S.Ha.R.K. User Manual

Volume VI

S.Ha.R.K. File System

Written by

Paolo Gai (pj@sssup.it)

Original italian version by

Massimiliano Giorgi (massy@gandalf.sssup.it)

Scuola Superiore di Studi e Perfezionamento S. Anna

RETIS Lab

Via Carducci, 40 - 56100 Pisa

Contents

1 File system architecture **1**
1.1 Features 1
1.2 Low Level: Device Drivers 1
1.3 High Level: File systems 3

2 Device drivers **5**
2.1 Interface 5
2.1.1 Initialization 5
2.1.1.1 How to create a new device driver 6
2.1.2 Device serial number 6
2.1.3 Implementation details 7
2.1.3.1 block_device_operation structure. 9
2.1.3.2 The "lodsk" module 9
2.1.3.3 Semaphores 10
2.1.4 Exported interface 11
2.2 I/O requests scheduling 12
2.3 Disk scheduling algorithms implemented in S.Ha.R.K. 12
2.3.1 FCFS 13
2.3.2 SSTF 13
2.3.3 LOOK 14
2.3.4 CLOOK 15
2.3.5 EDF 15
2.4 Device driver: the IDE interface 16
2.4.1 Description 16
2.4.2 Implementation 16
2.4.2.1 Initialization 17
2.4.2.2 Policy queues Interface 18
2.4.2.3 Minor numbers 19

3 File system **20**
3.1 Interface 20
3.1.1 Initialization 20
3.1.1.1 Struct file_system_type 22
3.1.2 Internal interface 22
3.1.2.1 struct super_operations 23
3.1.2.2 Struct dentry_operations 23
3.1.2.3 struct inode_operations 23
3.1.2.4 struct file_operations 24
3.1.3 External interface 25
3.1.3.1 Non standard functions 25
3.1.4 Initialization: an example 26
3.2 Internal structure 27

3.2.1 The data structures . . . 28
3.2.1.1 struct super_block . . . 28
3.2.1.2 struct dentry . . . 29
3.2.1.3 struct inode . . . 31
3.2.1.4 struct file . . . 33
3.2.1.5 struct file_descriptors . . . 34
3.2.2 Disk Cache . . . 35
3.2.2.1 How to use the cache . . . 35
3.2.2.2 struct dcache . . . 35
3.2.2.3 struct __rwlock_t . . . 38
3.3 Filesystem: MS-DOS (FAT16) . . . 39
3.3.1 Description . . . 39
3.3.1.1 Internal structure . . . 39
3.3.1.2 The boot sector . . . 39
3.3.1.3 The File Allocation Table . . . 39
3.3.1.4 The "root directory" . . . 42
3.3.1.5 The data area . . . 43
3.3.2 Implementation notes . . . 43

List of Figures

1.1 File system architecture . . . 2
1.2 Device Drivers . . . 2
1.3 Device Drivers - details . . . 3
1.4 High Level - File system details . . . 4

2.1 Device Drivers Initialization. . . . 6
2.2 Device Serial Number . . . 7
2.3 The IDE subsystem . . . 16
2.4 IDE minor numbers . . . 19

3.1 Filesystem init . . . 21
3.2 Super block tree . . . 29
3.3 Directory entry tree. . . . 30
3.4 Inode data structures . . . 32
3.5 Data structure for disk cache . . . 36
3.6 Internal structure of the MSDOS FAT16. . . . 40
3.7 An example of File Allocation Table (FAT) . . . 41
3.8 The FAT16 file serial number. . . . 43

List of Tables

2.1 The BDEV_PARMS structure 5
2.2 Block device registration functions 7
2.3 The phdsk structure. 8
2.4 block_device Structure 8
2.5 block_device_operations structure. 9
2.6 The lodsk module 9
2.7 The semaphore interface 10
2.8 The mutex interface 10
2.9 Low level functions 11
2.10 Queue handling functions 12
2.11 struct request_prologue_t 13
2.12 struct fcfs_queue_t 13
2.13 struct sstf_queue_t 14
2.14 struct look_queue_t 14
2.15 struct clook_queue_t 15
2.16 struct look_queue_t 15
2.17 Resource Module for deadlines 16
2.18 Data structures used to initialize the IDE driver. 17
2.19 The glue code for the IDE driver. 17
2.20 struct idereq_t 18

3.1 struct filesystem_parms 20
3.2 struct mount_opts 21
3.3 struct file_system_type 22
3.4 struct super_operations 22
3.5 struct dentry_operations 23
3.6 struct inode_operations 23
3.7 struct file_operations 24
3.8 mount/umount functions 25
3.9 struct super_block 28
3.10 struct dentry 30
3.11 struct qstr 31
3.12 struct inode 32
3.13 struct stat 32
3.14 struct file 33
3.15 struct file_descriptors 34
3.16 Disk cache interface 35
3.17 struct dcache 36
3.18 struct __rwlock_t 38
3.19 Lockers functions 38
3.20 The MS-DOS boot sector. 41
3.21 MSDOS Directory Entry. 42

Abstract

This document is the raw translation from Italian to English of Massimiliano Giorgi's Thesis. It has not been checked by anyone (neither me ;-)), except for an ispell pass.

This is the only documentation currently available for the S.Ha.R.K. File system. The document gives an insight on many data structures used internally by the file system. We can say this is a first step toward a *real* file system user manual. If you carefully read it, you will understand how it works, and how you can initialize and use it... or, at least, I hope that...

In any case, please send any comments to pj@sssup.it.

Enjoy,

Paolo

Chapter 1

File system architecture

1.1 Features

The library is composed by two main layers:

File system The file system contains all the data structures used to handle files.

Blocks devices This part handles directly the hardware. They provide a common interface that unifies the usage of all the block devices (i.e., the hard disks).

The architecture of the library is modular, and it is described in Figure 1.1.

The library is connected to the Kernel via a *glue* layer, to the application via a *wrapper* layer; the block devices and the file system operations are separated by a cache, that may be excluded.

The library needs a set of primitives that must be supported by the underlying kernel, as memory handling, string handling, and functions with a variable number of parameters. Moreover, the kernel should support functions that implement:

- Mutual exclusion.
- Allocation of low level resources (as I/O spaces and interrupts).
- Memory protection (not needed in S.Ha.R.K.).

1.2 Low Level: Device Drivers

The device drivers are divided in two main parts:

- A general part that exports the basic low level services. It also gives a set of internal routines to the internal modules.
- A set of internal modules that handle one or more peripherals. The application can choose which module must be linked, initialized and used at compile time.

The general part is composed by four modules:

Block device The main part; it exports the interface to the upper layers, inits the device driver subsystem and register all the drivers present in the system.

Logical disk This module provides functions that allow to inspect the logical structure of an hard disk, like start, end, size and type of each partition on the Hard Disk.

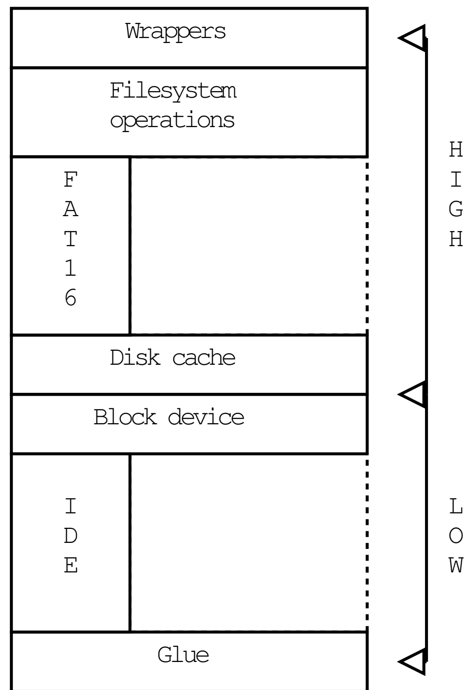

Figure 1.1: File system architecture

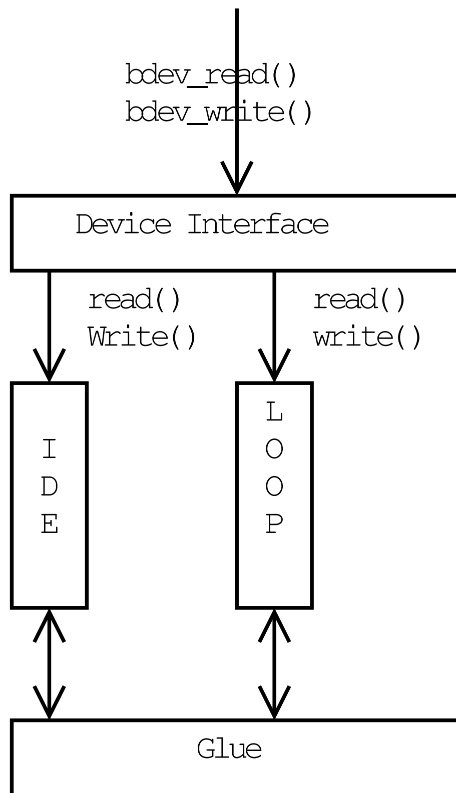

Figure 1.2: Device Drivers

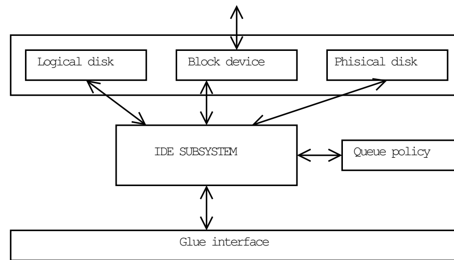

Figure 1.3: Device Drivers - details

Physical disk This module provide a small data base that contains informations about all the hard disks in the system; these informations can be useful for the module that must handle the low level pending request queues.

Queue policy This module handles the low level policies for the pending requests (SCAN, LOOK, EDF, ...).

1.3 High Level: File systems

The high level is divided in three parts:

- The first part handle the file systems, its initialization and its interface to the internal modules.
- A second part composed by a set of modules that implement a particular file system (i.e., FAT16).
- A cache module (whose policy can be easily changed).

The first part is composed by sub-modules (internal implementation of a sub-module can be changed without affecting the rest of the library):

Filesystems It is responsible for the initialization and termination of all the high level layer. It registers all the file systems, and it contains the routines for mounting/unmounting the partitions.

File It handles the descriptor tables and the files; it defines the `file` structure.

Super It handles the super blocks (that contains the global informations about partitions on a disk).

Inodes It handles the inodes (informations about a file in the file system).

Dentry It handles file names in a tree structure.

Open, read, umask, ... They implement the correspondent system primitives.

