MINIX VFS

Design and implementation of the
MINIX Virtual File system

Balázs Gerőfi

A master's thesis in
Computer Science

August, 2006

vrije Universiteit *amsterdam*

MINIX VFS

Design and implementation of the
MINIX Virtual File system

Balázs Gerőfi

APPROVED BY

prof. dr. Andrew S. Tanenbaum
(supervisor)

dr. Herbert Bos
(second reader)

Abstract

The Virtual File system (VFS) is an abstraction layer over the file system implementations in the operating system. It handles all system calls related to the file system and allows for client applications to access different types of file systems in a uniform way. It also provides a common interface to several kinds of file system implementations. The VFS layer was introduced first in the SunOS and it is present in many modern operating systems.

MINIX 3 is a microkernel based POSIX compliant operating system designed to be highly reliable, flexible, and secure. A minimal kernel provides interrupt handlers, a mechanism for starting and stopping processes, a scheduler, and interprocess communication. Standard operating system functionality that is present in a monolithic kernel is moved to user space, and no longer runs at the highest privilege level. Device drivers, the file system, the network server and high-level memory management run as separate user processes that are encapsulated in their private address space.

By splitting an operating system into small, independent modules, the system becomes less complex and more robust, because the smaller parts are more manageable and help to isolate faults.

This thesis describes the Virtual File system design and implementation in the MINIX 3 operating system. It also gives a comparison to other VFS designs. Exploiting modularity is a key idea behind MINIX, therefore the design of the Virtual File system layer is also driven by this idea. The result is a substantially different architecture from the Virtual File system layer in other UNIX-like operating systems.

The main contribution of this work is that the MINIX FS server was fully revised in order to divide it into an abstract layer and the actual MINIX file system driver. New data structures and methods were added to the virtual layer and modifications were realized in the actual file system implementation.

Contents

List of Figures

Listings

Chapter 1

Introduction

Reading the term ”Virtual File System” immediately raises the issue ”What is virtualization?”. *Virtualization* means different things to different people depending on the term’s context. According to the Open Grid Services Architecture Glossary of Terms [Tre04] *Virtualize* means ”Making a common set of abstract interfaces available for a set of similar resources, thereby hiding differences in their properties and operations, and allowing them to be viewed and/or manipulated in a common way”, which is a suitable definition in the file system case.

In this chapter MINIX 3 is introduced (Section 1.1) and a general description about the Virtual File system layer is given (Section 1.2). The outline of the thesis is also described (Section 1.3).

1.1 The MINIX 3 operating system

MINIX 3 is a microkernel based POSIX compliant operating system designed to be highly reliable, flexible, and secure [HBT06]. The approach is based on the ideas of modularity and fault isolation by breaking the system into many self-contained modules. In general the MINIX design is guided by the following principles:

- **Simplicity:** Keep the system as simple as possible so that it is easy to understand and thus more likely to be correct.
- **Modularity:** Split the system into a collection of small, independent modules and therefore prevent failures in one module from indirectly affecting another module.
- **Least authorization:** Reduce privileges of all modules as far as it is possible.

- **Fault tolerance:** Design the system in a way that it withstands failures. Detect the faulty component and replace it, while the system continues running the entire time.

The operating system is structured as follows. A minimal kernel provides interrupt handlers, a mechanism for starting and stopping processes, a scheduler, and interprocess communication. Standard operating system functionality that is usually present in a monolithic kernel is moved to user space, and no longer runs at the highest privilege level. Device drivers, the file system, the network server and high-level memory management run as separate user processes that are encapsulated in their private address space.

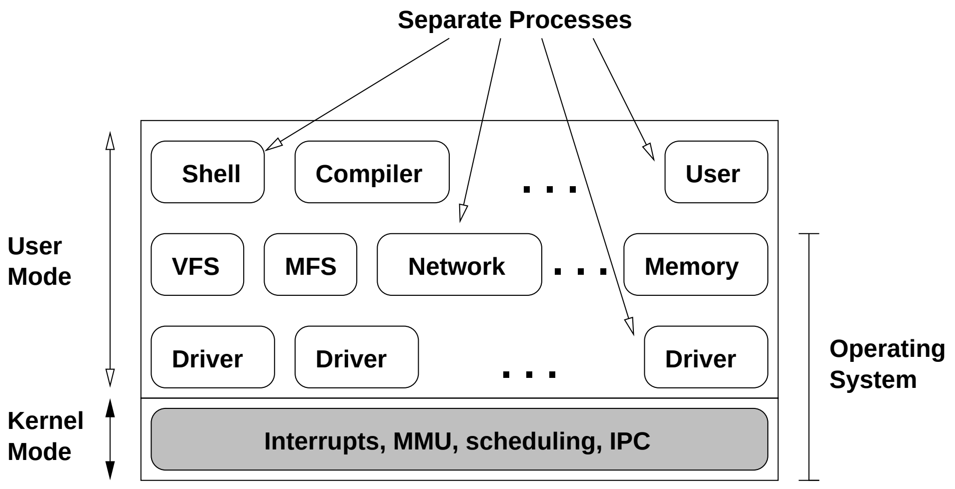


Figure 1.1: The structure of the system. The operating system runs as a collection of isolated user-mode processes on top of a tiny kernel.

Although from the kernel's point of view the server and driver processes are also just user-mode processes, logically they can be structured into three layers. The lowest level of user-mode processes are the device drivers, each one controlling some device. Drivers for IDE, floppy, and RAM disks, etc. Above the driver layer are the server processes. These include the VFS server, underlying file system implementations, process server, reincarnation server, and others. On top of the servers come the ordinary user processes including shells, compilers, utilities, and application programs. Figure 1.1 shows the structure of the operating system.

Because the default mode of interprocess communication (IPC) are synchronous calls, deadlocks can occur when two or more processes simultaneously try to communicate and all processes are blocked waiting for one another. Therefore, a deadlock avoidance protocol has been carefully devised that prescribes a partial, top-down message ordering. The message ordering roughly follows the layering that is described above. Deadlock detection is also implemented in the kernel. If a process unexpectedly were to cause a deadlock, the offending is denied and an error message is returned to the caller.

Recovering from failures is an important reliability feature in MINIX. Servers and drivers are started and guarded by a system process called the reincarnation server. If a guarded process unexpectedly exits or crashes this is immediately detected – because the process server notifies the reincarnation server whenever a server or driver terminates – and the process is automatically restarted. Furthermore, the reincarnation server periodically polls all servers and drivers for their status. If one does not respond correctly within a specified time interval, the reincarnation server kills and restarts the misbehaving server or driver.

1.2 The Virtual File System layer

As an explanation of the definition given above, the Virtual File System is an abstraction layer – over the file system implementations – in the operating system. It provides a common interface for the applications so that they can access different types of underlying file systems in a uniform way and therefore the differences in their properties are hidden. This interface consist of the file system related system calls.

The VFS also provides a common interface for the underlying file systems and manages resources that are independent from the underlying file systems. This common interface ensures that new file system implementations can be added easily.

Since the interface between the applications and the VFS is the standard POSIX interface our main concern is the design of the interface between the VFS and the actual file system implementations, which mainly depends on the functionalities of the single components.

1.3 Outline of the thesis

This document is structured as follows. The next chapter provides an overview about the MINIX Virtual File system. It covers the design principles, the components and their functionalities and gives an example in order to show the overall mechanism.

Chapter 3 discusses the VFS in details. The data structures and the operations related to them are described. It provides a general overview of the VFS/FS interface, covers the approach taken during the implementation and shows how the VFS behaves during boot-up.

Chapter 4 considers the system calls – related to the file system – and their implementation. It is organized according to the arguments of the system calls. The VFS/FS interface is detailed during the discussion of the system calls' implementation.

Chapter 5 provides a comparison between the original FS and the VFS from the performance point of view. It shows the duration of some system calls and examines the results.

Chapter 6 surveys related work in the Virtual File system topic. It describes the main evolutionary path that UNIX took from the early research editions. A brief description about the BSD and Linux operating systems from a file system perspective is provided. It also mentions the QNX file system architecture.

Finally, Chapter 7 concludes the thesis. It provides an overview of the major contributions by summarizing the results. It also describes possible areas of future work.

In the end, two appendices cover the details that did not fit in the main text. Appendix A provides a detailed description about the VFS/FS interface; Appendix B gives assistance to file system developers with an example file system implementation.

1.4 Acknowledgements

First and foremost, I wish to thank Andy Tanenbaum for giving me this great opportunity, I wish to thank for his excellent support and guideance. I would like to thank Jorrit N. Herder for his contributions and for nudging me in the right direction, I wish to thank Ben Gras and Philip Homburg for their help and constructive advice.

I wish to thank my friend Walley for the technical support and Diana for her excellent cooks. I like to thank Ma Li for her sweet smile that gave me strength all along the road.

Chapter 2

The MINIX Virtual File System

This chapter provides an overview of the MINIX Virtual File system and gives a comparison to other UNIX solutions. First the design principles are introduced (Section 2.1), then the components and their functionalities is described (Section 2.2), finally a short comparison is given to monolithic VFS designs (Section 2.3).

2.1 Design Principles

Exploiting modularity is a key idea behind MINIX, therefore the design of the Virtual File system layer is also driven by this idea. In contrast to the monolithic kernels, where the VFS layer access the implementation of the underlying file systems through function pointers, in MINIX the drivers are different processes and they communicate through IPC. During the design of the MINIX Virtual File system the most important decisions that had to be made were the followings:

- Which components are responsible for which functionalities.
- Which resources are handled by the VFS and which are handled by the actual file system implementations.
- Where to divide the former FS process in order to get an abstract virtual layer and the actual MINIX file system implementation.

Comparing the MINIX VFS to the VFS layer in other – monolithic – UNIX kernels some functionalities have to be handled in a different way. In monolithic kernels the communication between the VFS layer and the underlying file system implementation is cheap, simple function calls, while sending messages between processes is more expensive. For this reason, keeping the number of messages low during a system call is important.

It is also worth mentioning that in a monolithic kernel data structures can be easily referred at any point of the code. Between different processes this data structures have to be copied, which is again an expensive operation. On the other hand separating not related data structures into different address spaces prevents unauthorized access and therefore improves reliability and security. All in all, resources have to be distributed in an optimal way among the processes.

2.2 Components and functionalities

In this section the processes and their functionalities are introduced. The main steps of the execution of an example system call is shown in order to describe the overall mechanism.

2.2.1 Processes

The MINIX Virtual File system is built in a distributed, multiserver manner: it consists of a top-level VFS process and separate FS process for each mounted partition.

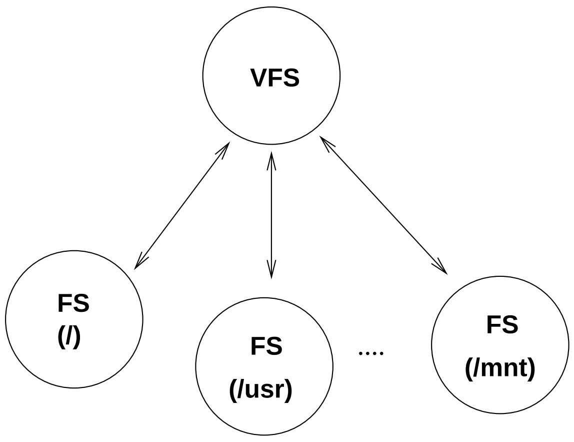


Figure 2.1: The two layers of the MINIX Virtual File system. The VFS is above the actual file system implementations according to the dependencies.

As we mentioned before, server processes are the same as the regular user processes from the kernel's point of view, although they can be layered according to the dependencies among them. Three main layer was shown by Figure 1.1. As the figure describes, the VFS and the FS processes are on the same level, however they can be divided in two sub levels.

The top-level VFS process receives the requests from user programs through system calls. If actual file system operation is involved the VFS requests the corresponding FS process to do the job. This dependency is depicted by Figure 2.2.1.

The interaction between the VFS process and the FS processes is synchronous. The VFS sends a request to the FS process and waits until the response arrives.

2.2.2 Functionalities

The top-level VFS process is responsible for the maintenance of the following data structures and the operations related to them.

- **Virtual nodes:** Virtual node is the abstract correspondence of a file on any kind of file system. It stores an identification number of the file (usually the inode number) on the underlying file system and the FS process kernel endpoint number.

- **Virtual mounts:** Virtual mounts store information about the mounted partitions. Most important attribute of this structure is the kernel endpoint number of the FS process that manages the given partition.

- **File objects:** The file object symbolizes an open file. Important fields of this structure is the corresponding virtual node and the current position in the file.

- **File descriptors:** A file descriptor is an offset value in the file descriptor table. Every slot of the file descriptor table either points to a file object or it is empty. Each process has its own file descriptor table.

- **Per-process information:** Per-process information holds information about the user who runs the process, it stores the current working and root directory. It also contains the file descriptor table.

- **File locks:** File lock table is responsible for the POSIX lock functionality. Each entry used refers to a virtual node that is locked.

- **Select entries:** Select entries are used for the implementation of the POSIX select operation.

- **Driver mapping:** Driver mapping holds entries between major device numbers and the kernel endpoint number of the given driver.

- **Character special files:** The VFS process is also in charge of handling the character special files, managing suspension and notification of the process that are interacting with character special files.

Each mounted partition is maintained by a different FS process. FS processes that handle inode based file systems usually manage the inodes and the superblock object of the partition, although a non inode – for example File Allocation Table (FAT) – based file system driver can have different structures. Each FS process has its own buffer cache.

For further details about the VFS data structures and methods please consult Chapter 3.

2.2.3 Main steps of the execution of a system call

In order to demonstrate the overall mechanism of the MINIX Virtual File system, consider the system call stat() with the argument ”/usr/src/vfs.c”. Let us assume that there is a partition mounted on the ”/usr” directory which is handled by a separate FS process.

Figure 2.2 shows the main steps. Regular lines with numbers mean messages that are exchanged during the system call, dashed lines and letters mean data that are copied.

1. The user process calls the stat() function of the POSIX library which builds the stat request message and sends it to the VFS process.

a. The VFS process copies the path name from userspace.

2. The VFS first issues a lookup for the path name. It determines that the given path is absolute, therefore the root FS process has to be requested to perform the lookup.

b. The root FS process copies the path name from the VFS’ address space.

3. During the lookup in the root FS process the root directory has to be read in order to find the string ”usr”. Let us assume that this information is not in the buffer cache. The root FS asks the Driver process to read the corresponding block from the disk.

