inode,
    proc_put_super,
    NULL,
    proc_statfs,
    NULL
};
```

The following call to `iget()` then uses this structure to generate the inode for the *Proc* root directory, which is entered in the superblock. The `parse_options()` function then processes the mount options `data` that have been provided (for example, '`uid=1701,gid=42`') and sets the owner of the root inode.

Let us now take a look at what happens when this file system is accessed. An interesting aspect is that in all cases the relevant data are only generated when they are needed. Accessing the file system is always carried out by accessing the root inode of the file system. The first access is made, as described above, by calling `iget()`. If the inode does not exist, this function then calls the `proc_read_inode()` function entered in the `proc_sops` structure.

```
void proc_read_inode(struct inode * inode)
{
   unsigned long ino, pid;
   struct task_struct * p;
   int i;
```

First, the inode is initialized with the default values:

```
   inode->i_op = NULL;
   inode->i_mode = 0;
   inode->i_uid = 0;
   inode->i_gid = 0;
   inode->i_nlink = 1;
   inode->i_size = 0;
   inode->i_mtime = inode->i_atime = inode->i_ctime
                  = CURRENT_TIME;
   inode->i_blocks = 0;
   inode->i_blksize = 1024;
   ino = inode->i_ino;
```

After this, the action depends on the type of inode. We are only interested here in cases in which the inode is the root node of the file system that has been mounted:

```
   if (ino == PROC_ROOT_INO) {
      inode->i_mode = S_IFDIR | S_IRUGO | S_IXUGO;
```

This inode describes a directory (S_IFDIR) with read (S_IRUGO) and execute permissions (S_IXUGO) for all processes. The next step is to calculate the number of references to the directory. As a rule, this will be two plus the number of subdirectories, as each of the subdirectories possesses a reference in the form of '..'. This raises a problem: as the function proc_read_inode() is only called once over the 'lifetime' of the inode in memory, i_link can only be calculated once. This means that the number of processes running at the time when the *Proc* file system was mounted can be taken from the directory listing, especially as the other subdirectories, such as net/, were not taken into account for i_nlink.

All that is required after that is for the inode operations to be set correctly.

```
        inode->i_nlink = 2;
        for (i = 1 ; i < NR_TASKS ; i++)
           if (task[i])
              inode->i_nlink++;
        inode->i_op = &proc_root_inode_operations;
        return;
     } /* if(ino == PROC_ROOT_INO) */
     ...
  } /* proc_read_inode() */
```

The structure proc_root_inode_operations only provides two functions: the component readdir in the form of the proc_readroot() function and the component lookup as the proc_lookuproot() function.

Both functions operate using the table root_dir[], which contains the invariable entries for the root directory.

```
static struct proc_dir_entry root_dir[] = {
   { PROC_ROOT_INO, 1, "." },
   { PROC_ROOT_INO, 2, ".." },
   { PROC_LOADAVG,  7, "loadavg" },
   { PROC_UPTIME,   6, "uptime" },
   { PROC_MEMINFO,  7, "meminfo" },
   { PROC_KMSG,     4, "kmsg" },
   { PROC_VERSION,  7, "version" },
#ifdef CONFIG_PCI
   { PROC_PCI,      3, "pci"  },
#endif
   { PROC_CPUINFO,  7, "cpuinfo" },
   { PROC_SELF,     4, "self" },    /* changes inode # */
     ...
   { PROC_IOPORTS,  7, "ioports"},
```

```
#ifdef CONFIG_PROFILE
   { PROC_PROFILE,  7, "profile"},
#endif
};
```

The individual structures contain the inode number, the length of the filename and the name itself. When the root directory is read, the `proc_readroot()` function accordingly returns the entries given in the field `root_dir[]` along with one entry per process running. However, these directory entries are only generated once the `proc_readroot()` function is called.

A more interesting function than `proc_readroot()`, however, is `proc_lookuproot()`, which determines the inode of a file by reference to the inode for the directory and the name of a file contained in it. In this procedure, the inode numbers are generated in such a way that they can be used later to identify uniquely the file that has been opened.