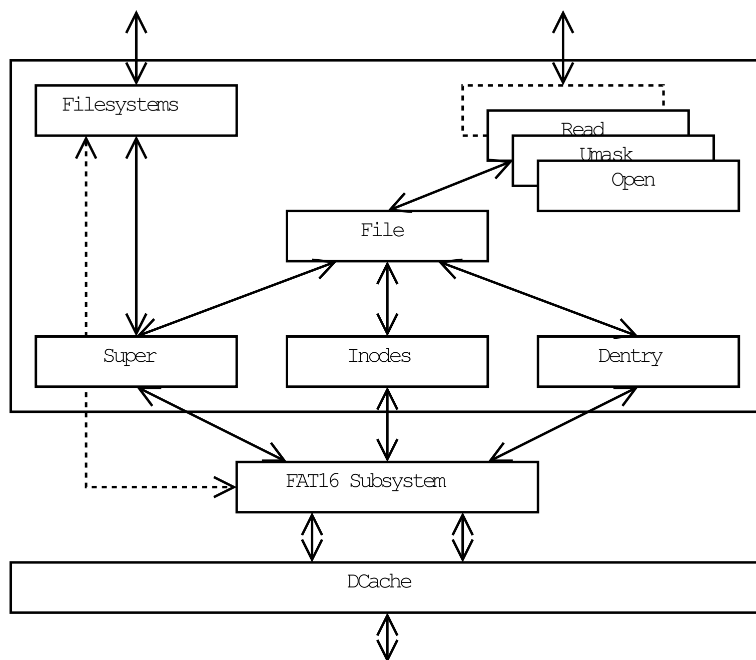

Figure 1.4: High Level - File system details

Chapter 2

Device drivers

2.1 Interface

2.1.1 Initialization

The bdev_init() function (see Table 2.1) must be called to initialize a device driver, passing a pointer to a BDEV_PARMS structure. Use the provided macros to init that structure.

The structure is composed by two parts: a generic part and a specific part. The generic part is composed by the following fields:

showinfo If this flag is set, the initialization of the device will be "verbose". To set this flag, use the macro bdev_def_showinfo.

config It is a string that can have a particular meaning for the device driver. The string is parsed by the device driver, that usually searches for some initialization parameter (something like "foo:[number]"). To set this string, use the macro bdev_def_configstring.

bmutexattr This parameter can be used to tell the system which is the kind of mutex to use to implement the mutual exclusion. To set this parameter, use the macro `bdev_def_mutexattrptr`.

dummy Used to align the data structure (see the macro BASE_DEV).

The specific part is directly specified by the device driver; the bdev_init function can be called can be called passing a pointer to a BDEV_PARMS structure initialized with a BASE_BDEV value. If NULL is specified, the default values are used.

```
typedef struct bdev_parms {
  __uint32_t showinfo:1;
  char *config;
  void *bmutexattr;
  __uint16_t dummy;
  #ifdef IDE_BLOCK_DEVICE
  IDE_PARMS ideparms;
  #endif
} BDEV_PARMS;

void bdev_def_showinfo (BDEV_PARMS s, int v);
void bdev_def_configstring(BDEV_PARMS s, char *v);
void bdev_def_mutexattrptr(BDEV_PARMS s,void *v);
```

Table 2.1: The BDEV_PARMS structure

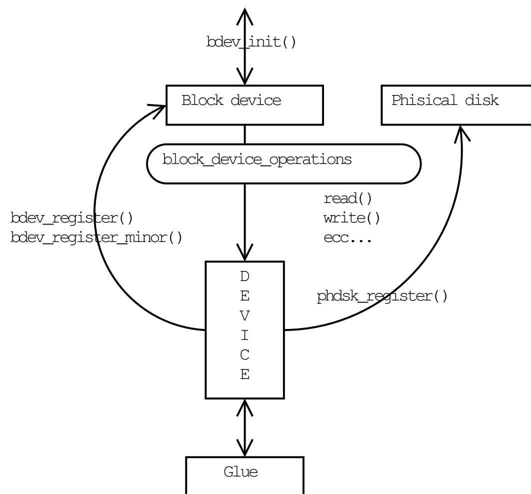

Figure 2.1: Device Drivers Initialization.

2.1.1.1 How to create a new device driver

1. Add a new *major* number in t the file include/fs/major.h (see 2.1.2);
2. Add a "#define" into the file include/fs/bdevconf.h (to enable selective compilation);
3. If needed, add fields to bdev_parms, and modify the BASE_BDEV macro with the new default parameters;
4. Modify the function bdev_init to call the init function of the new device driver (into drivers/block/bdev.c).

The device driver initialization function, after the initialization of its internal part, does the following actions (see Figure 2.1):

1. It registers itself calling bdev_register(); That function must be called for each device handled by the driver.
2. It registers every block device that it find into its interfaces to the Physical Disk Module.
3. It registers every logical device found calling bdev_register_minor().

Data structures and registration functions used in this phased are described in Section 2.1.3.

2.1.2 Device serial number

Every logical device (i.e., a logical partition of a hard disk) has a device serial number of type __dev_t, that must have a unique value (see the POSIX standard).

The number has an internal structure (see Figure 2.2) composed by a *major number*, and another number (whose number depends on the device implementation) called *minor number.*

Three macro exists (see include/fs/sysmacro.h) to handle the device numbers:

makedev This macro creates a __dev_t number from the major and the minor number.

major Returns the major number of a device.

minor Returns the minor number of a device.

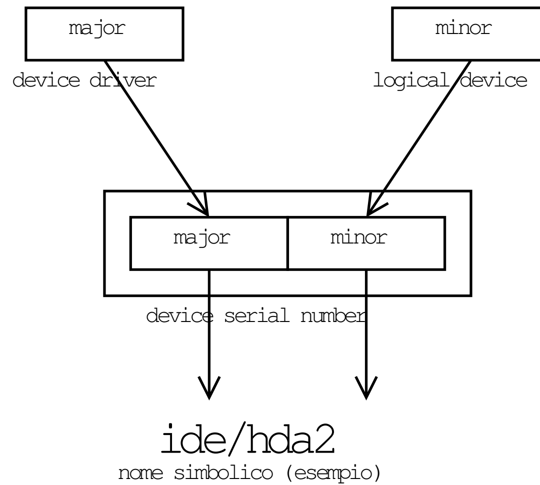

Figure 2.2: Device Serial Number

```
int bdev_register(__dev_t major, char *name, struct block_device *);
int bdev_register_minor(__dev_t device, char *name, __uint8_t fsind);
struct phdskinfo *phdsk_register( struct phdskinfo *disk);
```

Table 2.2: Block device registration functions

The major number is used to decide which device driver will handle a request (see Section 2.1.3). A device driver can register more than on major number. Minor numbers are device driver dependent, and the validity of a minor number is checked only by the driver.

To simplify the search of a device, every device has a symbolic name (see Section 2.1.4). Symbolic names are composed by two parts: a unique name for each major, and a specific name assigned to each minor. For example, the major MAJOR_B_IDE has a symbolic name “ide”; a minor name for that major can be “hda2”, so the full name is “ide/hda2”, that identifies the second partition on the master hard disk on the first IDE controller interface (see Section 2.4).

2.1.3 Implementation details

These are the registration functions for the device drivers (see Table 2.2):

bdev_register This function registers a device type (see Section 2.1.2); the function receives as parameter the major number, its symbolic name, and a pointer to a structure block_device. The function returns 0 in case of success, != 0 otherwise.

bdev_register_minor It registers a minor number, its symbolic name and the file system type (see include/fs/fsind.h). This information is used by bdev_finddevice_byname (see Section 2.1.4). The function returns 0 in case of success, != 0 otherwise.

phdsk_register This function is used to register every physical device found with parameters that can be used by the Disk Scheduling Algorithms (see Section 2.2). This function receives a pointer to a structure phdiskinfo that contains the device parameters, and returns NULL in case of error, or a pointer to the registered structure (that can be != from the parameter) in case of success.

The phdskinfo structure contains the physical parameters of a device. all the field must be filled before calling phdsk_register. The fields are:

```
struct dskgeometry {
  __uint16_t cyls;
  __uint16_t heads;
  __uint16_t sectors;
};

struct phdskinfo {
  char pd_name[MAXPHDSKNAME];
  __dev_t pd_device;
  __blkcnt_t pd_size;
  int pd_sectsize;
  struct dskgeometry pd_phgeom;
  struct dskgeometry pd_logeom;
};
```

Table 2.3: The phdsk structure.

```
struct block_device {
 char *bd_name;
 int bd_sectorsize;
 __uint32_t bd_flag_used:1;
 struct block_device_operations *bd_op;
};
```

Table 2.4: block_device Structure

pd_name Device name, used only for debug purposes.

pd_device device identificator.

pd_size number of logical sectors of the device.

pd_sectsize Dimension (bytes) of a logical sector.

pd_phgeom e pd_logeom Geometry of the device (logical and phisical); geometries are described using the struct dskgeometry (see Table 2.3):

- **cyls** number of cylinders (traces) of the device.
- **heads** number of heads.
- **sectors** number of sectors for each cylinder.

The struct block_device described in Table 2.4 is used to register the block devices:

bd_name Physical device name; for example, for the IDE device driver, "ide".

bd_sectorsize logical sector dimension (bytes; redundant info but useful ;-)).

bd_flag_used It should not modified by the device drivers. If set, the corresponding device driver is present (a table of all the possible devices is used to access the system in a fast way).

bd_op Virtual device operations.

```
struct block_device_operations {
  int (*_trylock) (__dev_t device);
  int (*_tryunlock)(__dev_t device);
  int (*read) (__dev_t device, __blkcnt_t blocknum, __uint8_t *buffer);
  int (*seek) (__dev_t device, __blkcnt_t blocknum);
  int (*write) (__dev_t device, __blkcnt_t blocknum, __uint8_t *buffer);
};
```

Table 2.5: block_device_operations structure.

```
struct lodskinfo {
  __uint8_t fs_ind;
  __blkcnt_t start;
  __blkcnt_t size;
};

typedef int (*lodsk_callback_func)(int, struct lodskinfo*, void *);

int lodsk_scan(__dev_t device, lodsk_callback_func func, void *data, int
showinfo, char *lname);
```

Table 2.6: The lodsk module

2.1.3.1 block_device_operation structure.

This structure contains the virtual operation that have to be supplied by every device driver:

read This function should read a logical pblock passed as parameter, writing the result on the buffer. It returns 0 in case of success, != 0 otherwise.

write Similar to read, but it writes ;-).

seek moves the head on the required block.

_trylock e _tryunlock Used to block/unblock a device. They return 0 in case of success, != 0 otherwise; after initialization, and usually, every device is in the unblocked state.

Why blocking/unblocking a device? The problem, described in Section 3.2.2.2, is that the data cache have to refer to a logical block in an unique way. Since there are logical devices that shares blocks (think at /dev/hda and /dev/hda2 in Linux), the mount operation must be done in mutual exclusion, blocking the device.