4. The driver reads the block and transfers back to the FS process. It reports OK.

c. The driver copies the disk content into the FS’ buffer cache.

5. The root FS process examines the ”usr” directories inode data and realizes that there is a partition mounted on this directory. It sends the EENTER_MOUNT message to the VFS that also contains the number of characters that were processed during the lookup.

6. The VFS looks up in the virtual mount table which FS process is responsible for the ”/usr” partition. The lookup has to be continued in that FS process. The VFS sends the lookup request and with the rest of the path name.

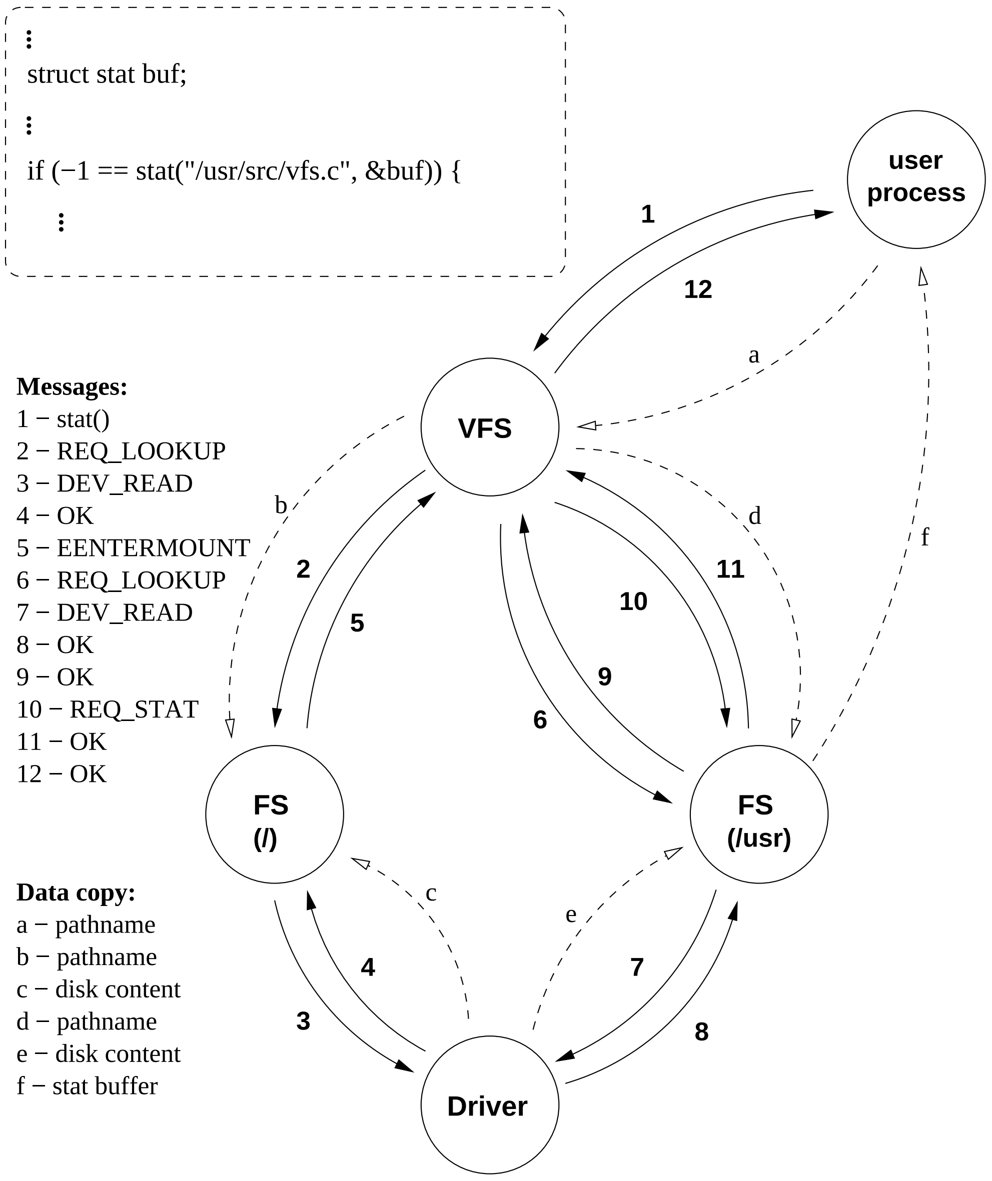


Figure 2.2: Messages changed and data copied during the stat() system call.

d. The ”/usr” FS process copies the path name from the VFS’ address space.

7. The FS process that handles the ”/usr” partition continues the lookup of the path name. It needs additional information from the disk, therefore it asks the driver process to read the given block and transfer it into the FS process' buffer cache.

8. The driver reads the disk and transfers back to the FS process. It reports success.

e. The driver copies the disk content into the ”/usr” FS process' buffer cache.

9. The FS process finishes the lookup and transfers back the inode's details to the VFS.

10. The VFS has all the necessary information in order to issue the actual REQ_STAT request. The FS process is asked to perform the stat() operation.

11. The FS process fills in the stat buffer. Let us assume that all the information needed for this operation is in the FS process' buffer cache, therefore no interaction is involved with the Driver process. The FS copies back to the user process' address space. It reports success for the VFS.

f. The FS process copies the stat buffer to the caller process' address space.

12. The VFS receives the response message from the FS process and sends the return value back to the POSIX library. The function reports success back to the user process.

2.3 Comparison

Monolithic kernels are finely tuned and optimized to be efficient. Performance is one of the key issue. In contrast, the MINIX design is about reliability and security. An immediate consequence of these is that the MINIX VFS has a different structure, it has different properties. Some of these differences are given in this section.

As we mentioned before, kernel data structures can be easily accessed in monolithic kernels and the communication between components are simple function calls. This implies that the border between the virtual layer and the actual file system implementations is not at the same place where it is in the MINIX VFS. Monolithic kernels keep as much functionality in the VFS layer as they can. Communication is free between the VFS and the underlying file system drivers therefore it makes sense to keep the virtual layer as abstract as it is possible and to reduce the functionality of the actual file system implementations. This make the implementations of a new file system easier.

Path name lookup is an other issue. In the Linux and BSD kernels, during a path name lookup the VFS calls the lookup function for every single component of the path that is not in the name cache. (See the name cache below.) This would cause a lot of messages in the MINIX case. Instead of sending messages for each component, in MINIX path name lookup is performed in the actual file system implementations. Transferring the path name and the number of characters – that were processed – is sufficient this way.

It is also worth mentioning that monolithic kernels tend to cache everything that they can. They use dynamically allocated memory for caches, while MINIX operates with static data structures. C programs with dynamically allocated memory use pointers a great deal and tend to suffer from bad pointer errors all the time, although static data structures are much simpler to manage and can never fail. MINIX VFS uses static data structures for the virtual node table, for the virtual mount table and so on.

One example of the caches in monolithic kernels is the directory name lookup cache (DNLC). The DNLC – introduced initially in 4.2BSD – provides a fast way to get from a path name to a vnode. MINIX VFS does not contain this feature, although a name cache for this purpose can be implemented in the FS processes.

Chapter 3

VFS in details

This chapter provides a detailed description of the MINIX VFS' data structures and the operations that are used for manipulating these objects (Section 3.1). It gives a general overview of the VFS/FS interface (Section 3.2) and describes the path name traversal control (Section 3.3).

3.1 VFS data-structures

This section gives a detailed description of the VFS data-structures and the related operations.

3.1.1 Vnode object

The vnode object refers to an inode. It contains the on disk inode number, the endpoint number of the File system where the file resides, the type of the file and the file size.

Type	**Field**	**Description**
int	v_fs_e	fs process endpoint
int	v_inode_nr	inode number
mode_t	v_mode	file type, protection, etc
off_t	v_file_size	file size
int	v_count	usage counter
dev_t	v_sdev	device number (in case of special file)
int	v_fifo_rd_pos	read position of a fifo
int	v_fifo_wr_pos	write position of a fifo

Table 3.1: Fields of the **vnode** structure

Vnode operations

Vnode objects are stored in a fixed-size array for simplicity and to avoid memory leaks. The following operations are used by the VFS to maintain the vnode structures:

get_free_vnode: Searches an unused vnode slot and returns it.

Return value:

*struct vnode**	Pointer to the free vnode slot.

find_vnode: Finds the vnode slot specified by the major device number and the inode number or returns a nullpointer.

Parameters:

[in] int fs_e	FS process endpoint number.
[in] int numb	Inode number.

Return value:

*struct vnode**	Pointer to the vnode slot (or NIL_VMNT).

dup_vnode: Increases the vnode's usage counter and requests the corresponding FS server to increase the inode's usage.

Parameters:

*[in] struct vnode *vp*	Pointer to the vnode object.

put_vnode: Decreases the vnode's usage counter and requests the corresponding FS server to decrease the inode's usage.

Parameters:

*[in] struct vnode *vp*	Pointer to the vnode object.

get_vnode: Requests the corresponding FS server to find/load the inode specified by the inode number parameter. The FS server sends back the details of the inode. The VFS checks whether the inode is already in use or not and increases the usage counter if it is. It returns back the vnode. If it is not already in use it acquires a free vnode object, fills in its fields and return back a pointer to it.

Parameters:

[in] int fs_e	FS process endpoint number.
[in] int numb	Inode number.

Return value:

*struct vnode**	Pointer to the vnode slot (or NIL_VMNT in case of error).

3.1.2 Vmnt object

The vmnt object represents a mounted file system. It contains the device number, the mount flags, the maximum file size on the given partition. It refers to the mounted file system's root vnode and to the vnode on which the file system is mounted on.

Type	**Field**	**Description**
dev_t	m_dev	whose mount struct is this?
int	m_flag	mount flag
int	m_fs_e	FS process' endpoint number
int	m_max_file_size	maximum file size
unsigned short	m_block_size	block size
struct vnode*	m_mountedon	vnode on which the file system is mounted on
struct vnode*	m_rootnode	mounted file system's root vnode

Table 3.2: Fields of the **vmnt** structure

The m_flag field of the structure indicates whether the mounted file system is read-only or not.

Vmnt operations

Vmnt objects are maintained in a fixed size array. There are two functions related to the vnode objects used by the VFS:

get_free_vmnt: Searches an unused vmnt slot and returns it with its index in the vmnt table.

Parameters:
*[out] short *index* Index of the free vmnt slot in the vmnt table.

Return value:
*struct vmnt** Pointer to the free vmnt slot.

find_vmnt: Finds the vmnt slot specified by the major device number or returns a nullpointer.

Parameters:
[in] int fs_e FS process endpoint number.

Return value:
*struct vmnt** Pointer to the vmnt slot (or NIL_VMNT).

3.1.3 Filp object

The filp object represents an opened file. It specifies how the file was opened, which vnode it refers to and the current file position. The rest of the fields are used by the implementation of the select system call.

Type	Field	Description
mode_t	filp_mode	RW bits, telling how file is opened
int	filp_flags	flags from open and fcntl
int	filp_count	how many file descriptors share this slot?
struct vnode*	filp_vno	referred vnode
off_t	filp_pos	file position
int	filp_selectors	select()ing processes blocking on this fd
int	filp_select_ops	interested in these SEL_* operations
int	filp_pipe_select_ops	fd-type-specific select()

Table 3.3: Fields of the **filp** structure

3.1.4 File_lock object

Structure used to manage file locking. The difference compared to the former FS implementation is the vnode pointer instead of the inode.

Type	Field	Description
short	lock_type	F_RDLOCK or F_WRLOCK
pid_t	lock_pid	pid of the process holding the lock
struct vnode *	lock_vnode	pointer to the vnode locked
off_t	lock_first	offset of the first byte locked
off_t	lock_last	offset of the last byte locked

Table 3.4: Fields of the **file_lock** structure

3.1.5 Fproc object

The fproc object maintains per-process information. Compared to the former FS implementation the differences are the working and root directories which are vnode pointers.

Type	**Field**	**Description**
mode_t	fp_umask	mask set by umask system call
struct vnode*	fp_wd	working dir's inode and fs proc reference
struct vnode*	fp_rd	root dir's inode and fs proc reference
struct filp *	fp_filp[OPEN_MAX]	the file descriptor table
uid_t	fp_realuid	real user id
uid_t	fp_effuid	effective user id
gid_t	fp_realgid	real group id
gid_t	fp_effgid	effective group id
dev_t	fp_tty	major/minor of controlling tty
int	fp_fd	place to save fd if rd/wr can not finish
char *	fp_buffer	place to save buffer if rd/wr can not finish
int	fp_nbytes	place to save bytes if rd/wr can not finish
int	fp_cum_io_partial	partial byte count if rd/wr can not finish
char	fp_suspended	set to indicate process hanging
char	fp_revived	set to indicate process being revived
int	fp_driver	which driver is proc suspended on
char	fp_sesldr	true if proc is a session leader
char	fp_execced	true if proc has exec()ced after fork
pid_t	fp_pid	process id
long	fp_cloexec	bit map for POSIX Table 6-2 FD_CLOEXEC
int	fp_endpoint	kernel endpoint number of this process

Table 3.5: Fields of the **fproc** structure

It is worth noting that there are fields that are necessary for an underlying file system implementation in order to perform its functionality (e.g. the user and group IDs during a lookup operation), but these values are logically pertain to the abstract layer. Therefore, they are transfered during the requests in case they are needed.

3.2 Interface description

The VFS communicates with the FS processes through messages. A request message contains general information about the caller process and additional arguments according to the request. Requests that refer to a file have to contain the inode number of the referred file. If data is transfered to the user process during the request, the user process' kernel endpoint number and the user space buffer address have to be transferred too.

The details of the messages exchanged and the behavior of the VFS is provided through the description of the system calls' implementation (Chapter 4). There is also a clean description given especially about the VFS/FS interface (Appendix A).

3.2.1 General VFS request message type

MINIX messages consist of a general header – which contains the sender process' endpoint number and the message type – and an additional part. The additional part is defined as an union of different structures that were needed for the communication between processes. A new structure has been defined which is used for most VFS requests. The type, name and size of the structure's fields are the following:

Type	Field	Num of bytes
long	m6_l1	4
long	m6_l2	4
long	m6_l3	4
short	m6_s1	2
short	m6_s2	2
short	m6_s3	2
char	m6_c1	1
char	m6_c2	1
char*	m6_p1	4
char*	m6_p2	4

The overall size of the structure is 28 bytes which is equal to the biggest structure that was used before. Thus the size of the union did not change.