```
static int proc_lookuproot(struct inode * dir,
                           const char * name,
                           int len, struct inode ** result)
{
   unsigned int pid, c;
   int i, ino;
```

First, the name of the file to be opened is checked to see if it is a name from the `root_dir[]` table.

```
   ...
   i = NR_ROOT_DIRENTRY;
   while (i-- > 0 && !proc_match(len,name,root_dir+i))
      /* nothing */;
   if (i >= 0) {
     ...
```

If it is, the inode number can be read directly from the table. In this case, the inode number `PROC_SELF` represents the directory `self/` and is replaced by an encoded form of the PID for the current process:

```
if (ino == 7) /* self modifying inode ... */
    ino = (current->pid << 16) + 2;
```

Otherwise, an attempt is made to convert the name into a number, which is then interpreted as the process number. This is followed by a check as to whether a matching process (still) exists; and if not, an error is returned. If it does exist, the process number is stored in the variable `ino`.

```
    ...
  {
     pid = string_to_integer(name);

     for (i = 0 ; i < NR_TASKS ; i++)
        if (task[i] && task[i]->pid == pid)
           break;
     if (!pid || i >= NR_TASKS) {
        iput(dir);
        return -ENOENT;
     }
     ino = (pid << 16) + 2;
  }
```

Now `iget()` is called again, to generate the inode. This function in turn calls the function `proc_read_inode()` described above with the relevant inode number.

```
  if (!(*result = iget(dir->i_sb,ino))) {
     iput(dir);
     return -ENOENT;
  }
  iput(dir);
  return 0;
} /* proc_lookuproot() */
```

If the requested inode is that of a process directory, the function finally returns an inode for which the inode operations are given in the structure `proc_base_inode_operations`. However, this structure in its turn contains only the components `readdir` and `lookup` to describe a directory.

This covers the representation of directories in a *Proc* file system, which only leaves the question of how normal files are created. By means of the function `proc_read_inode()`, the inode for most normal files is assigned the function vector `proc_array_inode_operations`. All that is implemented in this, however, is the function `array_read()` in the standard file operations to read the files.

If a process wishes, for example, to read the file `/proc/uptime`, it allocates a free page of memory to the function `array_read()` by calling `__get_free_page()` and passes it to the function `get_uptime()`. This in turn generates the content of the file by entering the required values in the memory page and returning the size of the buffer (in other words, the file). In the sources, this appears as follows:

```
static int get_uptime(char * buffer)
{
   unsigned long uptime;
   unsigned long idle;