2.1.3.2 The "lodsk" module

This module (see Table 2.6) offer a service to the device drivers: It recognizes the logical structure (partitions) of an HD as described into [?] (part of the code is derived from Linux).

before calling lodsk_scan, the device driver must be already registered; the function has five parameters:

1. The device ID where the search have to be done.
2. A callback function called for each partition found.
3. A generic pointer passed to the callback function.
4. A verbose flag.
5. The name of the logical device (i.e., "hdb", used if the verbose option has been set.

```
typedef internal_sem_t __sem_t;

#define __sem_init (ptr,val)
#define __sem_wait (ptr)
#define __sem_trywait (ptr)
#define __sem_signal (ptr)
```

Table 2.7: The semaphore interface

```
typedef internal_sem_t __mutex_t;
#define __mutex_init (ptr,attr)
#define __mutex_lock (ptr)
#define __mutex_trylock (ptr)
#define __mutex_unlock (ptr)
typedef SYS_FLAGS __fastmutex_t;
#define __fastmutex_init (ptr)
#define __fastmutex_lock (ptr)
#define __fastmutex_unlock (ptr)
```

Table 2.8: The mutex interface

Each callback function is called passing as arguments:

1. The logical number of the partition (the Linux numbering scheme is used).
2. A structure with informations on the partition found. The structure has the following fields:

   fs_ind The file system number (see include/fs/fsind.h).

   start The partition starting block.

   size The partition size (number of blocks).

3. A the generic pointer passed to lodsk_scan.

The low_level routines always check these values to avoid read/write operations outside the partition.

2.1.3.3 Semaphores

The files include/fs/mutex.h and include/fs/semaph.h contain the mutex/semaphore interface used by the file system.

In particular, the semaphore interface (see Table 2.7) is simply a wrapper to the S.Ha.R.K. Internal Semaphores.

The mutex interface requires two types of mutexes:

1. Fast mutexes: when only a few lines have to be protected (remapped to kern_fsave/kern_frestore).
2. Normal mutexes: without any restriction (remapped again to the internal semaphores).

Please note that these functions are not directly used in the code; instead, functions like __b_mutex_lock and __fs_mutex_lock are used. This is done to make possible the disabling of the mutual exclusion requirement when not needed (i.e., if the file system runs as a process with its own address space).

```
int bdev_read (__dev_t dev, __blkcnt_t blocknum, __uint8_t *buffer);
int bdev_write (__dev_t dev, __blkcnt_t blocknum, __uint8_t *buffer);
int bdev_seek (__dev_t dev, __blkcnt_t blocknum);
int bdev_trylock(__dev_t device); int bdev_tryunlock(__dev_t device);
__uint8_t bdev_getdeffs(__dev_t device);
__dev_t bdev_find_byname(char *name);
__dev_t bdev_find_byfs(__uint8_t fsind);
int bdev_findname(__dev_t device, char **majorname, char **minorname);
int bdev_scan_devices(int(*callback) (__dev_t,__uint8_t));
```

Table 2.9: Low level functions

2.1.4 Exported interface

The low level exports two kinds of functions: one to use the devices, one to search/get the logical devices present on a system.

The three main functions are:

bdev_find_byname This function accepts a symbolic device name, as described in Section 2.1.2, and returns 0 if the device is not registered, or its number otherwise.

bdev_find_byfs This function accepts a file system ID (see include/fs/fsind.h), and returns the first device that contains that file system, or 0 if no one provides it.

bdev_scan_device This function can be used to get a list of all the devices in the system; the user have to pass a callback that is called for each device in the system. The function returns 0 in case of success, -1 if an error occurred, or a value != 0 if the callback returns a value != 0. The callback is called with a device serial number and a file system type that should be present in the device.

These function are useful when at system startup the root file system have to be mounted (see Section 3.1.3.1).

The functions bdev_getdeffs and bdev_findname are used for these purposes:

bdev_getdeffs to know a probable file system present on a logical device.

bdev_findname to know the logical name of a device. The function returns != 0 in case of error, and 0 in case of success. In the latter case, the two buffers passed as parameter are filled with the major and minor names.

These functions simply check the parameter values and then they calls the device drivers functions (see Section 2.1.3.1):

bdev_read Reads a block on the device.

bdev_write Write a block on the device.

bdev_seek Move the head in the position passed as parameter.

bdev_trylock e bdev_tryunlock Tries to block/unblock a device.

These functions can block the calling thread for an unspecified amount of time. All the device drivers were thought to be used in a multithreaded systems.

If a device driver is not multithread, proper mutual exclusion must be ensured, because the file system high level functions will call a low level service before the previous request has been served. Note that if such a mutual exclusion is used, the access to the device driver will be forced by the mutex policy, and so all the disk scheduling algorithms will not work properly.

```
int bqueue_init (bqueue_t *);
int bqueue_numelements (bqueue_t *);
int bqueue_removerequest (bqueue_t *);
int bqueue_insertrequest (bqueue_t *,
struct request_prologue *);
struct request_prologue *bqueue_getrequest(
bqueue_t *);
```

Table 2.10: Queue handling functions

2.2 I/O requests scheduling

Massy's thesis here insert a dissertation on the various disk scheduling algorithms that has been removed :-)

2.3 Disk scheduling algorithms implemented in S.Ha.R.K.

The system currently support 5 different scheduling algorithms that can be changed/increased simply modifying system configuration.

Only one algorithm can be active at a time. All the device drivers should use the queuing interface listed in the bqueue.h file (see Table 2.10). The structure bqueue_t is specific for each scheduling algorithm. The semantic of each function in the interface is the following:

bqueue_init Initialize the pending request queue; it returns 0 in case of success, != 0 otherwise.

bqueue_numelements Returns the number of elements in the queue.

bqueue_insertrequest This function inserts a request in the queue, following the specific algorithm behavior. The function returns 0 in case of success, != 0 otherwise. Please note that requests are passed through a pointer to the structrequest_prologue_t (see tab:structreqprolog). That is, every algorithm should use a structure with a struct request_prologue as first field, passing a casted pointer to that structure.

bqueue_getrequest Returns the first request in queue (without removing it), and returns a pointer to that request, or NULL if there are not any pending requests.

bqueue_removerequest Removes the first request in queue, that is the request that would have been returned by bqueue_getrequest; it returns 0 in case of success, != 0 in case of error.

before calling bdev_insertrequest, all the fields of the struct request_prologue_t must be filled (see Table 2.11):

sector, head, cylinder This is the starting position in the hard disk.

nsectors This is the number of sectors to transfer.

operation The operation that must be performed:

- **REQ_DUMMY** request not specified.
- **REQ_SEEK** head seek to the specified position.
- **REQ_READ** read request.
- **REQ_WRITE** write request.

```
#define REQ_DUMMY 0
#define REQ_SEEK 1
#define REQ_READ 2
#define REQ_WRITE 3
struct request_prologue {
  unsigned sector;
  unsigned head;
  unsigned cylinder;
  unsigned nsectors;
  unsigned operation;
  /*--*/
  struct request_specific x;
};
#define request_prologue_t struct request_prologue
```

Table 2.11: struct request_prologue_t

```
typedef struct TAGfcfs_queue_t {
  struct phdskinfo *disk;
  /**/
  __b_fastmutex_t mutex;
  struct request_prologue *head;
  struct request_prologue *tail;
  int counter;
} fcfs_queue_t;
struct request_specific {
  void *next;
};
```

Table 2.12: struct fcfs_queue_t

To add algorithm specific fields, you can use the method used for the deadlines into Section 2.3.5.

struct request_specific (see Table 2.11) must be defined by each algorithm, and contains algorithm-specific fields.

struct bqueue_t contains a scheduling queue. Its first field must be a pointer to a struct phdiskinfo (see Section 2.1.3, and Table 2.3).

Here is a description about the implementation of various disk scheduling algorithms.

2.3.1 FCFS

The FCFS policy is implemented using the structure in table 2.12:

mutex A mutual exclusion semaphore.

head e tail a pointer to head and tail of a request queue.

counter number of pending requests.

Then, the struct request_specific contains a pointer that is used to link the list.

2.3.2 SSTF

This scheduling policy is implemented using the data structures in Figure 2.13:

mutex for mutual exclusion.

```
typedef struct TAGsstf_queue_t {
  struct phdskinfo *disk;
  /**/
  __b_fastmutex_t mutex;
  struct request_prologue *actual;
  struct request_prologue *lqueue;
  struct request_prologue *hqueue;
  int counter;
} sstf_queue_t;
struct request_specific {
  void *next;
};
```

Table 2.13: struct sstf_queue_t

```
typedef struct TAGlook_queue_t {
  struct phdskinfo *disk;
  /**/
  __b_fastmutex_t mutex;
  int dir;
  struct request_prologue *queue[2];
  int counter;
} look_queue_t;
struct request_specific {
  void *next;
};
```

Table 2.14: struct look_queue_t

actual the current request (that is returned by bqueue_getrequest()).

lqueue e hqueue two lists ordered in decreasing and increasing order: when a new request arrives, the request is inserted on one of the two queues depending on the value of the current request cylinder. The current request will become the nearest to the current cylinder. Note: This algorithm can cause starvation.

counter The request counter.

Then, the struct request_specific contains a pointer that is used to link the list.

2.3.3 LOOK

This scheduling policy is implemented using the data structures in Figure 2.14:

mutex for mutual exclusion.

dir Is the looking direction: 0 ascending, 1 descending.

queue two ordered lists (one ascending, one descending). queue[dir] is the queue used. When the list becomes empty, dir=!dir. a new request is inserted in the current queue if its position is after/before the actual position (otherwise, it is inserted in the other list).

counter the request counter.

This algorithm can be implemented in two variants:

```
typedef struct TAGclook_queue_t {
  struct phdskinfo *disk;
  /**/
  __b_fastmutex_t mutex;
  int act;
  struct request_prologue *queue[2];
  int counter;
} clook_queue_t;
struct request_specific {
  void *next;
};
```

Table 2.15: struct clook_queue_t

```
typedef struct TAGbd_edf_queue_t {
  struct phdskinfo *disk;
  /**/
  __b_fastmutex_t mutex;
  struct request_prologue *queue;
  int inservice;
  int counter;
} bd_edf_queue_t;
struct request_specific {
  void *next;
  long dl;
};
```

Table 2.16: struct look_queue_t

1. Requests with position = current position are inserted in the current queue (starvation can happen).
2. Requests with position = current position are inserted in the queue not currently used. This is the default choice; to modify it, you have to modify the look.c file.

2.3.4 CLOOK

This scheduling policy is implemented using the data structures in Figure 2.15:

mutex for mutual exclusion.

dir This is the current queue.

queue Two ascending lists. queue[dir] is used until it is emptied. Then, dir is changed. A new request is inserted in the current queue if the current position is greater than the current one.

counter the request counter.

This implementation can introduce starvation because insertions at the current position are done on the same queue.