Each VFS request and response message will be described in a table with five columns. Type, name, size and a short description is given for each field of a particular message. In addition a mapping to the a general message is also given. Messages that refer a path name will be mapped to the general VFS request message structure.

3.2.2 Get_node and put_node request messages

Get node and put node messages are used in many cases (as will be shown below). They increase and decrease the usage counter of the referred inode, respectively. Figure 3.1 shows the fields of the message.

Type	**Field**	**Bytes**	**Mapping**	**Description**
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number

Figure 3.1: Fields of the request message to increase/decrease inode usage counters. Sent by the VFS to the FS process.

3.3 Path name traverse control

Path name lookup has to be performed if the given system call has a path name argument. Lookup requests can have different intentions depending on the given system call. The intended behavior is specified by the action flag. Possible values of the action flag listed below.

The main steps that the VFS performs during the lookup call are the following: It checks whether the path name is absolute or relative and sets up the starting directory field of the message to the caller process' root or working directory inode number, respectively. During the path name traversal the process must not enter a directory which is above the process' root directory in the path tree. If the lookup is performed in the FS process which maintains the same file system on which the process' root directory lives, the root directory's inode number has to be transfered. As it is shown below, for this reason a special field is reserved in the request message which will have a nonzero value in order to indicate that the root directory is on the current file system.

Symbolic links are traversed unless it is prohibited by a specific action flag. When a symbolic link is encountered, the path name contained in the link concatenated with the rest of the path name is transferred back to the VFS and the number of characters processed is restarted. For this reason, the VFS informs the FS processes about a buffer address where the path name can be stored in case of a symbolic link. Each lookup request message contains the current value of the symbolic link loop counter in order to prevent translating symbolic links more times than a predefined limit.

Each mounted partition is maintained by a different FS process. During a path name lookup, mount points can be encountered. When a mount point is encountered, the current FS process sends back a reply which indicates that the lookup method should be continued on an other partition. The reply contains the inode number on which a partition is mounted and the offset in the path that has been already processed. The VFS looks up the vmnt table and finds the corresponding virtual mount object, thus the FS process that maintains the partition.

Note that the actual lookup takes place in the FS processes. When a new component has to be searched, the permissions of the caller process have to be checked. For this reason the VFS must transfer the caller process' user and group ID in the request. Depending on which system call is being performed these IDs can be the effective or real ones.

General description of the path name lookup method on the VFS level:

1. The VFS determines the starting directory vnode (FS process endpoint and inode number). If the path name is absolute the starting directory vnode will be the caller process' root directory vnode else the working directory vnode. Checks whether the process' root directory is on the same partition on which the lookup is to be performed and sets the root directories inode number. If the process' root directory is not on the given partition the field is set to zero, since zero is not a valid inode number.

2. Sends the request to the FS process.

3. Wait until response arrives.

    - If error, frees resources and reports failure.
    - If OK, return.
    - If a mountpoint encountered, the VFS gets the number of the inode on which a partition is mounted and looks it up in the virtual mount table. From the vmnt object the VFS can access the root vnode (FS process' endpoint and root inode number) of the partition. Checks whether the process' root directories inode is on the partition and sets the root inode number respectively. Fills in the request message with the new vnode and the rest of the path and issues the lookup request again. (Go back to 3.)
    - If the path name traversal leaves a mounted partition, the VFS checks which vnode the partition is mounted on. The FS process' endpoint and inode number can be determined from the vnode's fields. The VFS also checks whether the process' root directory resides on this partition and sets the root inode number respectively. It fills in the request message with the new vnode and the rest of the path and reissues the lookup request. (Go back to 3.)

- If the lookup encounters a symbolic link that contains an absolute path it transfers the link's path concatenated with the rest of the path name to the VFS. The increased symbolic link loop counter is also transferred back. The VFS restarts the counter for the characters that have been already processed and reissues the request with the new values. (Go back to 3.)

Note that when the path name traversal leaves a partition, the new request will contain the inode number on which the partition left is mounted on. In this case (and only in one) the FS server receives a lookup request with a starting inode on which an other partition is mounted. Since the lookup method is actually interested in the parent directory of the one on which the left partition is mounted and the last ".." component of the path is already processed in the FS process that maintains the partition that has been left, the current FS process has to lookup the parent of the starting inode first and than continue processing the path it received in the request.

Figure 3.2 shows the fields of the lookup request message:

Type	**Field**	**Bytes**	**Mapping**	**Description**
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the starting directory of the path name lookup
ino_t	req_chroot_nr	4	m6_l2	inode number of the process' root directory
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
char*	req_path	4	m6_p1	path argument
short	req_path_len	2	m6_s2	length of the path string
int	req_flag	4	m6_l3	action flag of the lookup request
char*	req_lastc	4	m6_p2	place where the last component can be stored
char	req_symloop	1	m6_c2	current value of the symbolic link loop counter

Figure 3.2: Fields of the path name lookup request message. Sent by the VFS to the FS process.

The action flag has one of the following values:

Name	Description
EAT_PATH	lookup the whole path name
EAT_PATH_OPAQUE	if the last component is a symbolic link, do not interpret it
LAST_DIR	lookup the directory that includes the last component of the path name (note: if the directory is a symbolic link, it is not interpreted)
LAST_DIR_EATSYM	if the parent directory is a symlink, interpret it

Figure 3.3: Possible values of the path name lookup's action flag.

When the VFS performs a lookup with the LAST_DIR or LAST_DIR_EATSYM intention the last component of the path name can be stored into a character array specified by the req_lastc field on the request message.

Figure 3.4 shows the fields of the response message in case of a mount point has been encountered during the lookup procedure.

Type	Field	Bytes	Mapping	Description
int	res_result	4	m_type	result value
ino_t	res_inode_nr	4	m6_l1	inode number on which a partition is mounted
short	res_offset	2	m6_s2	number of characters processed from the path
char	res_symloop	1	m6_c1	current value of the symbolic link loop counter

Figure 3.4: Fields of the response message for a lookup request in case of a mount point has been encountered. Sent by the FS process to the VFS.

The res_result value can indicate whether a mount point has been encountered and the lookup has to be continued in the FS process which maintains the partition described by vmnt object which has the vnode (identified by the inode number res_inode_nr) as its m_mounted_on field.

Figure 3.5 shows the fields of the reply message that the FS process is supposed to send for a successful lookup.

Type	Field	Bytes	Mapping	Description
int	res_result	4	m_type	result value
ino_t	res_inode_nr	4	m6_l1	inode number
mode_t	res_mode	2	m6_s1	file type, mode, etc..
off_t	res_file_size	4	m6_l2	file size
dev_t	res_dev	4	m6_l3	device number (special file)

Figure 3.5: Fields of the response message for a succesful path name lookup request. Sent by the FS process to the VFS.

3.4 Implementation steps

In this section the approach that were taken during the implementation is described. Since finding bugs in a parallel/distributed application is not easy, it seemed to be more convenient to implement an emulation of the whole procedure first and to keep everything in one process. This approach was also useful from the aspect of the verification. The original FS code was kept in the background so that the proper mechanism of the VFS layer could be verified.

The basic idea was to divide the VFS and FS related code and let them communicate only through messages. Since the final VFS and FS processes will access their – received and to be sent – messages as global variables, the interface consisted of global message variables in the emulation too.

The first function that was divided was open(). At the same time the close() also had to be modified. This made it possible to refer to the opened (regular) files as vnodes in the filp objects. Direct reference to the inode object were kept because most part of the FS code still used it. After the open(), the pipe() command was divided to its VFS and FS related parts. The next one was the internal readwrite() function to substitute the read and write operation and at the same time the clone_opcl() was also divided so that special character device files could be opened properly too. At this point the pipe related code, the data structures and code that handles the select() and the lock() got also be modified. The modification of the ftrunc(), the fstat() and the fstatfs() functions was the next step. At that point the direct reference to the file's inode object could be removed since the file system code that was related to files was already operating only on the vnode layer.

The following functions were modified at the same time: chown(), chmod(), access(), mknod(), mkdir(), inhibit read ahead for the lseek(), stat(), unlink(), rmdir(), utime(), lstat(). These functions perform path name traverse for which the starting directory of the caller process is needed. At this time the starting directories inode was directly referred from the process' fproc object, but the request and the result were transferred through messages.

In order to keep on track with the working and root directory of the running processes on the vnode layer, the init method and the mount() had to be modified. Both the chdir() and the chroot() functions use the same ”get directory” request, which had to be implemented in order to make them work. At that point the fproc objects referred through vnodes to the actual working and root directories, but the direct reference to the inode was still kept. The path name traverse was still using these direct references. The next step was to modify the path name traverse so that it started the lookup according to the inodes specified in the request messages. At that time the directly referred inodes were used to verify the process' directory vnodes at the beginning of a path name lookup. This method basically verified the mechanism of the chdir() operation too. Two special system calls, tlink() and rename() got modified next.

The next step was to implement the path name travers control in the VFS layer, which also needed some modification in parse_path() and in advance() functions in the FS code. Namely, they had to transfer back the special EENTER_MOUNT and ELEAVE_MOUNT error codes when a mount point is encountered instead of traversing it. After this step the working and root directories inode references could be removed from the fproc objects. Some modification was needed in the rename() and link() system calls. The check of the mount points changed, it is indicated by the err_code after the advance() function instead of the different device numbers between the inode of the last component's parent and the last component.

Since VFS related code was entirely operating on the vnode layer, the FS related code could be separated. The gen_io function had to be divided in order to achieve one which handles only block special devices so it could be used in the FS server. The division was necessary since the gen_io operation handles restart of suspended processes that are hanging on a character special I/O and the FS processes are not able to access per-process information.

First an experimental FS server was implemented and used only for one partition so that the proper behavior of the FS could be verified. Afterwards the FS/VFS code was completely separated into different processes. The mount system call had to be modified in order to ask the reincarnation server to execute the FS process before it sends the actual mount request to the VFS process. One MINIX FS process was included into the bootimage and into the RAM disk. See section 3.5 for more details about this.

3.5 Replacing the root partition

This section gives a detailed description about the behavior of the VFS process during the boot method.

When MINIX 3 starts up the first partition that gets mounted is the RAM disk. Since at this time there is no partition from which the FS process could be executed, one MINIX FS process is included into the bootimage. The VFS knows the kernel endpoint number of this process and it issues a request in order to mount the RAM disk. When the real on-disk root partition is about to be mounted the system executes a new MINIX FS process from the RAM disk, therefore the MINIX FS process binary is also present on the RAM disk. During the mount of the new root partition the VFS sends a request which contains a nonzero value in its req_isroot field in order to indicate that the partition is going to be the root partition of the system. The working and root directories of the processes are replaced with the root directory of the new partition. Afterwards the boot procedure can access the files on the disk.

3.6 Handling block special device files

Block special files are handled dynamically by the FS processes according to the mounted partitions. The main purpose of accessing a block special file through a FS process is the ability of caching without increasing the memory need of the VFS.

Minor devices that are not mounted are accessed through the root FS process, therefore it also maintains a mapping between major device numbers and driver endpoints. Minor devices that are mounted must be handled by the FS process that maintains the partition in order to avoid inconsistency between the buffer caches.

There are three special cases when additional care has to be taken:

- Mounting a partition which lives on a minor device that is opened.
- Opening a minor device which is mounted.
- Unmounting a partition that is opened as a block special file.

In the first case the already opened minor device is being handled by the root FS process, most probably with active blocks in the FS process' buffer cache. The handling of this device has to be moved transparently into the FS process that will handle the mounted partition. During the mount the VFS checks the vnode table whether there is an opened block special file with the same minor number as the partition that is about to be mounted. If there is, the root FS process has to be noticed in order to sync the buffer cache. In case the mount is successfully performed the requests for this minor is forwarded to the FS process that handles the partition.

During the open operation of a block special file the VFS checks whether the given minor is mounted or not. If it is, all the requests will be forwarded to the FS process that handles the partition.

Unmounting a partition that handles an open block special file requires to reassign the block special file handling to the root FS process. In this case the VFS sends the driver endpoint of the given device to the root FS process and modifies the vnode of the block special file so that all the request will be forwarded to the root FS.

3.7 Recovering from driver crash

The VFS is capable of surviving a driver crash by unmapping the driver endpoint and remapping the new endpoint of the restarted driver. The reincarnation server (RS) is responsible for restarting the new driver process. The recognition of a dead driver takes place in the low-level I/O function that interacts with the driver. Since this functionality moved to the underlying file system implementations but the driver mapping is maintained by the VFS, the recovery mechanism is more complicated in the VFS' case than it was in the former FS implementation.

When an I/O error occures in the FS server and the reported status is dead endpoint the FS process propagates back the error code to the VFS. Each VFS request is issued by a low-level function that handles the dead driver response. The VFS unmaps the endpoint for that device and waits until the RS sends the new endpoint. It maps the new endpoint number and sends it to each FS processes. Finally, it reissues the original request.

Chapter 4

System calls' implementation

The POSIX library provides wrapping functions for system calls. Such a wrapper function builds the appropriate message and sends it to the corresponding server process. This chapter provides a description about the file system related system calls' implementation. The system calls that have path name argument are described first (Section 4.1), then system calls that have a file descriptor argument are considered (Section 4.2). Finally, system calls without either path or file descriptor argument are described. (Section 4.3).

We emphasize that the aim of this Chapter is to describe the behavior of the virtual layer during the system calls. Although, the messages that are part of the VFS/FS interface are also detailed here, they are organized according to the system calls. For a clean description about the VFS/FS interface and the desired behavior of the underlying file system implementation please consult Appendix A. For further instructions about how to attach a new file system implementation to the VFS see in Appendix B.

4.1 System calls with a path name argument

In this section the system calls with a path name argument are considered since they are common in the sense that all of them performs a path name lookup before the actual request. The lookup operation translates the specified path name into the details of the file/inode that the path name refers. These values are used during the actual request. The system calls are classified into groups according to their arguments and the behavior that the VFS has to perform.

4.1.1 mkdir(), access(), chmod()

These system calls contains only a path name argument and a mode parameter: *const char *path, mode_t mode*. The chmod() and access() system calls issue a path name lookup request with the EAT_PATH flag. Figure 4.1 shows the fields of the actual request message.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the file
mode_t	req_mode	2	m6_s3	specified mode

Figure 4.1: Fields of the request message for checking or changing permissions of a file. Sent by the VFS to the FS process.