   uptime = jiffies;
   idle = task[0]->utime + task[0]->stime;
#if HZ!=100
   return sprintf(buffer,"%lu.%02lu %lu.%02lu\n",
                  uptime / HZ,
                  (((uptime % HZ) * 100) / HZ) % 100,
                  idle / HZ,
                  (((idle % HZ) * 100) / HZ) % 100);
#else
   return sprintf(buffer,"%lu.%02lu %lu.%02lu\n",
                  uptime / HZ,
                  uptime % HZ,
                  idle / HZ,
                  idle % HZ);
#endif
}
```

The functions for the individual files are implemented in `fs/proc/array.c` or in the special sources. The function `get_module_list()` for the file `/proc/module`, for example, is located in the file `kernel/module.c` in the implementation of the module.

6.4 The *Ext2* file system

As LINUX was initially developed under MINIX, it is hardly surprising that the first LINUX file system was the MINIX file system. However, this file system restricts partitions to a maximum of 64 Mbytes and filenames to no more than 14 characters, so the search for a better file system was not long in starting. The result, in April 1992, was the *Ext* file system – the first to be designed especially for LINUX. Although this allowed partitions of up to 2 Gbytes and filenames up to 255 characters, it left the LINUX community far from satisfied as it was slower than its MINIX counterpart and the simple implementation of free block administration led to extensive fragmentation of the file system. A file system which is now little used was presented by Frank Xia in January 1993: the *Xia* file system. This is also based on the MINIX file system and permits partitions of up to 2 Gbytes in size along with filenames of up to

248 characters; but its administration of free blocks in bitmaps and optimizing block allocation functions make it faster and more robust than the *Ext* file system.

At about the same time, Rémy Card, Wayne Davidson and others presented the *Ext2* file system as a further development of the *Ext* file system. It can be considered by now to be *the* LINUX file system, as it is used in most LINUX systems and distributions.

6.4.1 The structure of the *Ext2* file system

The design of the *Ext2* file system was very much influenced by BSD's *Fast File System* (BSD FFS). Thus, a partition is divided into a number of *block groups*, corresponding to the cylinder groups in FFS, with each block group holding a copy of the superblock and inode and data blocks, as shown in Figure 6.4. The block groups are employed with the aim of keeping

- data blocks close to their inodes, and
- file inodes close to their directory inode

and thus reducing positioning time to a minimum, thereby speeding up access to data. As well as this, every group contains the superblock, along with information on all the block groups, allowing the file system to be restored in an emergency.

The physical superblock – defined as the structure `ext2_super_block` – is shown in Figure 6.5. It contains the control information on the file system, such as the number of inodes and blocks. The block size used is not held directly, but as the dual logarithm of the block size minus the minimum block size supported by the *Ext2* file system – in a standard case 0. To use this, all that needs to be done is to 'shift' the minimum block size `EXT2_MIN_BLOCK_SIZE` by the value given. In addition, the superblock includes information on the number of inodes and blocks per block group, along with the times of the last mount operation, the last write to the superblock and the last file system test.

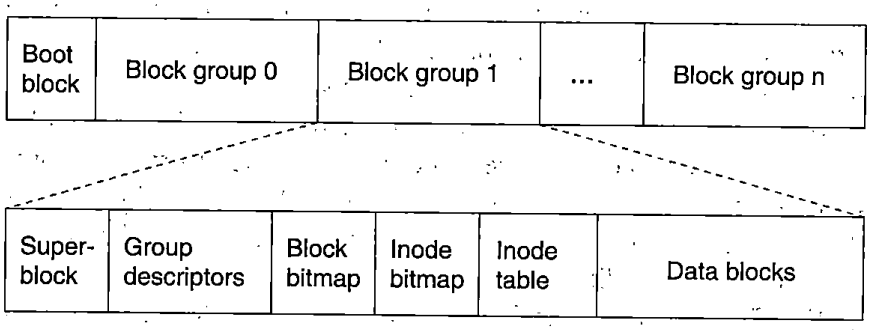


Figure 6.4 Structure of the *Ext2* file system.

	0	1	2	3	4	5	6	7
0	Number of inodes				Number of blocks			
8	Number of reserved blocks				Number of free blocks			
16	Number of free inodes				First data block			
24	Block size				Fragment size			
32	Blocks per group				Fragments per group			
40	Inodes per group				Time of mounting			
48	Time of last write				Status		Max. mount counter	
56	Ext2 signature		Status		Error behaviour		Pad word	
64	Time of last test				Max. test interval			
72	Operating system				File system revision			
80	RESUID		RESGID					

Figure 6.5 The superblock in the *Ext2* file system.

It also holds information on the behaviour of the file system in the event of errors, the maximum time interval to the next file system test, a mount counter and the maximum number of mount operations, which indicates when a mandatory file system test should be carried out. The values *resuid* and *resgid* specify which users or groups are allowed to use the reserved blocks in addition to the superuser.

The superblock is made up to a size of 1024 bytes – the minimum block size `EXT2_MIN_BLOCK_SIZE` – by inserting pad bytes. This makes it a simple matter both to use the space for expansions and to read the superblock using `bread()`.

The superblock is followed in each block group by the *block group descriptors*, which provide information on the block groups. Each block group is described by a 32-byte descriptor (*see* Figure 6.6). This contains the block numbers in the inode bitmap, block bitmap and inode table, the number of free inodes and blocks and the number of directories in this block group. The number of directories is used by the inode allocation algorithm for the directories, which attempts to spread directories as evenly as possible over the block groups – in other words, a new directory will be mounted in the block group with the smallest number of directories.

	0	1	2	3	4	5	6	7
0	Block bitmap				Inode bitmap			
8	Inode table				No. of free blocks		No. of free inodes	
16	No. of directories		Pad words					
24								

Figure 6.6 The block group descriptors in the *Ext2* file system.

The bitmaps are each the size of one block. This restricts the size of a block group to 8192 blocks for blocks of 1024 bytes.

The inode table for a block group lists consecutive blocks, starting with the one specified, and consists of inodes 128 bytes in size (*see* Figure 6.7). In addition to the data already mentioned, these contain the time when the file was deleted (to use in restoring deleted files), entries for ACLs (*Access Control Lists* to enable access permissions to be differentiated more precisely) and information specific to the operating system used. At present, ACLs are not implemented, which means that the function `ext2_permission()` tests only the UNIX permissions and the `S_IMMUTABLE` flag.

If the inode refers to a device file (that is, if `S_IFCHR` or `S_IFBLK` in `i_mode` is set) the first block number (`i_block[0]`) will give the device number. For a short symbolic link (`S_IFLNK`) the block numbers include the path, so that no additional data block is required and the 'number of blocks' field, `i_blocks`, will contain a value of zero. If the symbolic link is longer than