2.3.5 EDF

This is the only real-time disk scheduling algorithm implemented in S.Ha.R.K. This scheduling policy is implemented using the data structures in Figure 2.16:

```
typedef struct {
  RES_MODEL r;
  TIME dl;
} BDEDF_RES_MODEL;
#define BDEDF_res_default_model(res)
#define BDEDF_res_def_level(res,l)
#define BDEDF_res_def_dl(res,dl)
```

Table 2.17: Resource Module for deadlines

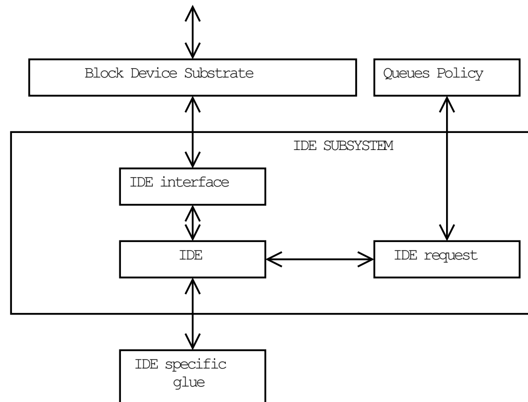

Figure 2.3: The IDE subsystem

mutex For mutual exclusion.

queue An ordered list, increasing with the deadline.

inservice This flag is set when a request is in service.

counter The request counter.

The request's deadline is got during task creation using the S.Ha.R.K. resource module shown in Figure 2.17.

If the resource module is not used, the deadline is set to infinity. That is, in that case the scheduling algorithm become a FCFS.

2.4 Device driver: the IDE interface

2.4.1 Description

A description of the IDE standard can be found in[**?**] and [**?**]. This interface allows the connection of up to 2 IDE peripherals using the ATA or ATAPI protocol (as described into[**?**]). This driver implements the ATA-4 protocol[1].

2.4.2 Implementation

The architecture of the IDE subsystem is shown in Figure 2.3: there is a part of the interface responsible for the system initialization (for example, for finding the IDE interfaces in the system, for finding the devices and their logical structure, for transforming the low level requests in ATA-4 commands), for command handling and another part for interfacing with the disk scheduling

[1]The original Massy's thesis have a small description of the standard, that I have removed.

```
typedef struct ide_parms {
  void *parm_initserver;
} IDE_PARMS;
struct ide_server_model {
  TIME dl;
  TIME wcet;
  TIME mit;
};
```

Table 2.18: Data structures used to initialize the IDE driver.

```
void ide_glue_send_request (int ideif);
int ide_glue_activate_interface (int ideif);
void ide_glue_unactivate_interface (int ideif);
```

Table 2.19: The glue code for the IDE driver.

queue handling. Finally, there is a small part OS-dependent that should be rewritten to port it to other OSes.

If possible, the driver uses the DMA; in any case, polling can be forced. DMA is only supported in bus-master mode [**?**].

The system actually use only some commands of the classes "PIO data in", "PIO data out", "Non-data" e "DMA"; it uses the "read", "write", "seek", "dma read", "dma write", "identify", "pidentify" and "set feature" commands; packet commands are not currently supported, so ATAPI devices (like cd-roms) are currently not supported.

2.4.2.1 Initialization

The system is initialized calling the ide_init() function into bdev_init(). This is automatically done if the IDE support is compiled. The user should only initialize the struct bdev_parms, as specified in Section 2.1.

The IDE_PARMS structure has only one parameter, and is included into the ideparms field in the global init data structure:

parm_initserver This parameter is OS-specific and is related to the glue code.

The system dependent part of the IDE driver is contained into the file ideglue.c (see Figure 2.19):

ide_glue_activate_interface It enables the interface passed as parameter: It creates an aperiodic task, and assigns to it the interface interrupt. The task is activated when the interrupt arrives.

ide_glue_activate_interface It disables the interface. The aperiodic task is killed.

ide_glue_send_request It executes the first available request: it explicitly activates the task interface; when the task is activated because of an interrupt, the pending request is terminated, and another request (if present) is activated.

the parm_initserver field contains a pointer to the struct ide_server_model, used to create the aperiodic tasks (see Figure 2.18):

dl task deadline in μsec.

wcet task worst case execution time in μsec.

mit the mean inter arrival time.

```
typedef struct TAGidereq_t {
  request_prologue_t info;
  int next;
  __uint8_t resetonerror;
  __uint8_t cmd;
  __b_sem_t wait;
  int result;
  __uint8_t features;
  __uint8_t cyllow;
  __uint8_t cylhig;
  __uint8_t seccou;
  __uint8_t secnum;
  __uint8_t devhead;
  __uint8_t *buffer;
} idereq_t;
```

Table 2.20: struct idereq_t

If these parameters are not specified, standard parameters are used.

These defines are used to conditionally compile the driver (see the source code):

FORCE_SCANNING this define force the search of the IDE peripherals also if the bus reset fails. This is useful if some peripherals does not follow the standard timings.

IDE_OLDATAPIDELAY this define forces a little delay that can be useful when using old ATAPI interfaces.

IDE_FORCERESET if defined a soft reset operation cannot be aborted because of a timeout. It must be specified if the peripheral does not follows the soft reset timings or if it cannot go in stand-by mode.

IDE_SKIPSTATUSTEST with this symbol we do not check if the command is being executed correctly (that is done reading on a status register); all the TX-based motherboard we found needs this symbol.

IDE_SKIPALLTEST when set, all the command are written into the registers without looking on the status register. This symbol implies the previous. This mode was needed on a TX motherboard where the polled mode for commands was unreliable.

2.4.2.2 Policy queues Interface

Table 2.20 shows struct idereq_t that is used by the driver to store the pending requests. It can be noted that the info field is of type request_prologue_t, as required by the modules that handle the scheduling policies (see 2.3).

Note that (since the IDE interface allows only a command at a time) there are two queues for each IDE interface (one for the master device, one for the slave). These request queues are prioritized. That is, the master queue always have priority on the slave queue.

Here a small description of the fields showed in Table 2.20:

info To handle the requests using a module for disk scheduling.

next used internally to handle a request pool.

resetonerror If set, an error during a command implies a soft reset.

cmd The command for the peripheral (see [?]).

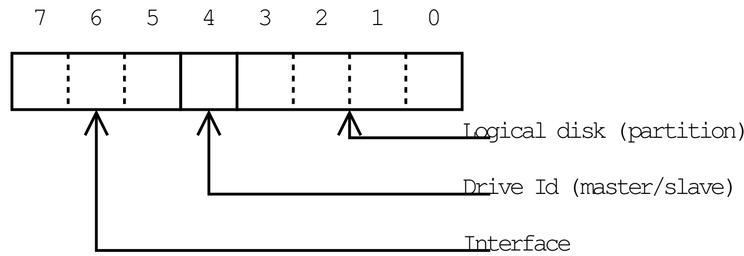

Figure 2.4: IDE minor numbers

wait Synchronization semaphore: when a task sends a command, it waits the end of the execution of the command.

result This is the result: 0 in case of success, != 0 in case of error (see idelow.c for the error codes).

features, cyllow, cylhigh, seccou, secnum e devhead Values to be inserted into the I/O registers of the IDE interface. The behavior is different depending on the peripheral mode (LBA or C/H/S) (see Section 2.4.1).

buffer read/write buffer (if needed).

2.4.2.3 Minor numbers

This section describes how minor numbers are handled.

As described in Section 2.1.4, the generic low level part uses the major number to pass a request to a device. The specific device is then selected using the minor number.

Figure 2.4 shows the minor number structure of this device:

- The first 3 bits identifies the interface (8 interfaces maximum).
- A bit for the peripheral (master/slave). This is used to enqueue request in the right queue.
- The last 4 bits says which partition should be used. This info is used to convert relative sector numbers into absolute sector numbers.

Minor numbers are accessed only via macros. To modify the minor number mapping you should only modify these macros.

Chapter 3

File system

With the word "file system" we mean the internal organization of a block device; the purpose of such an organization is to store a set of files, usually divided in directories.

A block device is seen by the file system as an ordered sequence of fixed sized blocks. The basic operations of a file system are read/write operations on the n-th block.

3.1 Interface

This section describes the interface between the system and the user tasks. It also describes the interface between a specific instance of file system and the system (just to make easy the expansion of the system with new real-time file systems).

3.1.1 Initialization

To initialize the higher level of the system we use an interface similar to the device drivers interface (see Section 2.1.1.

The function filesystem_init() is called to initialize the higher layer passing the parameters showed in Figure 3.1. These parameters must be initialized using the following macros:

filesystem_def_rootdevice Set the partition that must be used as *root device*, that is, the directory associated to "/". This field is mandatory, and that implies that the higher layer must be initialized *after* the device drivers.

filesystem_def_fs Selects the filesystem that must be used with the device. It must be one of the symbols defined into include/fs/fsind.h; if the special symbol FS_DEFAULT is used,

```
typedef struct filesystem_parms {
  __dev_t device;
  __uint8_t fs_ind;
  __uint32_t fs_showinfo:1;
  void *fs_mutexattr;
  struct mount_opts *fs_opts;
} FILESYSTEM_PARMS;
#define filesystem_def_rootdevice(par,rootdev)
#define filesystem_def_fs(par,fs)
#define filesystem_def_showinfo(par,show)
#define filesystem_def_options(par,opts)
int filesystem_init(FILESYSTEM_PARMS *);
```

Table 3.1: struct filesystem_parms

```
struct mount_opts {
  __uint32_t flags;
  union {
  struct msdosfs_parms msdos;
  } x;
};
/* Flags */
#define MOUNT_FLAG_RW 0x01
```

Table 3.2: struct mount_opts

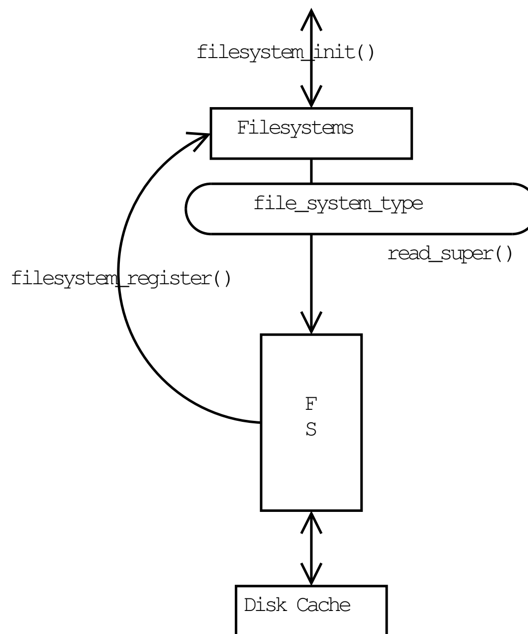

Figure 3.1: Filesystem init

the system tries to autodetect the filesystem type.

filesystem_def_showinfo If this flag is set, then initialization messages are printed on the screen.

filesystem_def_options There are the mount options.

The field fs_mutexattr has the same meaning of the correspondent field into bdev_parms (see Section 2.1.1).

You can use the macro BASE_FILESYSTEM to initialize a struct filesystem_parms with some default values. NULL is not allowed as a parameter of filesystem_init.

The mount options for every device must be inserted into a struct mount_opts showed in Table 3.2. The structure is composed by two parts: a general part and a filesystem-specific part. The general part actually contains the following fields:

flags This field contains the mount fields. The only flag currently defined is MOUNT_FLAG_RW, that allows write operations on the file system (default is read-only).