The mkdir() system call requests a lookup with the LAST_DIR flag and stores the last component of the path name. If the lookup of the parent directory of the new directory is successful the actual request message is sent. Figure 4.2 shows the fields of the message.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the parent directory
mode_t	req_mode	2	m6_s3	specified mode
char*	req_path	4	m6_p1	address of the last component string
short	req_path_len	2	m6_s2	length of the last component

Figure 4.2: Fields of the request message for creating a directory. Sent by the VFS to the FS process.

4.1.2 open(), creat()

Since the creat() system call is equivalent to open() with the flags equal to O_CREAT|O_WRONLY|O_TRUNC, creat() is considered as a special case of the open(). The open() system call's arguments are: *const char *path, int flags, mode_t mode*.

However, the open() and the creat() system calls perform a slightly different behavior on the VFS layer from the path name lookup point of view. Namely, the open() attempts to lookup the whole path name, while the creat requests only for the last directory. The actual open request differs in the fields related to the last component of the path name, the creat() includes the last component in the request, while the open() not.

The system call sees whether a free filp slot and a file descriptor is available. It also checks the vnode table for a free vnode slot. In case of successful allocation of these resources the VFS requests the lookup of the file/last directory, respectively. If the lookup was successful, the following request message is sent:

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the file (in case of open()) or the last directory (in case of creat())
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
mode_t	req_mode	2	m6_s3	specified mode
int	req_flags	4	m6_l3	flags
char*	req_path	4	m6_p1	last component of the path (in case of creat())
short	req_path_len	2	m6_s2	length of the last component (in case of creat())

Figure 4.3: Fields of the request message for creating or opening a file. Sent by the VFS to the FS process.

In case of success the FS process has to send back the details of the inode so that the vnode's fields can be filled in. If the file to be opened is a special file the device number also has to be transferred back. Figure 4.4 shows the fields of the reply message.

Type	Field	Bytes	Mapping	Description
int	res_result	4	m_type	result value
ino_t	res_inode_nr	4	m6_l1	inode number
mode_t	res_mode	2	m6_s1	file type, mode, etc..
off_t	res_file_size	4	m6_l2	file size
dev_t	res_dev	4	m6_l3	device number (special file)

Figure 4.4: Fields of the response message for an open request. Sent by the FS process to the VFS.

If the method fails, the VFS reports failure and the error value. The file descriptor table, the filp and the vnode objects were not changed so far.

If the method is successful, the VFS' further behavior depends on the file type. In case of a regular or a special file the VFS checks whether the inode specified in the response is already in use. If it is, the vnode – which already refers the given inode – has to associated with the filp slot and the vnode's usage counter has to be increased. If not the VFS fills in the fields of the free vnode and associates it with the filp slot. For special files the corresponding device's open function also has to be performed.

In case of a pipe the VFS checks if there is at least one reader/writer pair for the pipe, if not the caller has to be suspended, otherwise all other blocked processes – that hanging on the pipe – have to be revived. The VFS also checks whether there is co-reader or co-writer of the pipe. If there is, the same filp object has to be used, otherwise the free vnode's fields have to be filled with the values from the response message and it has to be associated with the free filp slot. Finally the file descriptor is returned.

Example scenario

In this section a detailed description is provided about the behavior of the VFS and FS servers and the content of the messages during the following system call:

```
fd = open("/usr/src/servers/fs/open.c", 0);
```

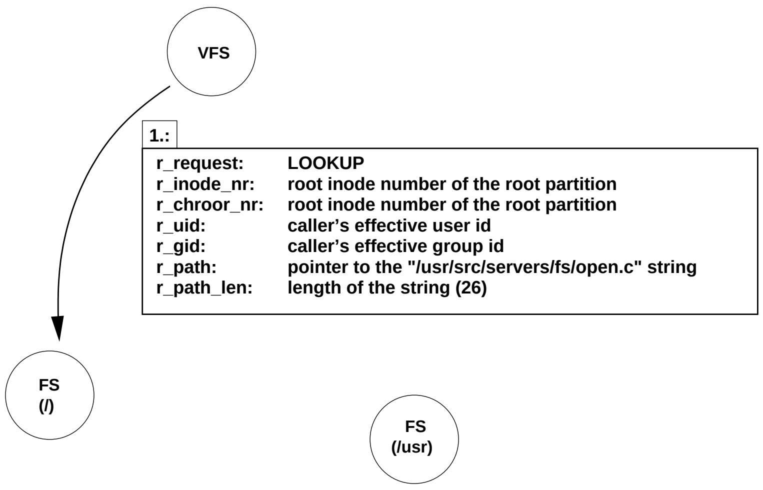
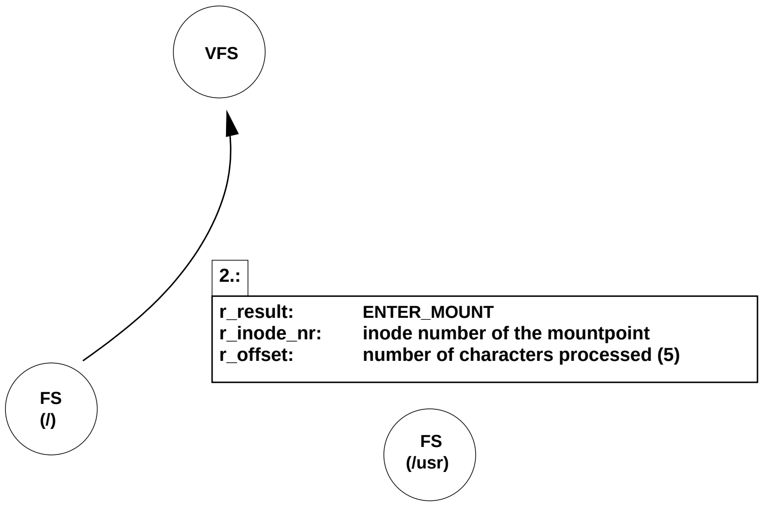

VFS
1.:
r_request: LOOKUP
r_inode_nr: root inode number of the root partition
r_chroor_nr: root inode number of the root partition
r_uid: caller's effective user id
r_gid: caller's effective group id
r_path: pointer to the "/usr/src/servers/fs/open.c" string
r_path_len: length of the string (26)
FS (/)
FS (/usr)

Figure 4.5: Lookup request to the root FS

In order to describe the exact behavior, some assumptions have to be made. Let us assume that there are two partitions, the root partition and one mounted on the ”/usr” directory, which has the index value 1 in the vmnt table. Let us also assume that the process' root directory is the root directory of the root partition and the user who is performing the system call is permitted to open the given file.

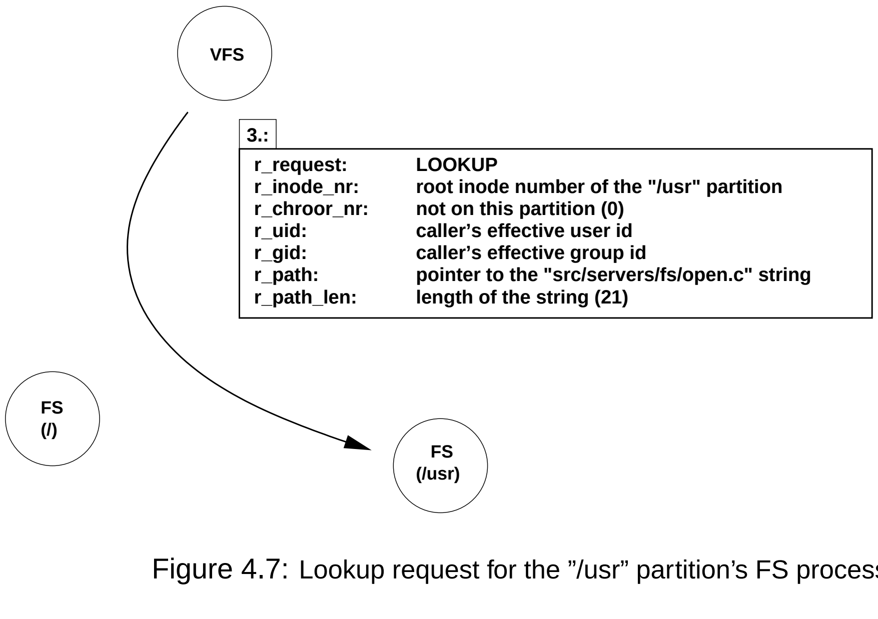
The VFS first determines that the request is an open() system call. Therefore, the path name has to be transferred from the caller process' user space. It allocates some resources, namely a free file descriptor slot, a free filp object and a free vnode have to be found. If any of these is not available the appropriate error value is reported back. Next step is to issue the lookup request message for the path name.



```
r_result:      ENTER_MOUNT
r_inode_nr:    inode number of the mountpoint
r_offset:      number of characters processed (5)
```

Figure 4.6: FS server's reply, mount point encountered

Since the path name is absolute and the process' root directory is on the root partition both the starting directories inode number and the process' root directories inode number will be the same, the root partition's root inode.



```
r_request:     LOOKUP
r_inode_nr:    root inode number of the "/usr" partition
r_chroor_nr:   not on this partition (0)
r_uid:         caller's effective user id
r_gid:         caller's effective group id
r_path:        pointer to the "src/servers/fs/open.c" string
r_path_len:    length of the string (21)
```

Figure 4.7: Lookup request for the "/usr" partition's FS process.

The path name field points the first character of the path name. The length field is the length of the whole path name. Figure 4.5 shows the message.

The FS server that maintains the root file system performs the path name lookup. It encounters a mount point after the translation of the ”usr” component.

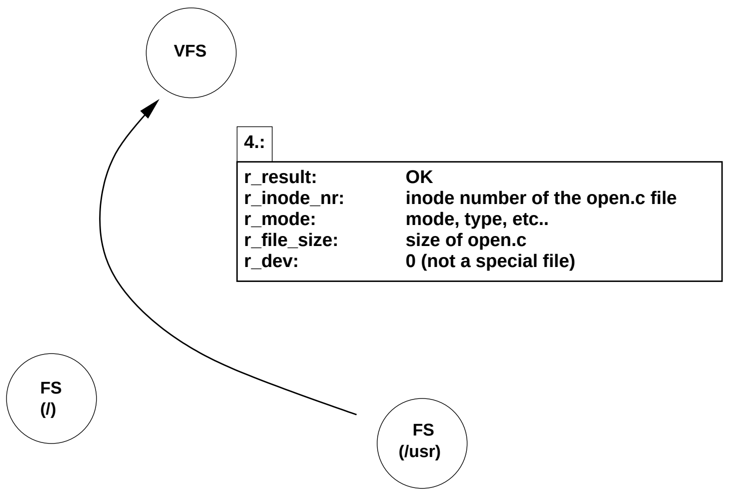

VFS
4.:
r_result: OK
r_inode_nr: inode number of the open.c file
r_mode: mode, type, etc..
r_file_size: size of open.c
r_dev: 0 (not a special file)
FS
(/)
FS
(/usr)

Figure 4.8: OK reply from the ”/usr” partition’s FS process.

This causes the FS process to send back a reply to the VFS and let it know that the path name lookup has to be continued in an other FS process.

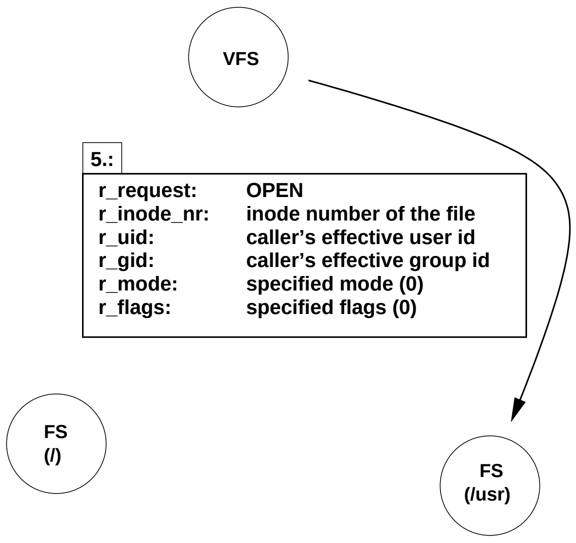

VFS
5.:
r_request: OPEN
r_inode_nr: inode number of the file
r_uid: caller’s effective user id
r_gid: caller’s effective group id
r_mode: specified mode (0)
r_flags: specified flags (0)
FS
(/)
FS
(/usr)

Figure 4.9: Open request for the ”/usr” partition’s FS process.

Figure 4.6 shows the fields of the message. The message contains the mount point’s inode number so that the VFS looks up the vmnt table and finds the one which refers the specified inode as its mount point. The VFS fills in the new the request message that has to be sent to the FS process determined by the root vnode of the referred vmnt object. The FS process that maintains the root file system already processed 6 characters from the path. The new request will contain only the remainder of the path name. Figure 4.7 shows the message. The FS process of the ”/usr” partition translates the given path name ”src/servers/fs/open.c” to the corresponding inode. It fills the reply message and sends back to the VFS process. Figure 4.8 illustrates the message.

The VFS can refer the file to be opened now. It fills in the actual open request message’s fields and sends the request to the FS process that maintains the ”/usr” partition. The mode and the flag parameters are set according to the values in the message sent to the VFS process. Figure 4.9 shows the message.

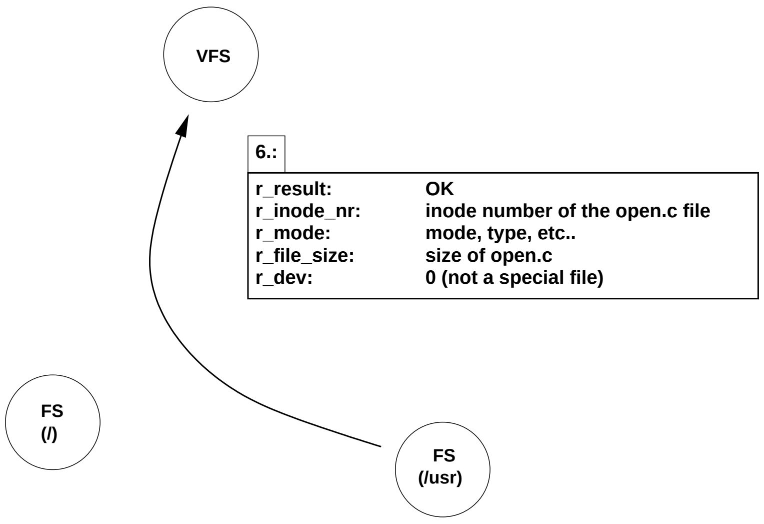

VFS
6.:
r_result: OK
r_inode_nr: inode number of the open.c file
r_mode: mode, type, etc..
r_file_size: size of open.c
r_dev: 0 (not a special file)
FS (/)
FS (/usr)

Figure 4.10: OK reply from the ”/usr” partition’s FS process.