```
EXT2_N_BLOCKS * sizeof (long)
```

it will be stored in the first block. This limits the maximum length of a reference to the size of a block.

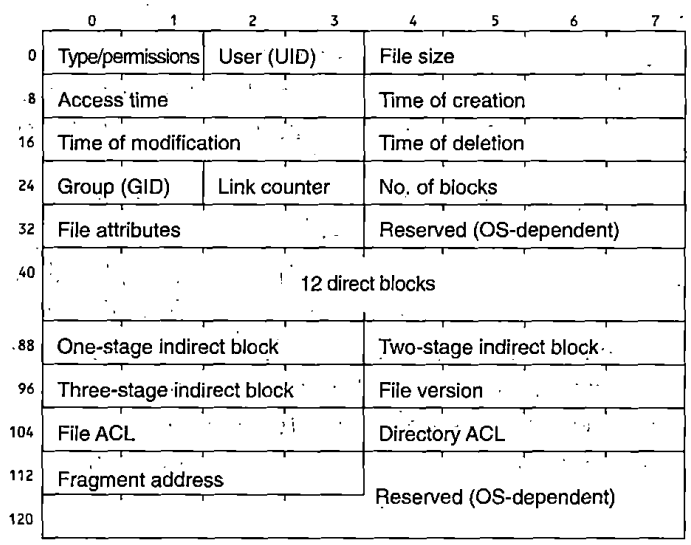


Figure 6.7 The inode in the *Ext2* file system.

6.4.2 Directories in the *Ext2* file system

In the *Ext2* file system, directories are administered using a singly linked list. Each entry in this list has the following structure.

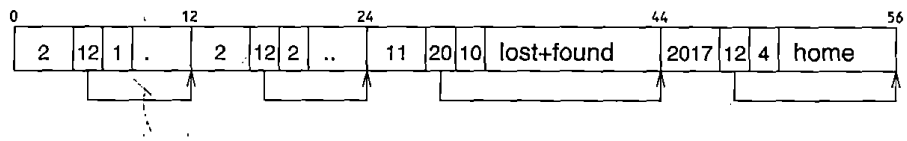


Figure 6.8 A directory entry in the *Ext2* file system.