Filesystem initialization proceeds as shown in Figure 3.1:

1. The user calls filesystem_init();

```
struct file_system_type {
  const char *name;
  int fs_flags;
  __uint8_t fs_ind;
  struct super_block *(*read_super) (
    __dev_t device,
    __uint8_t fs_ind,
    struct mount_opts *options
  );
};
```

Table 3.3: struct file_system_type

```
struct super_operations {
  int (*init_inode) (struct inode *);
  int (*read_inode) (struct inode *);
  int (*write_inode) (struct inode *);
  int (*put_super) (struct super_block *);
  int (*delete_inode) (struct inode *);
};
```

Table 3.4: struct super_operations

2. filesystem_init() initializes itself and its modules, then all the file systems are initialized. filesystem_init() have to be changed if a new filesystem is introduced.

3. The file systems initialization functions have to register themselves calling the function filesystem_register() with a struct file_system_type.

3.1.1.1 Struct file_system_type

This structure (see Table 3.3)stores the informations needed for the filesystem registration. Here a short description of its fields:

name Pointer to the filesystem name (e.g., "FAT16").

fs_flags Filesystem flags (currently not used).

fs_ind Filesystem ID (every filesystem has a different ID).

read_super This the filesystem initialization function. It is called when a partition is mounted to initialize the correspondent struct super_block.

3.1.2 Internal interface

This section describes the interface between the high layer (that handles the file systems), and the modules that implements a specific filesystem.

When a user mount a device into a directory using a filesystem, the system uses the file system's specific struct file_system_type, calling the function read_super() to read the superblock of the specified device.

The struct super_block contains a pointer to a structure that contains a set of "virtual operations" (see Section 3.2), that implements the behavior of a specific filesystem.

```
struct dentry_operations {
  int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
};
```

Table 3.5: struct dentry_operations

```
struct inode_operations {
  struct file_operations *default_file_ops;
  struct inode* (*create) (struct inode *,
    struct dentry *);
  struct inode* (*lookup) (struct inode *,
    struct dentry *);
  int (*unlink) (struct dentry *);
  void (*destroy) (struct inode *);
  int (*truncate) (struct inode *,
    __off_t len);
};
```

Table 3.6: struct inode_operations

3.1.2.1 struct super_operations

The structure in Table 3.4 is used for super block manipulation. All the functions should return 0 in case of success, or a value != 0 otherwise.

init_inode This function initializes a struct inode passed as parameter. That structure must have the fields i_sp and i_st.st_ino already initialized (see Section 3.2.1.3).

read_inode This function reads the inode passed as parameter from the filesystem. The inode structure must have the fields i_sp and i_st.st_ino already initialized.

write_inode This function writes a (valid, full initialized) inode into the filesystem.

delete_inode This function removes the inode from the filesystem (the inode must already exist on the filesystem).

put_super This function writes the super block passed as parameter into the filesystem. This function is called when a filesystem is unmounted. A dual function (that is, a function that reads a super_block from a filesystem) (which is only used when mounting the filesystem) can be found into the struct file_system_type (see Section 3.1.1.1).

3.1.2.2 Struct dentry_operations

This structure contains all the functions used to manipulate a struct dentry.

d_compare This function compares the two strings passed as parameter. it returns 0 if the two strings are equal, or != 0 otherwise. The filesystem do not use directly the strcmp() function, because filename comparison depends on the filesystem (e.g., case sensitive, case insensitive).

3.1.2.3 struct inode_operations

This struct contains the virtual operations on the struct inode. These functions returns 0 in case of success, !=0 in case of error, that will be interpreted as a negate errno() value (e.g., -EPERM instead of EPERM).

```
struct file_operations {
  __off_t (*lseek) (struct file *,
    __off_t, int);
  __ssize_t (*read) (struct file *,
    char *,
    __ssize_t);
  __ssize_t (*write) (struct file *,
    char *,
    __ssize_t);
  int (*readdir) (struct file *,
    void *);
  int (*open) (struct inode *,
    struct file *);
  int (*close) (struct inode *,
    struct file *);
};
```

Table 3.7: struct file_operations

default_file_ops these are the file_operations described into the paragraph 3.1.2.4, that must be used with the inode. By default, the field is initialized with the file_operation of its filesystem. These pointers can be modified to implement a kind of virtual filesystem that works on a device. For example, we can define an inode "/dev/ide/hda1" in a way that all read/write operations on it are not mapped in the operations of the filesystem where it is stored.

create This function creates a new inode (a new file) with the name passed as parameter into the struct inode passed as parameter (that must be an inode of type "directory"). It returns the new inode or NULL in case of error.

lookup This function searches an inode passed as parameter into another inode of type directory passed as parameter. It returns the inode just found or NULL in case of error.

unlink This function unlinks the filename passed as parameter from the parameter of the corresponding inode. The inode will be removed (by generic filesystem routines) if it remains without links associated to it.

destroy This function destroys the file linked to the inode. It frees the space occupied by the file. Often, it is sufficient to call the truncate function with length 0.

truncate This function truncates the file length of the inode passed as parameter at the length specified in the second parameter. If the given length is greater than the actual file length, the function does nothing.

3.1.2.4 struct file_operations

This structure contains all the function that must work with files. If not otherwise stated, the functions can operate both on files and on directory.

lseek Moves the input/output position of the file specified as parameter. The new position is specified as in the lseek primitive (described in [**?**] or in all the Unix programming books). That is, the position is specified with a delta with respect to a given position (start of file, end of file, current position); As described in section 3.2.1.4 during the description of the field f_pos, the new position can be greater than the end-of-file. This function must return a position inside the file or an error code.

```
int mount (dev_t device,
  u_int8_t fsind,
  char *where,
  struct mount_opts *options);
int umount (dev_t device);
dev_t fdevice (char *device_name);
```

Table 3.8: mount/umount functions

read Reads some data and puts it into the buffer. The function is called only for regular files. It returns the number of bytes read, that can be less than requested, or an error code.

write Writes some bytes from the specified buffer. It is called only for regular files. It returns the number of bytes written, that can be less than requested, or an error code.

readdir Reads the next filename of the directory parameter passed through a struct file. It inserts it into the struct dirent passed as parameter. Returns 0 in case of success, a negative number in case of error.

open Opens the file linked to the inode. Fills some internal fields in the filesystem-specific struct file. Returns 0 in case of success, an error otherwise.

close Closes the file passed as parameter. returns 0 in case of success, an error in case of failure.

Most of the function described in this section works on buffers passed by the user. the general routines checks that these buffer are corrects. In case the operating system implement some memory protection mechanisms, the functions copyfromuser() and copytouser() should be used (their syntax is similar to memcpy()). Note that Shark currently do not provide memory protection, so memcpy() can be used.

3.1.3 External interface

3.1.3.1 Non standard functions

The mount/umount functions are the non-standard functions provided by the filesystem (their prototype can be found in include/sys/mount.h). Here is a small description of these functions (see Table 3.8):

fdevice This function returns the device serial number of a given filename. See Section 2.1.2.

mount Is used to mount a partition. These are the informations that must be passed to it:

- The device serial number that must be mounted
- The filesystem identifier (fsind) that should be used (see include/fs/fsind.h).
- The directory where the device will be mounted.
- Other parameters (see Section 3.1.1).

The function returns 0 in case of success, != 0 in case of failure (errno is modified).

umount This function unmount a device that was previously mounted. This function cannot unmount the "root" device. During shutdown, all the filesystem will be unmounted in the proper order.

3.1.4 Initialization: an example

This is an example of the filesystem initialization in S.Ha.R.K. To initialize the system, we must register the kernel modules first:

```
TIME __kernel_register_levels__(void *arg)
{
  struct multiboot_info *mb=(struct multiboot_info*)arg;

  EDF_register_level(EDF_ENABLE_ALL);
  RR_register_level(RRTICK, RR_MAIN_YES, mb);
  CBS_register_level(CBS_ENABLE_ALL, 0);
  dummy_register_level();

  SEM_register_module();
  CABS_register_module();
  NOPM_register_module();
  return TICK;
}
```

The filesystem always require the registration of a module that can accept soft aperiodic tasks (such as the CBS, for example). Moreover, a mutex protocol must be present (NOPM, in this case).

```
TASK _{_{i}}nit_{_{(}}void *arg)
{
  extern int __bdev_sub_init(void);
  extern int __fs_sub_init(void);
  struct multiboot_info *mb=(struct multiboot_info*)arg;

  /* block devices initialization */
  __bdev_sub_init();
  /* filesystem initialization */
  __fs_sub_init();

  __call_main__(mb);
  return (void *)0;
}
```

This is a typical S.Ha.R.K. __init__ function. In this function, block devices and file systems should be initialized.

```
int __bdev_subinit(void)
{
  extern __dev_t root_device;
  BDEV_PARMS bdev=BASE_BDEV;

  /* low level initialization */
  bdev_def_showinfo(bdev,TRUE);
  bdev_init(&bdev);

  /* root device specification */
  root_device=bdev_scan_devices(choose_root_callback);
  if (root_device<0) {
```

```
    /* in caso di errore...  */
    sys_end();
    return -1;
  }

  return 0;
}
```

This function initializes the block device layer. This function sets the parameters of the bdev variable, calls the low level initialization functions and searches for the *root device*; basically, the function must specify in some way which is the root device. In the function, the low level layer lists all the available devices, calling the function choose_root_callback() for each of them. That function will choose the root device.

```
int choose_root_callback(dev_t dev,u_int8_t fs)
{
  if (fs==FS_MSDOS)
    return dev;
  return -1;
}
```

In this case, the choice of the root device is quite simple: the first FAT16 MSDOS partition becomes the root device. Usually, MSDOS calls that partition “C:”.

```
int __fs_sub_init(void)
{
  extern __dev_t root_device;
  FILESYSTEM_PARMS fs=BASE_FILESYSTEM;
  struct mount_opts opts;

  /* mount parameters */
  memset(&opts,0,sizeof(struct mount_opts));
  opts.flags=MOUNT_FLAG_RW;

  /* filesystem initialization */
  filesystem_def_rootdevice(fs,root_device);
  filesystem_def_fs(fs,FS_MSDOS);
  filesystem_def_showinfo(fs,TRUE);
  filesystem_init(&fs);

  /* C library initialization */
  libc_initialize();

  return 0;
}
```

This functions initializes the filesystem giving R/W permission on the root partition previously found. Then, the C library is initialized.

3.2 Internal structure

Many data structures are composed by two parts: a generic part, that is independent from the particular filesystem, and a filesystem specific part, that are usually stored into an union.

```
struct super_block {
  struct super_block *sb_parent;
  struct super_block *sb_childs;
  struct super_block *sb_next;
  __dev_t sb_dev;
  struct inode *sb_root;
  struct dentry *sb_droot;
  struct file_system_type *sb_fs;
  struct super_operations *sb_op;
  struct dentry_operations *sb_dop;
  struct mount_opts sb_mopts;
  int sb_busycount;
  __uint32_t sb_used:1;
  __uint32_t sb_blocked:1;
  union {
    struct msdos_super_block msdos_sb;
  } u;
};
```

Table 3.9: struct super_block

Almost all the data structures have a set of functions that modifies them. Pointers to these functions are included in a structure that has the name of the structure that is modified by the functions with a suffix "operations". For example, the structure super_block (see Section 3.2.1.1) has a structure super_operations (see Section 3.1.2.1) that contains a function create_inode.