The FS process verifies the permissions and sends back the details of the inode. Figure 4.10 illustrates the reply message.

The VFS process checks whether the inode in ther reply is already in use or not. If it is the usage counter has to be increased and the filp object will refer the used vnode. If not the VFS fills in the free vnode object’s fields according to the reply. It sets the filp object that it will refer the file’s vnode. It also sets up the file descriptor slot so that it will point to the filp object that has been filled in. Finally, the VFS returns the file descriptor number.

4.1.3 mknod()

The mknod() system call’s arguments are: *const char *path, mode_t mode, dev_t dev*. It requests a lookup for the parent directory and stores the last component of the path name. The actual request message contains the device number and the name of the last component, the fields are shown by Figre 4.11.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the parent directory
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
mode_t	req_mode	2	m6_s3	specified mode
dev_t	req_dev	4	m6_l3	device number
char*	req_path	4	m6_p1	name of the last component
short	req_path_len	2	m6_s2	length of the last component

Figure 4.11: Fields of the request message in order to create a special file. Sent by the VFS to the FS process.

Note: the mkfifo() system call is implemented with the same request, the POSIX library adds the S_IFIFO flag to the mode parameter in this case. Possible errors are handled in the FS and reported back.

4.1.4 chdir(), chroot()

The argument of the these system calls is: *const char *path*. The chdir() and chroot() system calls request a lookup for the whole path name. In case of the successful lookup the VFS checks whether the resulted inode is a directory or not. If it is a GET_DIR request is sent, with the fields shown by Figure 4.12.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the directory
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller

Figure 4.12: Fields of the request message for changing a process' working or root directory. Sent by the VFS to the FS process.

The FS process checks whether the caller is permitted to browse the directory and reports back the result. In case of success the VFS changes the working/root directory of the process, respectively.

4.1.5 unlink()

The unlink() system call has only a path name argument: *const char *path*. It request a lookup with the LAST_DIR flag and stores the last component. In case of successful lookup the actual request contains the fields shown by Figure 4.13.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the parent directory
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
char*	req_path	4	m6_p1	name of the last component
short	req_path_len	2	m6_s2	length of the last component

Figure 4.13: Fields of the request message for unlinking a file. Sent by the VFS to the FS process.

4.1.6 utime()

The utime() system calls arguments are: *const char *path, struct utimbuf *times*. The implementations of the utime() system call performs the lookup for the whole path name. In case of successful lookup the actual request message includes the access and modification timestamps. Figure 4.14 shows the fields.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the file
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
int	req_actime	4	m6_l2	access time
int	req_modtime	4	m6_l3	modification time

Figure 4.14: Fields of the request message in order to change access and modification time stamps of a file. Sent by the VFS to the FS process.

4.1.7 truncate()

The arguments of the truncate() system call are: *const char *path, off_t length*. The implementation of the truncate() system call does the following steps: first it requests a lookup for the whole path name. If the lookup is successful, the actual request message contains the new length of the file. Figure 4.15 shows the fields of the message.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the file
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
mode_t	req_mode	2	m6_s3	specified mode
int	req_length	4	m6_l3	length

Figure 4.15: Fields of the request message for changing size of a file. Sent by the VFS to the FS process.

4.1.8 chown()

The arguments of the chown() system call are: *const char *path, int owner, int group*. The implementations of the chown() system call is based on the general method that the VFS performs if a path name argument present. If the lookup successful, the request message contains the new user and group IDs, the fields are shown by Figure 4.16.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the file
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
uid_t	req_newuid	2	m6_s3	new user ID
gid_t	req_newgid	1	m6_c2	new group ID

Figure 4.16: Fields of the request message for changing owner and group ID of a file. Sent by the VFS to the FS process.

4.1.9 mount()

The mount() system call has the following arguments: *char *special, char *name, int flag*.

The VFS has to know the kernel endpoint number of the FS process which will handle the new partition. The mount command is modified for this reason. The following steps are performed by the new mount command. Figure 4.17 shows the steps.

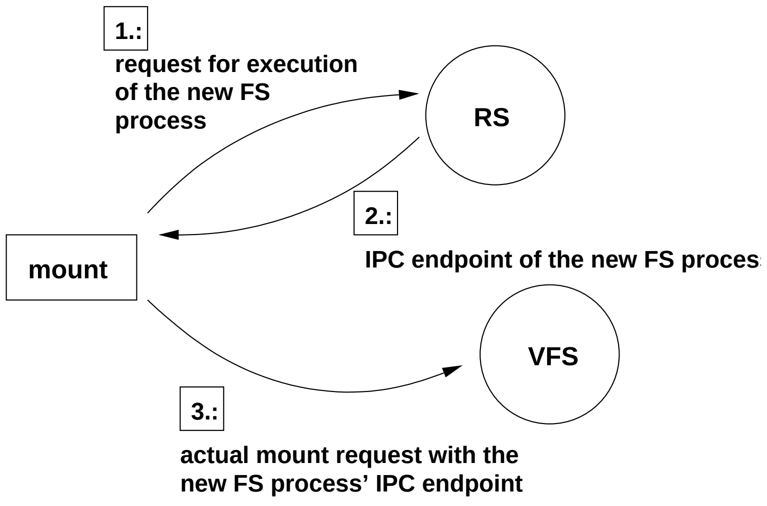

1.:
request for execution of the new FS process
RS
2.:
IPC endpoint of the new FS proces
mount
VFS
3.:
actual mount request with the new FS process' IPC endpoint

Figure 4.17: Steps performed by the mount command.

The POSIX library sends a request to the reincarnation server (RS) to start the new FS process. The reincarnation server forks a child that executes the FS process and the RS sends back its kernel endpoint number so that it can be transferred to the VFS server in the mount request.

When the VFS receives the mount request it has to check whether the caller is the superuser or not. It also stores the kernel endpoint number of the new FS process. In order to perform the mount the first step is to determine the device number from the name of the special file. For this reason the VFS performs a lookup request with path name of the special file. The corresponding FS process is supposed to send back the device number.

The VFS is now able to check the vmnt table whether this device is already mounted on or not. If it is, and the partition is not allowed to remount the VFS reports failure. (Note: replacing the root partition is a special case, it will be considered later in section 3.5.) If not, the VFS attempts to open the device and finds a free vmnt slot in the vmnt table.

At this point the VFS has a vmnt object which is either free or valid to remount. In case of a free vmnt, the VFS has to request the corresponding FS process to read the superblock and transfer back the root inode's details. Figure 4.18 shows the fields of the request message.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
int	req_boottime	4	m6_l1	timestamp of the boot time
int	req_driver_e	4	m6_l2	kernel endpoint number of the driver process
char	req_readonly	1	m6_c1	mount flag
char	req_isroot	1	m6_c2	indicates root partition
char*	req_slink_storage	4	m6_p1	buffer for storing symbolic link's content (in the VFS' address space)

Figure 4.18: Fields of the request message for reading the superblock of a partition and getting details of the root inode. Sent by the VFS to the FS process.

The request contains the driver process' endpoint number, the mount flag so that the FS process is able to register whether the partition is read-only or not and it also contains the boottime timestamp since the current time is a relative value and the FS process needs the current time in many situations. There is a special field in the message that indicates whether the partition is the root partition of the system or not. (This value is needed when the partition is to be left during a path name lookup. The FS process has to know if it is the root partition or it should report an ELEAVEMOUNT error code.) The message also contains a buffer address in the VFS' address space where a path name can be stored in case of a symbolic link has been encountered.

The FS attempts to read the superblock. If the partition does not contain a valid superblock the error value is reported back.

The next step is to send a request message to lookup the mount point and get the corresponding inode. If the lookup is successful a special mountpoint request is sent. Figure 4.1 shows the fields of this message.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the starting directory of the path name lookup

Table 4.1: Fields of the request message in order to register a mount point on a partition. Sent by the VFS to the FS process.

The answer message contains the details of the inode, the message has the same layout as the reply for the lookup. Figure 3.5 shows the fields. Possible errors related to the mount point is handled by the FS process. The mount point's inode is now determined.

Finally the VFS fills in the fields of the vmnt structure and reports success.

4.1.10 unmount()

First step of the unmount() system call is to determine the device number from the name of the special file. For this reason the VFS performs a request with path name of the special file. The corresponding FS process is supposed to send back the device number.

The VFS can check the vmnt table whether the device is mounted. If it is the VFS sends the unmount request for the FS process that manages the partition. If the given partition is not in use the unmount can be performed. The VFS drops the vnode on which the partition was mounted and clears the fields of the vmnt object.

4.1.11 rename()

The rename() system call has two path name arguments. The call can be performed only if the two names refer to the same partition. Since different partitions are maintained by different FS processes the VFS has to perform this system call using multiple messages. The first step is to determine the inode number of the parent directory of the first path name and to save the last component. In case of the successful lookup the request message contains the parent directories inode number, otherwise the appropriate error value is reported back.

The VFS performs a full path name lookup for the new name in order to determine whether it is a directory or not. During the rename this directory is to be removed, but removing a directory is not allowed if it is a working or root directory of any processes, therefore the VFS verifies this condition. If the directory can be removed or the last component is not a directory the VFS requests a parent lookup for the new name and stores the last component.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_old_dir	4	m6_l2	inode number of the old name parent dir
ino_t	req_new_dir	4	m6_l3	inode number of the new name parent dir
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
char*	req_path	4	m6_p1	last component of the old name
short	req_path_len	2	m6_s2	length of the component
char*	req_user_addr	4	m6_p2	last component of the new name
short	req_slen	2	m6_s3	length of the component

Figure 4.19: Fields of the request message for renaming a file. Sent by the VFS to the FS process.

If the last directory of the new name and the old name are on the same partition the VFS can request the actual rename. This message contains the inode number of the last directory of the existing name and the last component of the existing path. It also contains the inode number of the last directory of the new name and the last component of the new name path (i.e. the new name). The fields of the message are shown by Figure 4.19.

4.1.12 link()

The link() system call has two path name argument. Linking is only possible if the two names are on the same partition. Since different partitions are maintained by different FS processes the VFS has to perform this system call using multiple messages.

First the file to be linked has to be found. A lookup request is sent for the whole path name for this reason. Second step is to find the last directory of the link name. This is performed by a parent lookup request and the last component of the path stored. The VFS checks whether the file to be linked and the parent directory of the link name is on the same partition. If not, the appropriate error value is reported back. The final request is the actual linking. It contains the inode number of the file to be linked, the inode number of the parent directory of the link and the last component of the link name. The FS process performs the actual rename and reports success or the appropriate error value. Figure 4.20 shows the fields of the message.

Type	**Field**	**Bytes**	**Mapping**	**Description**
int	req_request	4	m_type	request code
ino_t	req_linked_file	4	m6_l1	inode number of the file to be linked
ino_t	req_link_parent	4	m6_l2	inode number of link's parent dir
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
char*	req_path	4	m6_p1	last component of the link name
short	req_path_len	2	m6_s2	length of the component

Figure 4.20: Fields of the request message to create a hard link. Sent by the VFS to the FS process.

4.1.13 slink()

The slink() system call has two path name argument. The target file's path name (i.e. the content of the symbolic link) and the link's name itself.

The implementation first issues a parent lookup for the link's path name and saves the last component of the path in a character array.

The fields of the actual request message is shown by Figure 4.21.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the file
uid_t	req_uid	2	m6_s1	effective user ID of the caller
gid_t	req_gid	1	m6_c1	effective group ID of the caller
char*	req_path	4	m6_p1	last component of the link name
short	req_path_len	2	m6_s2	length of the component
int	req_who_e	4	m6_l3	kernel endpoint number of the caller
char*	req_user_addr	4	m6_p2	user space buffer address (link content)
short	req_slen	2	m6_s3	length of the referred path

Figure 4.21: Fields of the request message for creating a symbolic link. Sent by the VFS to the FS process.

4.1.14 rdlink()

The rdlink() system call has two arguments: a path name – the link's path – and the size of the buffer in the user's address space. The implementation requests a lookup for the path name without interpreting the link (i.e. with the EATH_PATH_OPAQUE flag). The actual request contains the inode number of the symlink file, the kernel endpoint of the caller, the user space buffer address and the size of the buffer. Figure 4.22 shows the fields of the message.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the file
uid_t	req_uid	2	m6_s1	effective user ID of the caller
gid_t	req_gid	1	m6_c1	effective group ID of the caller
int	req_who_e	4	m6_l3	kernel endpoint number of the caller
char*	req_user_addr	4	m6_p2	user space buffer address
short	req_slen	2	m6_s3	size of the buffer

Figure 4.22: Fields of the request message for reading a symbolic link's content. Sent by the VFS to the FS process.

4.1.15 stat()

The arguments of the stat() system call are: *const char *path*, *struct stat *buf*, The implementation of the system call is based on the general method that the

VFS performs if a path name argument present. In case of the successful lookup the actual request message contains the kernel endpoint of the caller process and the user space buffer's address, the fields are shown by Figure 4.23.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_inode_nr	4	m6_l1	inode number of the file
uid_t	req_uid	2	m6_s1	effective or real user ID of the caller
gid_t	req_gid	1	m6_c1	effective or real group ID of the caller
int	req_who_e	4	m6_l3	kernel endpoint number of the caller
char*	req_user_addr	4	m6_p2	user space buffer address

Figure 4.23: Fields of the request message for getting statistics of a file. Sent by the VFS to the FS process.

4.2 System calls with file descriptor argument

In this section the system calls which operate on a file descriptor are considered. Each system call determines first which filp object and therefore which vnode the file descriptor refers to. If the file descriptor is not valid it reports failure. Otherwise the corresponding vnode specifies the FS process and the inode number on which the particular system call operates.

In general, the reply message that the FS process is supposed to send contains the result value, stored in the m_type field of the response message. This value is reported back to the caller process. In case the FS process is supposed to send additional information, the message is explicitly described.