```
struct ext2_dir_entry {
   unsigned long  inode;     /* inode number                    */
   unsigned short rec_len;   /* length of directory entry       */
   unsigned short name_len;  /* length of filename              */
   char           name[EXT2_NAME_LEN];  /* filename             */
};
```

The field `rec_len` contains the length of the current entry, and is always rounded up to a multiple of 4. This enables the start of the next entry to be calculated. The `name_len` field holds the length of the filename. It is perfectly possible for a directory entry to be longer than is required to store the filename. A possible structure is shown in Figure 6.8.

An entry is deleted by setting the inode number to zero and removing the directory entry from the linked list: that is, the previous entry is simply extended. This eliminates the need for shift operations in the directory, which might otherwise exceed the limits of the buffers. However, the 'lost space' is not wasted, but is reused when a new name is entered, either by overwriting an entry with a value of 0 or by using the additional space provided by removal of the link.

6.4.3 Block allocation in the *Ext2* file system

A problem commonly encountered in all file systems is the fragmentation of files – that is, the 'scattering' of files into small pieces as a result of the constant deleting and creating of new files. This problem is usually solved by the use of 'defragmentation programs', such as `defrag` for LINUX. Some file systems attempt to prevent fragmentation as far as possible by sophisticated systems of block allocation. The *Ext2* file system similarly uses two algorithms to limit the fragmentation of files.

- ***Target-oriented allocation***
  This algorithm always looks for space for new data blocks in the area of a 'target block'. If this block is itself free, it is allocated. Otherwise, a free block is sought within 32 blocks of the target block, and if found, is allocated. If this fails, the block allocation routine tries to find a free block which is at least in the same block group as the target block. Only after these avenues have been exhausted are other block groups investigated.

- ***Pre-allocation***
  If a free block is found, up to eight following blocks are reserved (if they are free). When the file is closed, the remaining blocks still reserved are released. This also guarantees that as many data blocks as possible are collected into one cluster. Pre-allocation of blocks can be deselected by removing the definition of `EXT2_PREALLOCATE` from the file `<linux/ext2_fs.h>`.

How is the target block itself determined? Let `n` be the relative number in the file of the block to be allocated and `l` the logical block number of the last block allocated. The block allocation algorithm then applies the following heuristics in the order given:

- The last block allocated had the relative number `n-1`. The target block is therefore `l+1`.
- All existing blocks in the file, starting at block number `n-1`, are scanned to confirm that they have been assigned logical blocks (that is, the block is not a 'gap'). The target block is given by the number `l` of the first already allocated block found.
- The target block is the first block in the block group in which the inode for the file is located.

6.4.4 Extensions of the *Ext2* file system

The *Ext2* file system has additional file attributes beyond those which exist in standard UNIX file systems (*see* Table 6.4). In version 0.5a, which is current at the time of writing, these are:

EXT2_SECRM_FL
: If a file has this attribute, its data blocks are first overwritten with random bytes before they are released via the `truncate` function. This ensures that the content of the file cannot possibly be restored after it has been deleted.

EXT2_UNRM_FL
: This attribute will eventually be used to implement the restoration of deleted files. At present, however, this function is not implemented.

EXT2_COMPR_FL
: This attribute will be used to indicate that the file has been compressed. Up to the present, online compression has not yet been implemented.

EXT2_SYNC_FL
: If a file has this attribute, all write requests are performed synchronously, that is, not delayed by the buffer cache.

Table 6.4 File attributes in the *Ext2* file system (n.i. = not yet implemented).

Macro	Value	Description
EXT2_SECRM_FL	1	Secure deletion
EXT2_UNRM_FL	2	Undelete (n.i.)
EXT2_COMPR_FL	4	Compressed file (n.i.)
EXT2_SYNC_FL	8	Synchronous write
EXT2_IMMUTABLE_FL	16	Unmodifiable file
EXT2_APPEND_FL	32	'Append only' file
EXT2_NODUMP_FL	64	Do not archive file

EXT2_IMMUTABLE_FL

Files with this attribute cannot be deleted or amended. Renaming and the setting up of further hard links are also prohibited. Even the superuser cannot modify the file so long as it possesses this attribute. Directories with this attribute cannot be changed – that is, no new files can be created or deleted. Existing files or subdirectories, however, can be modified as desired.

EXT2_APPEND_FL