Since the high layer functions influences the global system throughput, some informations are duplicated in the data structures to speed up the execution.

The system is basically structured with a Unix-like approach, with 4 fundamental structures:

super contains informations on each device that is mounted (a table is used);

inode contains informations on the files in a filesystem (double lists with hash keys are used);

dentry contains the file names (a tree is used);

file contains informations on each open file (a descriptor table is used).

3.2.1 The data structures

3.2.1.1 struct super_block

From a user point of view, the filesystem exports a unix-like view of the system. That is, partitions can be mounted into the directory tree. When the system starts, the root directory is mounted as "/".

The struct super_block contains the fundamental information of the filesystem (a unix-like filesystem usually duplicates this informations a few times into the Hard disk). Every time a partition is mounted a new structure is allocated and initialized. These structures are maintained in a tree, that is, if a partition is mounted into a directory the new structure becomes a leaf of the structure that contains the mount directory.

The root super block is stored into the global pointer root_superblock; Figure 3.2 shows a possible super block tree.

Here a short description of the struct super_block fields (see Table 3.9):

sb_parent, sb_childs, sb_next are used to store the tree structure: sb_parent points to the father of the structure, whereas sb_childs points to the first structure of the childs; sb_next is used to create the brothers list (that is the children list of the father).

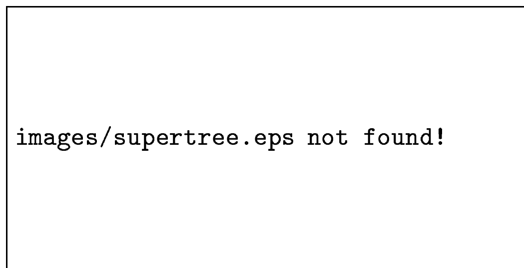

Figure 3.2: Super block tree

sb_dev Contains the device, that is the partition that contains the filesystem (the low level part is responsible of the devices, that are typically associated to an hard disk partition. It is possible for example to extend the low level to associate to a device other entities, such a file on another filesystem (this is usually possible on Unix system through loopback devices).

sb_root This is the root directory inode of the filesystem.

sb_droot This is the directory entry (the filename) of the root directory.

sb_fs This is the pointer to the filesystem present in the filesystem. Every supported file system is registered at init time (see Section 3.1.1).

sb_op Pointer to a structure that contains the specific filesystem functions used to use the struct super_block.

sb_dop These are the specific function used for the directory entries.

sb_mopts Mount options used during initialization.

sb_busycount This is a counter that counts the number of entities that are using the filesystem (e.g., a partition can be unmounted if someone is reading it).

sb_used A flag that signals if the structure is used.

sb_blocked Some operations on this structure must be executed in mutual exclusion; if this flag is set there is some operation on the structure. This flag can not be used/tested/modified directly by the user functions.

3.2.1.2 struct dentry

The struct dentry contains the file names independently from the file type (regular file, directory, ...)[1]. Each file in the filesystem is always stored in two structures:

- struct dentry contains the filename;
- struct inode contains the other informations on the file.

This separation is made because in most file systems (not in the MSDOS FAT16) more than one name can be linked to the same file (inode).

These data structures are stored internally in a tree structure. Please note that the inodes are not structured as a tree; every dentry has a pointer to the correspondent inode.

Figure 3.3 shows a directory entry tree. Directories are showed with a continuous line, whereas files are showed with a dashed line (note that the internal structures for files and directories are the same).

[1] "dentry" means *D*irectory *ENTRY*.

```
struct dentry {
  struct dentry *d_next;
  struct dentry *d_prev;
  struct dentry *d_parent;
  struct dentry *d_child;
  __time_t d_acc;
  struct qstr d_name;
  short d_lock;
  struct dentry_operations *d_op;
  struct inode *d_inode;
  struct super_block *d_sb;
};
```

Table 3.10: struct dentry

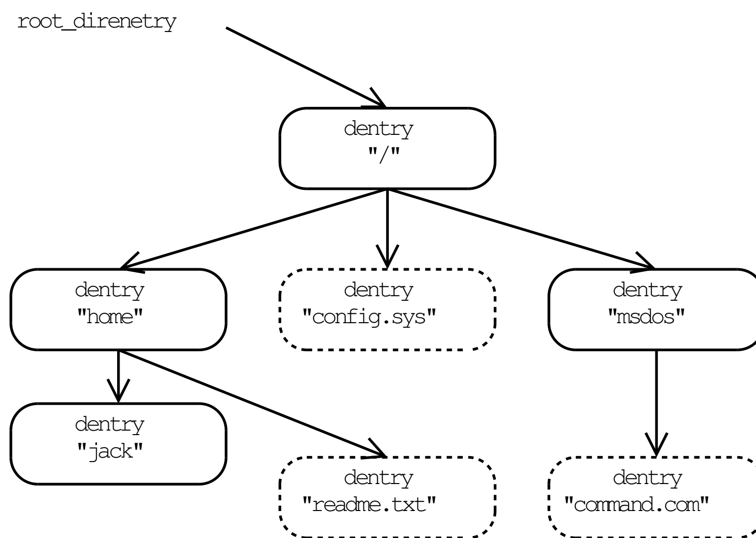

Figure 3.3: Directory entry tree.

```
struct qstr {
  char name[MAXFILENAMELEN+1];
  char *nameptr;
};
```

Table 3.11: struct qstr

The system do not load all the file names in the filesystem. Only the recent filenames are stored in memory in a partial tree. These informations are continuously updated removing and adding dentry struct.

Directory names are stored into a struct qstr (see Table 3.11). A string is not used to avoid too much sting copies. The structure is composed by two fields:

nameptr If != NULL, it is the name to use.

name If nameptr==NULL, it is the name to use.

The QSTRNAME can be used on a structure QSTR to get a (char *). When filling the structure, one of the two fields can be used. If nameptr is used, the pointed string must be statically allocated. If name is used, nameptr must be set to NULL.

The fields of the struct dentry contains the following informations:

d_next, d_prev, d_parent e d_child can be used to store a dentry tree structure: d_parent is a pointer to the father's dentry, d_child is a pointer to a list of children dentry, d_next and d_prev are used to navigate into the children dentry list.

d_acc It is the time of the last access to the dentry structure.

d_name Is the name associated to the dentry.

d_lock is a blocking counter: it is incremented every time a routine needs to use the dentry, and it is decremented when it is no more needed. In that way, it is possible to know which are the structures currently in use. The filesystem routines should not directly modify this field.

d_op Pointer to the virtual operations used to handle this structure.

d_inode is the inode associated to the structure.

d_sb It is the super_block that contains the directory. This a redundant information used to speed-up the code.

3.2.1.3 struct inode

An inode is a file into a filesystem. It contains all the file informations except the name that is maintained into a struct dentry (see 3.2.1.2; more details can be found into [?]).

All the inodes temporarily present in memory are stored into an hash bucket structure as showed in Figure 3.4: for each device a number is computed using an hash function, the inode number (file serial number) and the device number. All the inodes with the same hash entry are stored in a list. The first inode in the list is stored in a table at the index given by the hash key.

An inode contains the following fields:

i_st Contains all the standard informations for a file (e.g., the length, the type); a task can inspect these informations using the stat() primitive. These informations are included into a stat structure that is shared between the user libraries and the internal implementation, in a way that a memcopy should be enough to pass these informations.

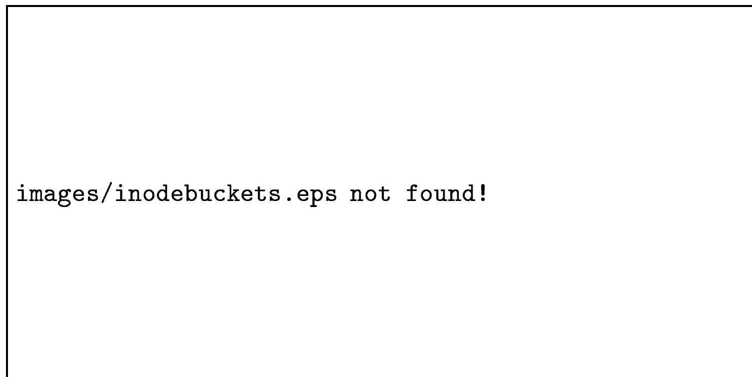

Figure 3.4: Inode data structures

```
struct inode {
  struct stat i_st;
  struct inode *i_next;
  struct inode *i_prev;
  int i_hash;
  __rwlock_t i_lock;
  __uint16_t i_dlink;
  __uint32_t i_dirty:1;
  struct inode_operations *i_op;
  struct super_block *i_sb;
  union {
    struct msdos_inode_info msdos_i;
  } u;
};
```

Table 3.12: struct inode

```
struct stat {
  __dev_t st_dev;
  __ino_t st_ino;
  __mode_t st_mode;
  __nlink_t st_nlink;
  __uid_t st_uid;
  __gid_t st_gid;
  __off_t st_size;
  unsigned long st_blksize;
  __time_t st_atime;
  __time_t st_mtime;
  __time_t st_ctime;
};
```

Table 3.13: struct stat

```
struct file {
  struct file *f_next;
  struct dentry *f_dentry;
  struct file_operations *f_op;
  __loff_t f_pos;
  unsigned int f_count;
  __uint32_t f_flag_isdir:1;
  union {
    struct msdos_file_info msdos_f;
  } u;
};
```

Table 3.14: struct file

i_next, i_prev e i_hash These fields are used to implement the double linked list and the hash key: i_next and i_prev implement a double list, whereas i_hash is the hash key (redundant information).

i_lock It is a lock for the structure (see Section 3.2.2.3).

i_dlink Number of directory entries that points to the inode (see Section 3.2.1.2).

i_dirty A flag that says if this inode has been modified.

i_op Virtual operations used for working with this inode. Usually they are filesystem specific.

i_sb Super block where the inode is stored (see Section 3.2.1.1).

The struct stat showed in Table 3.13, contained into the struct inode and described in the paragraph 5.6 of [?], contains the following fields:

st_mode It is the file mode (informations about who can read/modify/execute a file).

st_ino An unique number that identifies a file ("file serial number" in the Posix terminology).

st_dev It is the device serial number; st_dev and st_ino are necessary and sufficient to uniquely identify a file in the system.

st_nlink Number of hard links to a file (Always 1 for FAT16 file systems).

st_uid e st_gid File group and user identifiers. 0 is used for the system administrator. The FAT16 filesystem always set these fields to 0.

st_size File length (bytes).

st_atime, st_mtime e st_ctime File times: st_atime is the time of the last access, st_mtime is the time of the last modification and st_ctime is the time of the last state change (e.g., when the file has been created or modified using chmod).