4.2.1 lseek()

The lseek() system call has the following arguments: *int fd, off_t offset, int whence* and it is mainly performed on the VFS level. It updates the file position according to its argument. In case of the new position is different than the old one the VFS request the FS process to inhibit the read ahead on the inode that the vnode refers to. The request message contains the message code and the inode number. The fields of the message are shown by Figure 4.24.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_fd_inode_nr	4	m2_i1	inode number of the file

Figure 4.24: Fields of the request message for inhibiting read ahead feature on a given file. Sent by the VFS to the FS process.

4.2.2 read(), write()

The read() and write() system calls have the following arguments: *int fd, void *buff, size_t nbytes*. During the execution of each of these system calls the VFS finds the corresponding filp object and the vnode specified by the file descriptor. The VFS checks whether the user process has the memory it needs. It checks whether the file is a special file. If it is, the corresponding IO call is performed. If the vnode refers to a pipe and the operation is write() the VFS has to check how many bytes are allowed to write (partial count). The minimum of the requested bytes and the partial count has to be transfered. Figure 4.25 shows the fields of the request message.

The FS process performs the request and reports back the result. In case of any error the corresponding error value is sent back.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_fd_inode_nr	4	m2_i1	inode number of the file
int	req_fd_who_e	4	m2_i2	kernel endpoint number of the caller
int	req_fd_seg	4	m2_l2	segment
off_t	req_fd_pos	4	m2_i3	current position
size_t	req_fd_nbytes	4	m2_l1	number of bytes to be transfered
char*	req_fd_user_addr	4	m2_p1	user space buffer address

Figure 4.25: Fields of the request message for reading from or writing to a file. Sent by the VFS to the FS process.

If the operation succeeds the number of bytes transferred is sent back. In case of a write() the new file size also has to be transferred back. The fields of the reply message are shown by Figure 4.26.

Type	Field	Bytes	Mapping	Description
int	res_result	4	m_type	result value (number of bytes transferred or error value)
off_t	res_file_size	4	m2_i1	new file size (in case of write())
int	res_cum_io	4	m2_i2	number of bytes transfered

Figure 4.26: Fields of the response message for a read/write request. Sent by the FS process to the VFS.

Example scenario

In this section a detailed description will be given about the behavior of the VFS and FS servers and the content of the messages during the following system call:

```
read(fd, buf, 1024);
```

The VFS server first finds the filp object and the vnode which are specified by the file descriptor. It checks whether the user process has the memory that it needs in order to perform the required transfer.

Let us assume that the file descriptor in the example specifies a regular file on the root partition. The VFS process will send a message to the FS process – specified by the vnode. Figure 4.27 shows the request and the response messages.

The request message is filled in by the values of the filp object and the referred vnode.

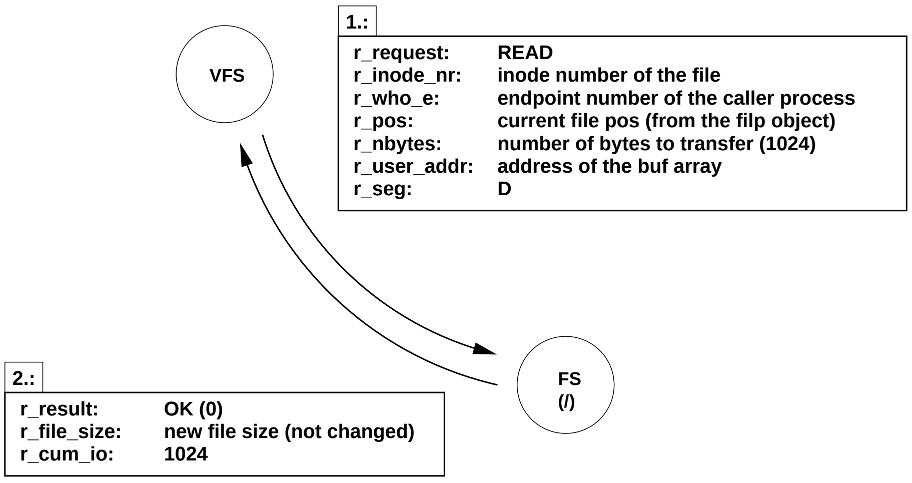

1.:
r_request: READ
r_inode_nr: inode number of the file
r_who_e: endpoint number of the caller process
r_pos: current file pos (from the filp object)
r_nbytes: number of bytes to transfer (1024)
r_user_addr: address of the buf array
r_seg: D
VFS
FS
(/)
2.:
r_result: OK (0)
r_file_size: new file size (not changed)
r_cum_io: 1024

Figure 4.27: Request and response for reading 1024 bytes.

Let us also assume that the request can be performed without any error by the FS process. In this case it sends back a message with the number of bytes transferred.

4.2.3 close()

The close() system call checks whether the corresponding vnode represents a special file. If it does the VFS closes the device. It clears the file descriptor. If the filp object's usage counter becomes zero it drops the referred inode with the put_node message. It releases the lock if the file was locked and returns.

4.2.4 fchdir()

The fchdir() system call drops the current working dir's vnode with a put_node message and replaces is with the vnode that is referred by the file descriptor given in the argument. It increases the usage counter of the corresponding inode with a get_node message.

4.2.5 pipe()

The pipe() system call first acquires two file descriptors. Afterwards it sends a request for the root file system so that it allocates an inode for the pipe. The message contains only the request code and causes the FS process to allocate an inode with the I_NAMED_PIPE mode flag. The FS process increases the inode's counter since it is needed twice. The reply contains the inode number of the allocated inode.

4.2.6 fstat(), fstatfs()

The fstat() and the fstatfs() have the arguments *int fd, struct stat *buf* and *int fd, struct statfs *buf* respectively. Each of them sends the same request message. Figure 4.28 shows the fields.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_fd_inode_nr	4	m2_i1	inode number of the file
int	req_fd_who_e	4	m2_i2	kernel endpoint number of the caller
char*	req_fd_user_addr	4	m2_p1	user space buffer address

Figure 4.28: Fields of the request message for getting statistics of an opened file or getting statistics of a partition. Sent by the VFS to the FS process.

4.2.7 ftruncate()

The ftruncate() system call has the following arguments: *int fd, off_t length*. After it determines the corresponding vnode it sends the actual request message is sent. The fields of the message are shown by Figure 4.29.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
ino_t	req_fd_inode_nr	4	m2_i1	inode number of the file
off_t	req_fd_length	4	m2_i2	new length

Figure 4.29: Fields of the request message for changing size of an opened file. Sent by the VFS to the FS process.

4.2.8 dup()

The dup() system call operates on the VFS level, it does the same as the former FS implementation.

4.2.9 fcntl()

The fcntl() system call has the following arguments: *int fd*, *int cmd* and an optional data arguments which can be a user space buffer address. The fcntl() operation is performed mainly on the VFS level, except when a section of a file is to be freed. In this case the actual request message is sent. Figure 4.30 shows the fields.

Type	**Field**	**Bytes**	**Mapping**	**Description**
int	req_request	4	m_type	request code
ino_t	req_fd_inode_nr	4	m2_i1	inode number of the file
off_t	req_fd_start	4	m2_i2	start position
off_t	req_fd_end	4	m2_i3	end position

Figure 4.30: Fields of the request message for freeing a section of a file. Sent by the VFS to the FS process.

4.3 System calls without arguments

4.3.1 fork(), exit()

The fork() and the exit() system calls work the same way how they worked in the former FS server, except that they operate on vnodes (i.e. they use the put node and get node routines).

4.3.2 sync(), fsync()

Although the fsync() system call has a file descriptor argument in MINIX it behaves the same as the sync() call. Both of them iterate through the vmnt objects and send a sync request to each mounted partition.

4.3.3 clone_opcl()

Some devices which can be accessed through character special files need special processing upon open. Such a device is replaced with a new unique minor device number after open. This causes the VFS to request the root FS server to create a new character special inode with the new device number. The fields of the request message is shown by Figure 4.31.

Type	Field	Bytes	Mapping	Description
int	req_request	4	m_type	request code
dev_t	req_dev	4	m6_l3	device number

Figure 4.31: Fields of the request message for cloning a character special file. Sent by the VFS to the FS process.

The FS server allocates the new inode, sets the device number and returns back the inode's details. The VFS than replaces the vnode referred by the filp object with a new one that refers to the inode just created.

Chapter 5

Performance measurements

This chapter provides a brief comparison between the original FS server and the VFS system from the performance point of view. The duration of POSIX system calls have been measured with and without VFS on the 3.1.3 version of MINIX 3. The hardware was a 400MHz PII with 256 megabytes of RAM and a 30G IDE hard drive.

System call	**original FS**	**VFS**	**Overhead**
Lseek	6.5μs	6.7μs	3%
Open+Close	24.6μs	65.6μs	166%
Read 1K	29.0μs	42.8μs	47%
Read 8K	45.8μs	61.2μs	33%
Read 64K	382.3μs	435.2μs	13%
Read 128M	124s	131s	5%
Creat+Write+Del	120.5μs	216.3μs	80%
Write 128M	189s	193s	2%
Fork	1456.8μs	1508.4μs	3%
Fork+Exec	2790.2μs	2910.1μs	4%
Mkdir+Rmdir	109.6μs	182.3μs	66%
Rename	39.8μs	117.2μs	200%
Chdir	15.4μs	41.2μs	173%
Chmod	24.4μs	32.6μs	33%
Stat	15.1μs	22.4μs	48%

Table 5.1: Duration of the system calls with and without VFS.

A user program was programmed to record the real time in units of clock ticks, then make a system call millions of times, then record the real time again. The system call time was computed as the difference between the end and start time divided by the number of calls, minus the loop overhead, which was measured separately. The number of loop iterations was different for each test because testing getpid 100 million times was reasonable but reading a 128-MB file 100 million times would have taken too long. All tests were made on an idle system.

Let us briefly examine the results. Lseek() performs basically with the same efficiency, which is reasonable, it is almost doing the same. Open() and close() have to manage resources on the VFS layer and due to the split it also changes messages, the overhead is significant. The read() system calls show that the more data is being transferred the less overhead it means. The creat()+write()+del(), the mkdir()+rmdir() and the rename() cause a siginificant overhead. In this cases resources on the virtual layer have to be managed and extra data (path name) has to be copied during the system calls. Chdir() changes two messages and manages the vnode table. Chmod() and Stat() do the same in the two cases, except the split and therefore the messages involved in the operation. As the table shows in both cases the difference is around 7-8μs.

Let us take a closer look at the system calls, which cause significant overhead. Rename() gives the highest, although it is basically not important since it is a rare operation. The main reason is the number of messages exchanged and the extra data copy involved. During this system call two lookup requests are issued – each of them involves data copy – and the actual rename request is sent, which again involves the copying of the last components of the path names. Chdir() issues two requests. The first is a lookup for the directory name, the second is the special get directory request. It also has to scan the vnode table in order to find out whether the specified node is already in use or not. Although, chdir() is not a frequent system call either. The most important is the open()+close() case. Open() has basically the same overhead with the chdir(), the two operations almost do the same – in case of opening a regular file – except that open() does more verification. Although, open() is a frequent and therefore important system call.

Decreasing the number of messages exchanged during these system calls could be managed with the introduction of an intention flag in the lookup request. It could handle special cases – like open(), chdir() and rename() – and perform the actual request in case the inode has been found. Thus, the extra message for the actual request would be unnecessary. A more sophisticated method could be implemented for the rename() system call. Since it can be successful only if the two inodes are on the same partition, during the lookup the last components of the path names could be stored in the FS server. Thus, the actual operation would not need any extra data copy for the pathnames. Moreover, the second lookup could contain a rename intention and the actual operation could be immediately performed.

Chapter 6

Related work

This chapter surveys related work on Virtual File systems. It describes the main evolutionary path that UNIX took from the early research editions through to System V Release 4, which involved the last major enhancements to the UNIX file system architecture. It gives an overview of the development of the File System Switch (FSS) architecture in SVR3, the Sun VFS/vnode architecture in SunOS, and then the merge between the two to produce SVR4. Many different UNIX and UNIX-like vendors adopted the Sun VFS/vnode interface, however their implementations differed in many areas. A brief description about the BSD and Linux operating systems from a file system perspective is also given. The chapter is based on the book ”UNIX File systems: Evolution, Design and Implementation” [Pat03].

The need for change

The research editions of UNIX had a single file system type. Before long, the need to add new file system types, including non-UNIX file systems, resulted in a shift away from the old style file system implementation to a newer architecture that clearly separated the different physical file system implementations from those parts of the kernel that dealt with file and file system access.

6.1 The System V File System Switch

Introduced with System V Release 3.0, the File System Switch (FSS) architecture provided a framework under which multiple different file system types could coexist in parallel. As with earlier UNIX versions, SVR3 kept the mapping between file descriptors in the user area to the file table to in-core inodes.

The boundary between the file system-independent layer of the kernel and the file system-dependent layer occurred mainly through a new implementation of the in-core inode. Each file system type could potentially have a very different on-disk representation of a file. Newer diskless file systems such as Network File system (NFS) had different, non-disk-based structures. Thus, the new inode contained fields that were generic to all file system types such as user and group IDs and file size, as well as the ability to reference data that was file system-specific.

The set of file system-specific operations was defined in a structure. An array held an entry with this structure for each possible file system. When a file was opened for access, the corresponding field of the inode was set to point to the entry – in the array, which holds the structures for the file system-specific operations – for that file system type. In order to invoke a file system-specific function, the kernel performed a level of indirection through a macro that accessed the appropriate function.

All file systems followed the same calling conventions so they clould all understand how arguments would be passed.

6.2 The Sun VFS/Vnode Architecture

Developed on Sun Microsystem's SunOS operating system, the world first came to know about vnodes through Steve Kleiman's often-quoted Usenix paper "Vnodes: An Architecture for Multiple File System Types in Sun UNIX" [Kle86]. The paper stated four design goals for the new file system architecture:

- The file system implementation should be clearly split into a file system independent and file system-dependent layer. The interface between the two should be well defined.

- It should support local disk file systems such as the 4.2BSD Fast File System (FSS), non-UNIX like file systems such as MS-DOS, stateless file systems such as NFS, and stateful file systems such as RFS.

- It should be able to support the server side of remote file systems such as NFS and RFS.

- File system operations across the interface should be atomic such that several operations do not need to be encompassed by locks.

Because the architecture encompassed non-UNIX- and non disk-based file systems, the in-core inode that had been the memory-based representation of a file over the previous 15 years was no longer adequate. A new type, the vnode was introduced. This simple structure contained all that was needed by the file system-independent layer while allowing individual file systems to hold a reference to a private data structure; in the case of the disk-based file systems this may be an inode, for NFS, an rnode, and so on.