As for the previous attribute, files with this attribute cannot be deleted, renamed or relinked. However, this attribute does allow a write to the file to add fresh data. Directories with this attribute will only allow new files to be created. These will inherit the `EXT2_APPEND_FL` attribute when they are created.

EXT2_NODUMP_FL

This attribute is not used by the kernel. It is intended to be used to mark files which are not required in a backup.

However, these attributes can be changed using the `chattr` program. The program `lsattr` displays them.

The development of the *Ext2* file system is not yet complete. The list of planned expansions includes:

- restoration of deleted files,
- ACLs, and
- automatic file compression.

7 Device drivers under LINUX

A computer terminal is not an old lump of a television
with a typewriter keyboard sitting in front of it.
It is an interface
connecting body and spirit with the universe
and enabling bits of it to be moved around.

Douglas Adams

7.1 Character and block devices
7.2 Polling and interrupts
7.3 The hardware
7.4 Implementing a driver
7.5 An example of DMA operation

There is a wide variety of hardware available for LINUX computers. This means that a wide variety of software is required to operate this hardware. This is the job of device drivers. Without these, an operating system like LINUX would have no means of input or output (such as a keyboard and a monitor) and no file systems. Device drivers are the nuts and bolts of any operating system.

In addition, the computer hardware in UNIX systems is meant to be hidden from the user, without limiting its functions. This is done by having physical devices represented by files, which allows portable programs to be developed that can access both the various devices and the files with the same system calls, for example `read` and `write`. To handle this, device drivers are integrated into the LINUX kernel and given exclusive control of the hardware.

As a result, if a device driver has been properly implemented, the corresponding device can never be used wrongly by the user. This *protective function* of a device driver should not be underestimated.

In this chapter, we shall demonstrate the functioning and the correct implementation of device drivers. The example chosen is the PC speaker driver, which supports the output of sound samples to the internal speaker or a digital–analog converter connected to the parallel interface. It is also designed to be compatible with the sound card driver.

As a number of device drivers have to exist side by side in the LINUX kernel, they are uniquely identified by their *major numbers*. A device driver may be controlling a number of physical and virtual devices, for example a number of hard disks and partitions; thus, the individual device is accessed via its *minor number*, an integer between 0 and 255.

The exception to this rule is the device driver for terminals and serial interfaces, which uses the two major numbers 4 and 5. The devices with the major number 4 are virtual consoles, simple serial interfaces (call-in devices) and *pseudoterminals*.[1] Virtual consoles are given the minor numbers 0 (for `tty0`) to 63. The special device `/dev/tty0` or `/dev/console` is always the current virtual console.

For every serial interface there are two logical devices, the *dial-in device* `ttyS`*n* and the *call-out device* `cua`*n*. When the dial-in device is opened, a process, such as `getty`, will be blocked until the DTR line at the interface is active. A process opening the call-out device, usually a dial-out program, will be given immediate access to the serial interface if no other process is using it. This will also continue to block a process wishing to open the dial-in device. The serial dial-in devices are given the minor numbers 64 (for `ttyS0`) to 127.

The remaining minor numbers from 128 to 255 are used for pseudoterminals. The master terminal `pty`*n* is given the minor number 128+*n*, while the corresponding slave terminal `ttyp`*n* has the minor number 192+*n*.

Major number 5 is reserved for the current terminal and for the call-out devices. The device `/dev/tty` with the minor number 0 is always the terminal belonging to the process. The call-out devices `cua`*n* have the corresponding minor numbers 64+*n* and thus differ from their 'twins' only in their major numbers.

In the same way as for file systems, device drivers need to be made known to the LINUX kernel. This is done when the system is started up or when the driver modules are initialized, and uses one of the functions

```
int register_chrdev(unsigned int major,
                    const char * name,
                    struct file_operations *fops);
int register_blkdev(unsigned int major,
                    const char * name,
                    struct file_operations *fops);
```

The file operations specified and their symbolic names are entered under the major numbers given in the table `chrdevs[]` (for character devices) or

[1] Pseudoterminals are pairs of master and slave terminals, acting together like one terminal unit. The slave terminal is the interface, acting like a terminal unit as far as the user program is concerned, while the master represents the other end of the link (on a terminal, this is the user) (*see* Stevens (1992b)).