3.2.1.4 struct file

Every open file has a struct file associated to it (see Table 3.14).

f_next Pointer used internally to the filesystem modules.

f_dentry contains the filename (see Section 3.2.1.2). From here we can get the corresponding inode number.

f_op Contains a pointer to the virtual operation that works on struct file, described in Section 3.1.2.4.

```
struct file_descriptors {
  struct file *fd_file[MAXOPENFILES];
  __uint16_t fd_flags[MAXOPENFILES];
  __uint32_t fd_fflags[MAXOPENFILES];
  __mode_t fd_umask;
  char fd_cwd[MAXPATHNAMELEN+1];
  struct dentry *fd_cwden;
  __mutex_t fd_mutex;
};
```

Table 3.15: struct file_descriptors

f_pos Contains the current position into the file. read/write operations are done starting from this point; moreover, Posix specification allow a seek to a position beyond file termination: in that case read operation should fail, whereas write operation should fill the gap with zeroes.

f_count Contains the number of file descriptors that are using this structure for input/output operations on the file.

f_flag_isdir Flag that says if this is a directory (redundant information).

msdos_f FAT16 specific informations.

The applications use a file descriptor to index the file to use. A file descriptor is a number meaningful only for the processor that obtained it (e.g. through an open operation) that is assigned to a struct file through a descriptor table. Every process has a different table, contained into the struct file_descriptors described in section 3.2.1.5. The whole S.Ha.R.K. Kernel can be considered a monoprocess multithread application, and for that reason it has only one descriptor table.

3.2.1.5 struct file_descriptors

This struct is used to store all the general parameters assigned to a process (e.g., the current directory). Again, S.Ha.R.K. has only one structure because it can be considered like a monoprocess multithread system. A mutex is used to guarantee mutual exclusion when the struct is used by more than one thread. The structure contains the following fields:

fd_file This is the descriptor table; every descriptor (with value between 0 and MAXOPENFILES) is associated an element that can have the following values:

- NULL if it is a free descriptor;
- RESERVED if it is reserved for special uses (e.g., descriptors assigned to the keyboard or to the console)
- pointer to the file

fd_flags Flags associated to the file (e.g., FD_CLOSEXEC), described in detail in [**?**], when the fcntl() primitive is described.

fd_fflags Flag used when the file was opened (e.g., O_APPEND, O_RDONLY, ecc...).

fd_umask Mask used during file creation;can be modified with the function umask().

fd_cwd Current working directory used by all the primitives that receives a relative pathname (that does not start with "/" (redundant information).

fd_cwden The current working directory; it can be modified using the function chdir().

fd_mutex Mutual exclusion semaphore used for concurrent multithread access.

```
dcache_t *dcache_lock (__dev_t dev,
  __blkcnt_t lsector);
void dcache_unlock (dcache_t *);
dcache_t *dcache_acquire (__dev_t dev,
  __blkcnt_t lsector);
void dcache_release (dcache_t *);
void dcache_dirty (dcache_t *d);
void dcache_skipwt(dcache_t *d);
```

Table 3.16: Disk cache interface

3.2.2 Disk Cache

3.2.2.1 How to use the cache

The filesystem modules use the disk cache to get all the informations they need.

The disk cache interface is showed in Table 3.16:

dcache_lock This function can be used to require a read only access to the cache. More than one thread can lock a block. the parameters are the device number and the sector number; it returns a pointer to the struct dcache_t in case of success, or NULL in case of error. This function do not return until the access is granted or an error occurred. To avoid critical blocking times in case more than one block is needed, you must use a proper access protocol: the blocking must be done with increasing block numbers; moreover, if possible, only one block should be blocked at a time.

dcache_unlock This function can be used to unblock a cache entry previously blocked.

dcache_acquire This function behaves as dcache_lock(), but it requires a read/write access. Only one task at a time can acquire the lock. This access do not imply that the cache entry is considered dirty (see the dcache_dirty() function).

dcache_release This function is similar to dcache_unlock(), and works with the locks acquired by dcache_acquire().

dcache_dirty Signal that the cache entry has been modified. This function must be called only if we own a read/write lock and if the cache has been modified.

dcache_skipwt This function signals to the system that the cache entry contains system informations that are often modified. That means that the write_through flag should not be considered. See section 3.2.2.2 for a detailed description.

3.2.2.2 struct dcache

This data structure, used by the module that implements the data cache, represents a cache entry. All the high level functions uses the cache module as an interface to the lower layer.

These structures are maintained in the system wit a structure similar to the inodes, that is a set of double linked lists that I can access using an hash key, as showed in Figure 3.5; the hash key is computed using the device (that is the physical peripheral) and the lsector (the logical sector).

Every struct dcache is an elementary block that can be transferred to a device. It is not possible to have 2 data structures that refers to the same block. That is, the pair (device,lsector) must uniquely identify the data sector on the physical device (e.g., the block 64 in “ide/hda” could be the block 0 of “ide/hda1”, as explained in section 2.1.2).

The field of the structure (see Table 3.17) are:

prev, next e hash Are used to store the structures into the lists that are accessed using the hash key: next and prev are used to implement a double linked list, whereas hash is the hash key.

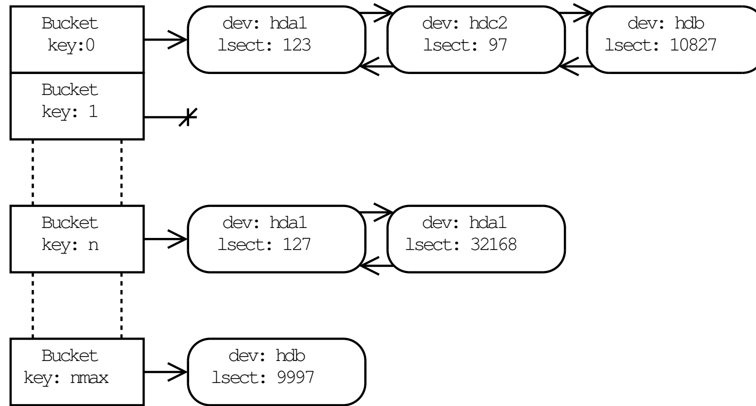

Figure 3.5: Data structure for disk cache

```
typedef struct TAGdcache {
  int prev;
  int next;
  int hash;
  __dev_t device;
  __blkcnt_t lsector;
  __uint8_t buffer[MAXSECTORSIZE];
  int used;
  __time_t time;
  int numblocked;
  __fs_sem_t sync;
  __fs_sem_t esync;
  __rwlock_t rwlock;
  __uint32_t dirty:1;
  __uint32_t ready:1;
  __uint32_t esyncreq:1;
  __uint32_t skipwt:1;
} dcache_t;
```

Table 3.17: struct dcache

device, lsector e buffer These fields contain the physical device, the logical sector and the data contained in the corresponding physical block. These are the unique fields that a high level routine should use (and that shall be provided by a data cache).

used The number of tasks that are using the block; -1 means that no one is using the block, and that the block is free.

time The last block access time.

numblocked It is the number of tasks that are waiting the availability of the block, (that is, they are waiting the flag ready to be set). That is, if a task asks for a block already asked by another task, but not yet available (not made available by the lower layer), then the task blocks on a synchronization semaphore, and it increments that field.

sync This is the synchronization semaphore used by the blocked tasks that waits the setting of the ready flag.

esync It is a synchronization semaphore on the errors needed during the cache "purging" phase (see later).

rwlock This is a locker used to require the R/W access to the block (see Section 3.2.2.3).

dirty This flag is set if the block has been modified.

ready This flag is set if the block is available.

esyncreq This flag is set if the synchronization on errors is required (that is, the field esync is used).

skipwt This flag says if the write through handling should be done.

The fields esync and esyncreq are used during the "purging phase": a task requires a read or write operation on a block, the block is not into the data cache, and the cache is full. That is, another block must be discarded from the cache. To discard a block it can happen that a disk write operation must be done. While this write operation is in progress other tasks can require the same (discarded, in writing) block. These tasks can block waiting for synchronization on the block. In that case the cache behavior could be the following: the requests on the block are aborted (the failure is only needed if we are able to free the block; the blocked tasks are awakened with an error); or, the requests on the block are not aborted (the block can not be freed, and the tasks blocked on it are woken up). The problem is that after the task has wake up all the blocked tasks, it must wait that all the awakened tasks check for an error. To do that, the task set the esyncreq flag, and it blocks on the esync semaphore; the last task that checked for errors, checks the esyncreq flag and signal the esync semaphore waking up the waiting task.

The disk cache can work in three different modes: "copy-back", "write-through" and "modified write-thought". Usually most time-sharing OSes handle a copy-back cache. that is, write operations are done only in cache; a low priority task is used to synchronize cache and disk data. In such a system, if a cache request arrives and the cache is full, a block is freed eventually writing it on the disk. On a real-time system such a behavior can give some problems, because the time spent freeing the cache can be accounted to the running task. For example, a task that only reads can be delayed by another task that fills the cache memory writing blocks on the disk. To solve this problem there are 2 solutions: the cache is not used (that is, no caching on write), or the cache behavior is set to write-through, that is the write operations are immediately synchronized on the disk, in a way that who dirties the cache is also responsible for its synchronization.

If the cache is used in write-through mode, another problem arises: sometimes a few bytes are modified frequently (e.g. the filesystem information for the allocation of a file on the disk), that leads to continuous disk writes. For that reason, the disk cache provides the flag skipwt, that forces the copy-back mode for a particular block. By default, this flag is not set. When the write-through policy must be disabled, the file system should call the function dcache_skipwt() on the cache-entry before releasing it.

```
typedef struct {
  __mutex_t mutex;
  int active_readers;
  int active_writers;
  int blocked_readers;
  int blocked_writers;
  __sem_t readers_semaph;
  __sem_t writers_semaph;
} __rwlock_t;
```

Table 3.18: struct __rwlock_t

```
void __rwlock_init (__rwlock_t *ptr);
void __rwlock_rdlock (__rwlock_t *ptr);
void __rwlock_wrlock (__rwlock_t *ptr);
int __rwlock_tryrdlock (__rwlock_t *ptr);
int __rwlock_trywrlock (__rwlock_t *ptr);
void __rwlock_rdunlock (__rwlock_t *ptr);
void __rwlock_wrunlock (__rwlock_t *ptr);
```

Table 3.19: Lockers functions

3.2.2.3 struct __rwlock_t

The cache module and other modules need a locker protocol, that is a protocol that solves the reader/writer problem: basically, there is a shared resource where some processes must read or write. These rules must be imposed by the protocol:

1. If someone is writing, no one is authorizes to read or write.
2. If someone is reading, other tasks are authorized to read, but not to write.

In literature, there exist different methods to solve the problem with different behaviors. This filesystem implements the protocol proposed into paragraph 4.2.3.1 of [?]. To replace that policy, the rwlock.c file should be modified with another algorithm, maintaining the same interface.