There is nothing in the vnode that is UNIX specific or even pertains to a local file system. Of course not all file systems support all UNIX file types. For example, the DOS file system does not support symbolic links. However, file systems in the VFS/vnode architecture are not required to support all vnode operations. For those operations not supported, the appropriate field was set to a special function used only for this reason. All operations that can be applied to a file were held in the vnode operations vector. The set of vnode operations were accessed through macros.

To provide more coherent access to files through the vnode interface, the implementation provided a number of functions that other parts of the kernel could invoke. This layer was called the ”Veneer layer”. The Veneer layer avoids duplication throughout the rest of the kernel by providing a simple, well-defined interface that kernel subsystems can use to access file systems.

The Sun VFS/vnode interface was a huge success. Its merger with the File System Switch and the SunOS virtual memory subsystem provided the basis for the SVR4 VFS/vnode architecture. There were a large number of other UNIX vendors who implemented the Sun VFS/vnode architecture. With the exception of the read and write paths, the different implementations were remarkably similar to the original Sun VFS/vnode implementation.

6.3 The SVR4 VFS/Vnode Architecture

System V Release 4 was the result of a merge between SVR3 and Sun Microsystems' SunOS. One of the goals of both Sun and AT&T was to merge the Sun VFS/vnode interface with AT&T's File System Switch.

The new VFS architecture, which has remained largely unchanged for over 15 years, introduced and brought together a number of new ideas, and provided a clean separation between different subsystems in the kernel. One of the fundamental changes was eliminating the tight coupling between the file system and the VM subsystem which was particularly complicated resulting in a great deal of difficulty when implementing new file system types.

With the introduction of SVR4, file descriptors were allocated dynamically up to a fixed but tunable limit. The Virtual File System Switch Table, which contains an entry for each file system that can reside in the kernel was built dynamically during kernel compilation. The vnode structure had only some subtle differences. The vnode operations were still accessed through the use of macros.

6.3.1 The Directory Name Lookup Cache

Introduced initially in 4.2BSD and then in SVR4, the directory name lookup cache (DNLC) provided an easy and fast way to get from a path name to a vnode. For example, in the old inode cache method, parsing the path name /usr/src/server-s/fs would involve working on each component of the path name one at a time. The inode cache merely saved going to disk during processing of iget(), not to say that this is not a significant performance enhancement. However it still involved a directory scan to locate the appropriate inode number. With the DNLC, a search may be made by the name component alone. If the entry is cached, the vnode is returned.

The structures held by the DNLC are hashed to improve lookups. This alleviates the need for unnecessary string comparisons. To access an entry in the DNLC, a hash value is calculated from the filename and parent vnode pointer. The appropriate entry in the hash array is accessed, through which the cache can be searched.

6.4 BSD File System Architecture

The first version of BSD UNIX, introduced in 1978, was based on 6th Edition UNIX. Almost from day one, subtle differences between the two code bases started to appear. However, with 3BSD, introduced in 1980 and based on 7th Edition, one can still see very similar code paths between 3BSD and 7th Edition UNIX.

The three the most significant contributions that the Berkeley team made in the area of file systems were quotas, the directory name lookup cache (DNLC), and the introduction of the Berkeley Fast File System (FFS), which would eventually be renamed UFS (UNIX File System).

Around the time of 4.3BSD, traces of the old UNIX file system had disappeared. The file system disk layout was that of early UFS, which was considerably more complex than its predecessor. The in-core file structure still pointed to an in-core inode but this was changed to include a copy of the disk-based portion of the UFS inode when the file was opened. The implementation of namei() also became more complex with the introduction of the name cache (DNLC).

The vnode layer which was introduced by Sun Microsystems is also present in the BSD kernel since the 4.4BSD version. However, there are many differences in the implementation. For further information on the BSD Virtual File system layer please consult the book ”The Design and Implementation of the 4.4BSD Operating System” [MBKQ96].

6.5 Linux Virtual File System Switch

The Linux community named their file system architecture the Virtual File System Switch, or Linux VFS which is a misnomer because it was substantially different from the Sun VFS/vnode architecture and the SVR4 VFS architecture that preceded it. However, as with all POSIX-compliant, UNIX-like operating systems, there are many similarities between Linux and other UNIX variants.

This section gives some notes about the changes in the Linux 2.4 and a brief description about the VFS in Linux 2.6 kernel series. For further details on the earlier Linux kernels see [BHBK+96], for further details on the 2.6 series consult the book ”Understanding the Linux Kernel” [BC05].

6.5.1 Linux from the 2.4 Kernel Series

The Linux 2.4 series of kernels substantially changes the way that file systems are implemented. Some of the more visible changes are:

- File data goes through the Linux page cache rather than through the buffer cache. There is still a tight relationship between the buffer cache and page cache, however.
- The dcache is tightly integrated with the other file system-independent structures such that every open file has an entry in the dcache and each dentry (which replaces the old dir_cache_entry structure) is referenced from the file structure.
- There has been substantial rework of the various operations vectors and the introduction of a number of functions more akin to the SVR4 page cache style vnode ps.
- A large rework of the SMP-based locking scheme results in finer grain kernel locks and therefore better SMP performance.

The migration towards the page cache for file I/O actually started prior to the 2.4 kernel series, with file data being read through the page cache while still retaining a close relationship with the buffer cache.

6.5.2 Linux 2.6 VFS

The key idea behind the Linux VFS consists of introducing a *common file model* capable of representing all supported file systems. This model strictly mirrors the file model provided by the traditional Unix file system. This is not surprising, because Linux wants to run its native file system with minimum overhead.

However, each specific file system implementation must translate its physical organization into the VFS' common file model.

One can think of the common file model as object-oriented, where an object is a software construct that defines both a data structure and the methods that operate on it.

The common file model's most important object types:

Superblock : Stores information concerning a mounted file system. For disk-based file systems, this object usually corresponds to a file system control block stored on disk.

Inode : Stores general information about a specific file. For disk-based file systems, this object usually corresponds to a file control block stored on disk. Each inode object is associated with an inode number, which uniquely identifies the file within the file system.

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

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

6.5.3 The Dentry Cache

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

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

6.6 QNX Neutrino RTOS

QNX is a commercial, real-time operating system (RTOS) with a microkernel architecture [Ltd06]. Because of its real-time properties it is widely used in embedded devices. QNX was originally created at the University of Waterloo, but has been commercialized and produced by QNX Software Systems since 1981.

QNX Neutrino supports a variety of file systems. Like most service-providing processes in the QNX OS, these file systems execute outside the kernel and applications use them by communicating via messages generated by the shared-library implementation of the POSIX API. Most of these file systems are so-called resource managers. Each file system adopts a portion of the path name space (i.e. a mountpoint) and provides file system services through the standard POSIX API. File system resource managers take over a mountpoint and manage the directory structure below it. They also check the individual path name components for permissions and for access authorizations.

According to the website this implementation means that:

- File systems may be started and stopped dynamically.
- Multiple file systems may run concurrently.
- Applications are presented with a single unified path name space and interface, regardless of the configuration and number of underlying file system.
- A file system running on one node is transparently accessible from any other node.

When a file system resource manager registers a mountpoint, the process manager creates an entry in the internal mount table for that mountpoint and its corresponding server ID.

This table joins multiple file system directories into what users perceive as a single directory. The process manager handles the mountpoint portion of the path name, the individual file system resource managers take care of the remaining parts of the path name.

The file systems are implemented as shared libraries (essentially passive blocks of code resident in memory), which can be be dynamically loaded to provide file system interfaces and services.

Chapter 7

Summary and conclusion

This chapter gives a summary of this master's thesis. Section 7.1 starts with an overview of the major contributions of this project by summarizing the results presented in the previous chapters. Section 7.2 gives a description of open issues and directions for future research.

7.1 Contributions

The main contribution of this work is that the original FS server was fully revised in order to split it into a virtual layer and the actual MINIX file system implementation. The MINIX Virtual File system is built in a distributed, multi-server manner, which is a substantially different architecture compared to other UNIX-like solutions.

7.1.1 Virtual File system layer

An abstract layer has been designed and implemented, which is in charge of controlling the overall mechanism of the VFS by issuing accurate requests for the appropriate FS servers during the system calls. It is also in charge of maintaining abstract data structures that are independent from the underlying file system drivers and are playing important roles within the Virtual File system's functionality. In order to achieve this several data structures and functions operating on them had to be designed and added to the former FS server. The communication with underlying file system drivers also had to be implemented.

7.1.2 MINIX file system driver

The MINIX file system implementation had to be separated from the original FS code and some part of it had to be rewritten so that it would be capable of cooperating with the virtual layer. Modifications relating to the inode handling and the path name lookup had to be realized. The communication according to the VFS/FS interface also had to be implemented.

7.1.3 Other contributions

As part of this work the mount system call got modified so that it issues a request for the reincarnation server to execute a new FS binary. The functionality of reporting back a newly executed server's kernel endpoint number had to be added to the reincarnation server. One instance of the MINIX FS server was included into the boot image and into the RAM disk.

As a reference document the VFS/FS interface is detailed in this thesis. A developer's guide is also provided by an example binary tree file system in order to show how to realize and attach a new file system implementation to the MINIX VFS.

7.2 Future work

7.2.1 Buffer cache as shared library

FS processes have their own buffer cache. Very often a process that handles a special partition is idle. Since the buffer cache is allocated by static memory even in an idle state the FS process holds the memory and prevents it to be used by other processes. This phenomenon also means that the distribution of the overall buffer cache memory is not proportional by the needs of the single partitions (again, one FS' buffer cache can not be used by an other FS). For this reason a shared library buffer cache could be implemented and linked to each FS binary so that the buffer cache could adapt for the needs.

7.2.2 Asynchronous VFS/FS interface

Currently the VFS/FS interface supports only synchronous interaction. It has its advantage, namely it can be easily implemented and it is also easy to use. Although, many cases would require the opportunity of asynchronous interaction. For instance, a network based file system. An asynchronous interface could be implemented with a similar – notification based – technique, how the VFS communicates with character special files.

7.2.3 Framework for BSD file system implementations

The VFS layer in monolothic UNIX kernels – like the BSD – performs functionalities that are entirely moved to the FS processes in the MINIX VFS case. For instance, the path name lookup in monolothic kernels is performed on the VFS layer and low-level lookup functions provide access for finding a component in a directory. This functionality is common in each FS process in the MINIX VFS system. Therefore a general framework could be written that handles the lookup and calls the low-level file system specific functions. This layer could behave as an interface between the MINIX VFS and the actual file system implementations in a monolothic – for example the BSD – kernel. The low-level I/O functions that are called by the file system implementation code could also be provided by the framework and could be translated to MINIX I/O messages in order to interact with the driver processes.

Bibliography

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Publishing, 3rd edition, 2005. ISBN: 0-596-00565-2.

[BHBK$^+$96] M. Beck, M. Dzaizka H. Bohme, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 1996. ISBN: 0-201-87741-4.

[HBT06] Jorrit N. Herder, Herbert Bos, and Andrew S. Tanenbaum. A lightweight method for building reliable operating systems despite unreliable device drivers. Technical Report IR-CS-018, Vrije Universiteit, Amsterdam, 2006. `http://www.cs.vu.nl/~jnherder/ir-cs-018.pdf`.

[Kle86] S. Kleiman. Vnodes: An architecture for multiple file system types in sun unix. *USENIX Conference*, pages 238–247, 1986.

[Ltd06] QNX Software Systems Ltd. *QNX Documentation Library*. QNX Website, 2006. `http://www.qnx.com`.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996. ISBN: 0-201-54979-4.

[Pat03] Steve D. Pate. *UNIX Filesystems: Evolution, Design and Implementation*. Wiley Publishing, 2003. ISBN: 0-471-16483-6.

[Tre04] J. Treadwell. Open grid services architecture glossary of terms. 2004. `http://www.ggf.org/Meetings/GGF12/Documents/draft-ggf-ogsa-glossary.pdf`.

Appendix A

VFS/FS interface

This Appendix covers the details of the VFS/FS interface. It is organized according to the request codes that the VFS sends to an underlying file system implementation, thus the interface is shown from the file system implementation point of view. The fields of the input and output messages are given and the desired behavior of the file system implementation is described.

The possible requests that the VFS can send to the FS process is declared in the *"include/minix/vfsif.h"* header file. This file also contains the macro declarations for the mappings used in the fields of the interface messages.

The Appendix is structured as follows. First an overall view of the request message types is given in Table A. Then, each request type is considered one by one. Only the additional part of the message type is shown. The m_type field of the message is determined by the request code. (See Section 3.2.1 for avoiding confusion.) The exact input and output messages are shown in tables with three columns. The type, the mapping macro declaration and a short description is given for each field of the message. The operation that the FS server is supposed to perform is also given.

The following request types are used during the interaction between the VFS and the actual file system implementations: [1]

Request type	**Description**
REQ_GETNODE	Increase node's counter
REQ_PUTNODE	Decrease node's counter
REQ_OPEN	Open file
REQ_PIPE	Create pipe
REQ_READ	Read file
REQ_WRITE	Write file
REQ_CLONE_OPCL	Create temporary character special file
REQ_TRUNC	Truncate a file
REQ_FTRUNC	Truncate an already opened file
REQ_CHOWN	Change owner/group
REQ_CHMOD	Change mode
REQ_ACCESS	Check permissions
REQ_MKNOD	Create a special file
REQ_MKDIR	Create a directory
REQ_INHIBREAD	Inhibit read ahead
REQ_STAT	Get file's statistics
REQ_FSTAT	Get an already opened file's statistics
REQ_UNLINK	Unlink file
REQ_RMDIR	Remove directory
REQ_UTIME	Set file time
REQ_FSTATFS	Get file system statistics
REQ_GETDIR	Increase counter for directory file
REQ_LINK	Create hard link
REQ_SLINK	Create soft link
REQ_RDLINK	Read soft link
REQ_RENAME	Rename file
REQ_MOUNTPOINT	Register mount point
REQ_READSUPER	Read superblock and root node
REQ_UNMOUNT	Unmount partition
REQ_SYNC	Sync buffer cache
REQ_LOOKUP	Lookup path name
REQ_STIME	Set time
REQ_BREAD	Read block spec file
REQ_BWRITE	Write block spec file

Table A.1: VFS/FS interface request types.