Table 3.18 contains the internal data structure of the implemented protocol. To implement a "locker", all the needed data must be inserted into a structure __wrlock_t. The following functions have also to be provided (see Table 3.19):

__rwlock_init Initializes the structure passed as parameter. That structure is passed to all the functions and it is used to protect a shared resource.

__rwlock_rdlock Read request (read lock); when this function returns, the calling task is authorized to read the shared resource.

__rwlock_wrlock Write request (write lock).

__rwlock_tryrdlock Conditional read request (try read lock); The function return 0 if the calling task can read, another number otherwise. At the end of the read operation (if the lock has been granted), __rwlock_rdunlock must be called.

__rwlock_trywrlock Conditional write request (try write lock); The function return 0 if the calling task can write, another number otherwise. At the end of the write operation (if the lock has been granted), __rwlock_wrunlock must be called.

__rwlock_rdunlock Read unlock; the task has finished the use of the shared resource.

__rwlock_wrunlock Write unlock; the task has finished the use of the shared resource, that is again in a consistent state.

The functions __rwlock_rdlock and __rwlock_rdunlock must always be used together. Nested calls are not allowed. The same rules apply to __rwlock_wrlock and __rwlock_wrunlock.

Please note that __rwlock_rdlock and __rwlock_wrlock can cause undefined task blocking. The other functions do not block the calling task.

3.3 Filesystem: MS-DOS (FAT16)

3.3.1 Description

A technical description of the FAT16 filesystem can be found in [?] or [?].

3.3.1.1 Internal structure

As showed in Figure 3.6, this filesystem is divided in areas:

Boot sector Contains a small program that loads the MSDOS operating system, as specified into [?]. It also contains other informations like the number of FATS, the kind of support and the dimension of the Root Directory; It can be thought as a super-block. The informations contained in this sector are showed in Table 3.20.

FAT tables This data area contains some copies of the File Allocation Table (FAT) These informations are used to know where a file can be found on the hard disk.

Root directory This block is structured as all the blocks that contains a directory, with the exceptions that it is allocated in a contiguous way and that it has a fixed size.

Data area This is the area where the files and directories are stored. The minimal allocation unit is a cluster, that is a set of contiguous sectors on the hard disk. The boot sector stores the number of sectors that compose a cluster.

3.3.1.2 The boot sector

Table 3.20 shows the boot sector structure. The boot sector is always 512 bytes large. The struct is only the first part of the sector, the second part is the boot loader. Here is a small description of some fields:

sectorpercluster stores the number of sectors included into a cluster.

sectorsperfat stores the number of sectors in the FAT

fatsnumber stores the number of FATs on the disk.

A note: when MSDOS formats (creates a FAT16 filesystem) a disk, the informations that comes from a pre-existing filesystem have precedence over the informations given by the hardware. That is, if an hard disk declares to have 8 heads, and the pre-existing filesystem has headsperdisk=13, the new file system will have 13 heads :-).

3.3.1.3 The File Allocation Table

Every FAT is organized as a table of 16 bit elements. Every element is a cluster; the element value gives informations about the cluster:

- 0 means the cluster is available.
- values between 0xfff0-0xfff7 are reserved; for example, 0xfff7 means bad (damaged) cluster.

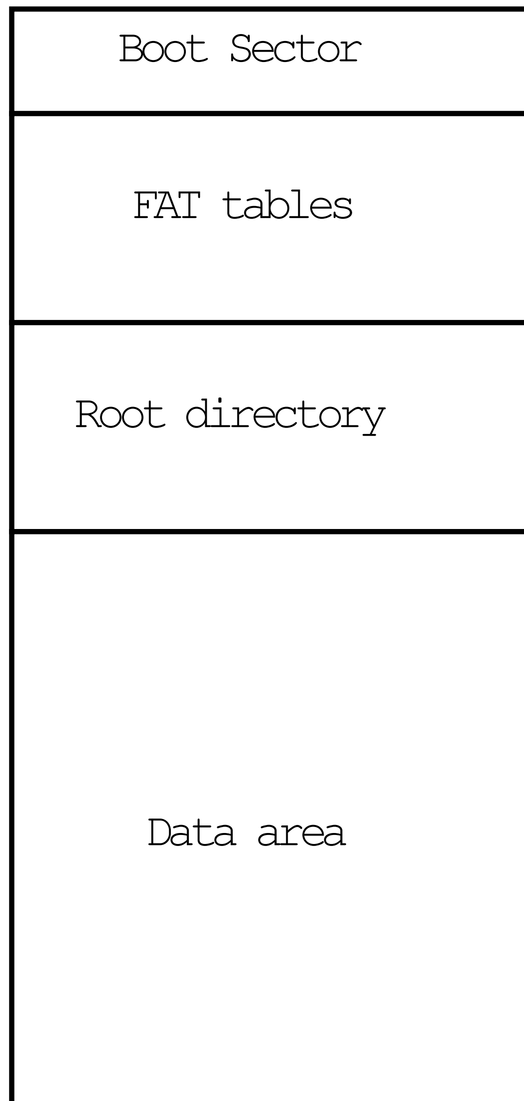

Figure 3.6: Internal structure of the MSDOS FAT16.

```
struct bootrecord {
  __uint8_t reserved[3];
  char oemname[8];
  __uint16_t bytespersector;
  __uint8_t sectorspercluster;
  __uint16_t hiddensectors;
  __uint8_t fatsnumber;
  __uint16_t rootentry;
  __uint16_t sectors;
  __uint8_t media;
  __uint16_t sectorsperfat;
  __uint16_t sectorpertrak;
  __uint16_t headsperdisk;
  __uint32_t hiddensectors32;
  __uint32_t sectors32;
  __uint16_t physicaldrive;
  __uint8_t signature;
  __uint8_t serialnumber[4];
  char volumelabel[11];
  char fattype[8];
};
```

Table 3.20: The MS-DOS boot sector.

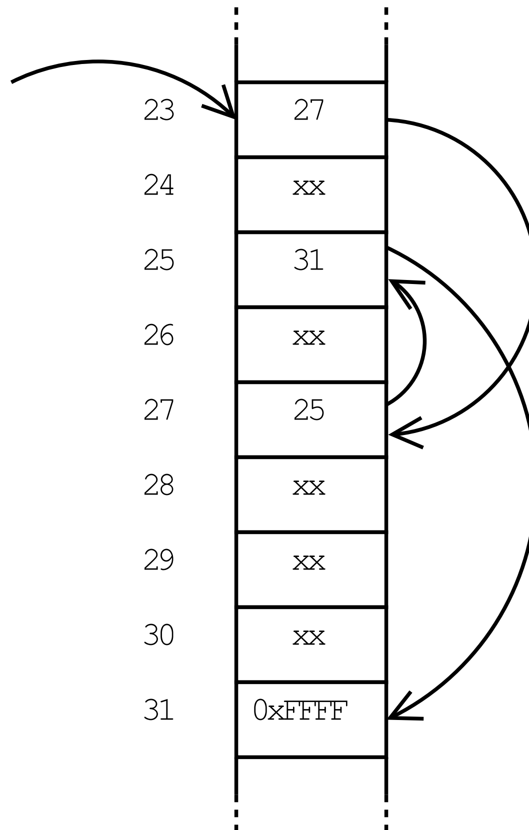

Figure 3.7: An example of File Allocation Table (FAT)

```
struct directoryentry {
  __uint8_t name[8];
  __uint8_t ext[3];
  __uint8_t attribute;
  __uint8_t reserved[10];
  __uint16_t time;
  __uint16_t date;
  __uint16_t cluster;
  __uint32_t size;
};
```

Table 3.21: MSDOS Directory Entry.

- 0xfff8-0xffff says that the cluster is the last cluster in a file.
- Every other value index which is the next cluster of the file.

The FAT is used to create a set of chains that make a file. For example, Figure 3.7 shows the information for a file stored in 4 clusters (23, 27, 25, 31). The first cluster of the chain is contained into the directories. Please note that the first cluster of the data area is the number two, because cluster 0 and 1 (the first 4 bytes) are used to store the type of FAT (12 or 16 bit), and the disk format.

Directories (except the root directory) are common files with a particular structure. The system usually give access to directories through other primitives (mkdir, creat, and so on).

3.3.1.4 The "root directory"

This area has the same structure of a directory file, except that it has fixed dimension, it is not allocated into the data area and its allocation is contiguous. Common directories have the same structure except that they are allocated into the data area, and that they can be fragmented.

The directory structure is showed in Figure 3.21:

name and ext Contains the filename in the classic format 8+3. Non used chars are filled with spaces.

attribute File flags: they specify if it is regular, a directory, a volume label, it has been archived or if it is read-only.

time Creation time (2 seconds resolution) (all the 3 times specified by the POSIX standard are mapped on that date). The time format is MSDOS specific.

date Creation date (until 2099). The date format is MS-DOS specific.

cluster The first cluster of the file. A value 0 means empty file (the first available cluster is the cluster 2).

size File size (32 bit).

Note the difference with the typical Unix file systems, where file names and file informations are stored in different areas.

Directory entries are not contiguous. If the first character name is 0xe5, the file has been deleted, if it is 0x00, it has been never used. Any other character means that the entry is valid. 0x00 is used for efficiency: if it is found, directory scan is ended also if there are other sectors in the cluster that contains a directory.

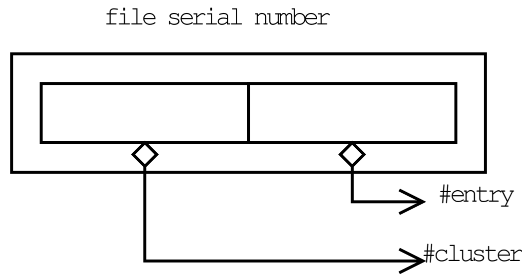

Figure 3.8: The FAT16 file serial number.

3.3.1.5 The data area

The data area is divided in clusters, that are a set of sectors. It starts from cluster number 2, and it stores regular files and directories.

linear addresses are used to identify a sector on a hard disk, and its cluster. Hard disk sectors are numbered starting from sector 0. The filesystem specifies an algorithm that associates to every linear address an address C/H/S (Cylinder/Head/Sector).

Some MS-DOS implementation supposes a particular structure depending on the media used: a 1.44 Mb floppy disk has sectors composed by 1 sector, and a root directory of 14 sectors. The best way to implement a filesystem is to rely only on the super block informations.

3.3.2 Implementation notes

The filesystem maintains 2 structures to store file informations: an inode structure and a dentry structure. In the MSDOS FAT16 filesystem unfortunately there is an unique correspondence between file names and inodes, and that information is contained into the filesystem directory entry.

When the system refers to a file, it use the inode number. That is, every file has a number that is unique inside the file system.

The FAT16 serial number has been specified as shown in Figure 3.8:

- The most significant part is the cluster number (the cluster where the directory is stored).
- The less significant part is the number into the directory entry.

For example, the inode number 0x01230008, refers to the 8th directory entry into the cluster 0x123.

The cluster 1 is used to index the directory entries contained into the root directory, whereas the cluster 0 is used to index the root directory.