[1] REQ_STAT and REQ_FSTAT, REQ_TRUNC and REQ_FTRUNC are different from the inode handling point of view.

A.1 Request, operation, response

This section details the interface messages one by one. Each request is described as the request message, the operation that the FS is supposed to perform and the response message. In case the response message is not detailed the FS server is supposed to transfer OK or the appropriate error code in the m_type of the response message.

A.1.1 REQ_GETNODE

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number

Desired FS behavior:

The FS server finds the inode in the inode table and increases the counter. If the inode is not in the table it has to be loaded. It fills the response message's fields and reports OK. In case of the inode can not be found or loaded EINVAL has to be returned.

Response message fields:

Type	Field	Description
int	RES_INODE_NR	Inode number
short	RES_MODE	File type, mode, etc.
int	RES_FILE_SIZE	File size
int	RES_DEV	Device number for special files

A.1.2 REQ_PUTNODE

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
short	REQ_INODE_INDEX	Index in the inode table

Desired FS behavior:

The FS server finds the inode in the inode table and decreases the counter. In case of the inode can not be found EINVAL has to be returned.

A.1.3 REQ_OPEN

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
short	REQ_UID	User ID
char	REQ_GID	Group ID
int	REQ_FLAGS	Open flags
short	REQ_MODE	Open mode
char*	REQ_PATH	Last components address (in the VFS)
short	REQ_PATH_LEN	Last component's length

Desired FS behavior:

If the creat flag is set the FS server transfers the last component of the path name and tries to create the file, otherwise the FS server finds the inode in the inode table and increases the counter. If the inode is not in the table it has to be loaded. The FS server checks the permission for the specified mode. The appropriate error code has to transfered back in case of error. Otherwise, the FS fills the response message's fields and reports OK.

Response message fields:

Type	Field	Description
int	RES_INODE_NR	Inode number
short	RES_MODE	File type, mode, etc.
int	RES_FILE_SIZE	File size
int	RES_DEV	Device number (for special files)
int	RES_INODE_INDEX	Index in inode table
short	RES_UID	Owner's user ID
char	RES_GID	Owner's group ID
int	RES_CTIME	Time of inode's change (for regular files)

A.1.4 REQ_PIPE

Request message fields:

Type	Field	Description
short	REQ_UID	User ID
char	REQ_GID	Group ID

Desired FS behavior:

The FS server allocates an inode, sets the usage counter to two and the file type to I_PIPE. It fills the response message's fields and reports OK. In case of the inode can not be allocated the appropriate error code has to be transfered back.

Response message fields:

Type	Field	Description
int	RES_INODE_NR	Inode number
short	RES_MODE	File type, mode, etc.
int	RES_FILE_SIZE	File size
int	RES_INODE_INDEX	Index in inode table

A.1.5 REQ_READ

Request message fields:

Type	Field	Description
int	REQ_FD_INODE_NR	Inode number
short	REQ_FD_INODE_INDEX	Index in the inode table
int	REQ_FD_WHO_E	User process' kernel endpoint
int	REQ_FD_SEG	Segment
int	REQ_FD_POS	Position
int	REQ_FD_NBYTES	Number of bytes to be transferred
char*	REQ_FD_USER_ADDR	User's buffer address

Desired FS behavior:

The FS server transfers at most REQ_FD_NBYTES bytes from the specified position of the specified file to the user buffer. It reports success and the number of bytes transferred or sends the appropriate error code.

Response message fields:

Type	Field	Description
int	RES_FD_POS	New position
int	RES_FD_CUM_IO	Bytes transferred
int	RES_FD_SIZE	File size

A.1.6 REQ_WRITE

Request message fields:

Type	Field	Description
int	REQ_FD_INODE_NR	Inode number
short	REQ_FD_INODE_INDEX	Index in the inode table
int	REQ_FD_WHO_E	User process' kernel endpoint
int	REQ_FD_SEG	Segment
int	REQ_FD_POS	Position
int	REQ_FD_NBYTES	Number of bytes to be transferred
char*	REQ_FD_USER_ADDR	User's buffer address

Desired FS behavior:

The FS server transfers at most REQ_FD_NBYTES bytes from the user buffer to the user specified position of the specified file. It reports success, the number of bytes transferred and the new file size or sends the appropriate error code.

Response message fields:

Type	Field	Description
int	RES_FD_POS	New position
int	RES_FD_CUM_IO	Bytes transferred
int	RES_FD_SIZE	File size

A.1.7 REQ_CLONE_OPCL

Request message fields:

Type	Field	Description
int	REQ_DEV	Device number

Desired FS behavior:

The FS server allocates an inode and sets the mode to character special file, device number is specified in the request message. Since interaction with character special files are performed on the VFS layer, the real purpose of this operation is to provide a valid inode which can be fstated. The FS server fills the response message's fields and reports OK. Otherwise the appropriate error value has to be transferred back.

Response message fields:

Type	Field	Description
int	RES_INODE_NR	Inode number
short	RES_MODE	File type, mode, etc.
int	RES_FILE_SIZE	File size
int	RES_DEV	Device number for special files

A.1.8 REQ_TRUNC

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
short	REQ_UID	User ID
char	REQ_GID	Group ID
int	REQ_LENGTH	New file length

Desired FS behavior:

The FS server loads the inode and changes the file size. Note that it is possible that the inode is not in use, in this case the FS server has to load the inode from the disk. It reports success or the appropriate error value has to be transferred back.

A.1.9 REQ_FTRUNC

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
short	REQ_UID	User ID
char	REQ_GID	Group ID
int	REQ_FD_START	Start position of freeing
int	REQ_FD_END	End position of freeing

Desired FS behavior:

This request is also used by the fcnt() for freeing a section of a file. The FS server finds the inode and frees the file section or changes the file size. Zero in REQ_FD_END indicates the simple truncate, where the new file size is defined in the REQ_FD_START field of the message. Note that it is not possible that the inode is not in use, in this case the FS server has to report error, otherwise it reports success.

A.1.10 REQ_CHOWN

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
short	REQ_UID	User ID
char	REQ_GID	Group ID
short	REQ_NEW_UID	New user ID
char	REQ_NEW_GID	New group ID

Desired FS behavior:

The FS server has load the inode and check whether the caller has the permission to perform the change. If the caller is permitted then the FS assigns the new values. It reports success or the appropriate error value.

A.1.11 REQ_CHMOD

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
short	REQ_UID	User ID
char	REQ_GID	Group ID
short	REQ_MODE	New mode

Desired FS behavior:

The FS server loads the inode and checks whether the caller has the permission to perform the change. If the caller is permitted then the FS assigns the new value. It reports success or the appropriate error value.

A.1.12 REQ_ACCESS

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
short	REQ_UID	User ID
char	REQ_GID	Group ID
short	REQ_MODE	New mode

Desired FS behavior:

The FS server loads the inode and checks whether the caller has the permission specified in the REQ_MODE field of the message. It reports OK or the appropriate error value.

A.1.13 REQ_MKNOD

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number of the parent directory
short	REQ_UID	User ID
char	REQ_GID	Group ID
short	REQ_MODE	Specified mode
int	REQ_DEV	Device number
char*	REQ_PATH	Last component's address (in the VFS)
short	REQ_PATH_LEN	Last component's length

Desired FS behavior:

The FS server is supposed to allocate a new inode and set the mode and the device number according to the request message, create a directory entry in the directory specified by the REQ_INODE_NR with the name specified by the REQ_PATH. It reports success or the appropriate error value.

A.1.14 REQ_MKDIR

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number of the parent directory
short	REQ_UID	User ID
char	REQ_GID	Group ID
char*	REQ_PATH	Last component's address (in the VFS)
short	REQ_PATH_LEN	Last component's length

Desired FS behavior:

The FS server is supposed to allocate a new inode, create a directory entry in the directory specified by the REQ_INODE_NR with the name specified by the REQ_PATH. It has to create the "." and ".." entries in the directory just created. It reports success or the appropriate error value.

A.1.15 REQ_INHIBREAD

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number of the parent directory

Desired FS behavior:

The FS finds the inode in the inode table and turns the readahead feature on the given inode off. It reports success or the appropriate error value.

A.1.16 REQ_STAT

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
int	REQ_WHO_E	User process' kernel endpoint
char*	REQ_USER_ADDR	User's buffer address

Desired FS behavior:

The FS server loads the inode specified by the REQ_INODE_NR field of the message, fills in a stat structure and transfers back to the caller process to the address specified by the REQ_USER_ADDR. Note that it is possible that the FS server has to load the inode details from the disk, the inode does not have to be necessarily in the inode table. It report success or the appropriate error value.

A.1.17 REQ_FSTAT

Request message fields:

Type	Field	Description
int	REQ_FD_INODE_NR	Inode number
int	REQ_FD_WHO_E	User process' kernel endpoint
int	REQ_FD_POS	Current position in the pipe (if a pipe is fstated)
char*	REQ_FD_USER_ADDR	User's buffer address

Desired FS behavior:

The FS server finds the inode in the inode table, fills in a stat structure and transfers back to the caller process to the address specified by the REQ_FD_USER_ADDR field of the message. Note that during an fstat request the specified inode has to be in use and therefore in the inode table, in case if it is not there it is an error. It report success or the appropriate error value.

A.1.18 REQ_UNLINK

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number of the parent directory
short	REQ_UID	User ID
char	REQ_GID	Group ID
char*	REQ_PATH	Last component's address (in the VFS)
short	REQ_PATH_LEN	Last component's length

Desired FS behavior:

The FS server is supposed to check whether the caller has the permission to perform the unlink. If it has the FS deletes the directory entry specified by the REQ_PATH field of the message from the directory specified by the REQ_INODE_NR. The link counter of the file's inode has to be decreased. The FS reports success or the appropriate error value.

A.1.19 REQ_RMDIR

Request message fields:

Type	**Field**	**Description**
int	REQ_INODE_NR	Inode number of the parent directory
short	REQ_UID	User ID
char	REQ_GID	Group ID
char*	REQ_PATH	Last component's address (in the VFS)
short	REQ_PATH_LEN	Last component's length

Desired FS behavior:

The FS server is supposed to check whether the caller has the permission to perform the rmdir. If it has the FS checks for emptiness and deletes the directory specified by the REQ_PATH field of the message from the directory specified by the REQ_INODE_NR. Note that the "." and ".." entries from the directory have to be erased too. The FS reports success or the appropriate error value.

A.1.20 REQ_UTIME

Request message fields:

Type	**Field**	**Description**
int	REQ_INODE_NR	Inode number
short	REQ_UID	User ID
char	REQ_GID	Group ID
int	REQ_ACTIME	Access time
int	REQ_MODTIME	Modification time

Desired FS behavior:

The FS server check the permission whether the caller is allowed to change the times or not. It changes the inode's time values if the caller is permitted. It report success or the appropriate error value.

A.1.21 REQ_FSTATS

Request message fields:

Type	Field	Description
int	REQ_FD_INODE_NR	Inode number
int	REQ_FD_WHO_E	User process' kernel endpoint
int	REQ_FD_POS	Current position in the pipe (if a pipe is fstated)
char*	REQ_FD_USER_ADDR	User's buffer address

Desired FS behavior:

The FS server finds the inode in the inode table, fills in a stat structure and transfers back to the caller process to the address specified by the REQ_FD_USER_ADDR field of the message. Note that during an fstats request the specified inode has to be in use and therefore in the inode table, in case of it is not there it is an error. It report success or the appropriate error value.

A.1.22 REQ_GETDIR

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
short	REQ_UID	User ID
char	REQ_GID	Group ID

Desired FS behavior:

The FS server loads the inode and checks whether it is a directory and if the caller has the execution permission (i.e. it can browse the directory). If so, the usage counter is to be increased and inode's details is transferred back. Otherwise the appropriate error value has to be transferred back. Note that this request is different from the REQ_GETNODE, since the permissions and the file type has to be checked here.

Response message fields:

Type	Field	Description
int	RES_INODE_NR	Inode number
short	RES_MODE	File type, mode, etc.
int	RES_FILE_SIZE	File size

A.1.23 REQ_LINK

Request message fields:

Type	Field	Description
short	REQ_UID	User ID
char	REQ_GID	Group ID
int	REQ_LINKED_FILE	File to be linked
int	REQ_LINK_PARENT	Parent directory of the link
char*	REQ_PATH	Last component's address (in the VFS)
short	REQ_PATH_LEN	Last component's length

Desired FS behavior:

The FS server creates a new directory entry in the directory specified with REQ_LINK_PARENT with the name contained in REQ_PATH. The inode of the entry is specified in REQ_LINKED_FILE. The link counter of the inode has to be increased. FS reports success or the appropriate error value.

A.1.24 REQ_SLINK

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Parent directory of the link
short	REQ_UID	User ID
char	REQ_GID	Group ID
char*	REQ_PATH	Last component's address (in the VFS)
short	REQ_PATH_LEN	Last component's length
int	REQ_WHO_E	User process' kernel endpoint
char*	REQ_USER_ADDR	Symbolic link path
short	REQ_SLENGTH	Length of the path

Desired FS behavior:

The FS server is supposed to allocate an inode, create a new directory entry with the name contained in REQ_PATH in the directory specified by REQ_INODE_NR and transfer the path name from the user address REQ_USER_ADDR into the file. The type of the file has to be symlink. FS reports success or the appropriate error value.

A.1.25 REQ_RDLINK

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Parent directory of the link
short	REQ_UID	User ID
char	REQ_GID	Group ID
int	REQ_WHO_E	User process' kernel endpoint
char*	REQ_USER_ADDR	Symbolic link path
short	REQ_SLENGTH	Length of the path

Desired FS behavior:

The FS server loads the link's inode and transfers the link content to the user address specified by the message. At most REQ_SLENGTH number of bytes is allowed to be transfered. FS reports success or the appropriate error value.

A.1.26 REQ_RENAME

Request message fields:

Type	Field	Description
int	REQ_OLD_DIR	Inode number of the old parent directory
int	REQ_NEW_DIR	Inode number of the new parent directory
short	REQ_UID	User ID
char	REQ_GID	Group ID
char*	REQ_PATH	Old name's address (in the VFS)
short	REQ_PATH_LEN	Old name's length
char*	REQ_USER_ADDR	New name's address (in the VFS)
short	REQ_SLENGTH	New name's length

Desired FS behavior:

The FS server is supposed to remove the directory entry from the REQ_OLD_DIR directory with the name specified by the REQ_PATH, the old file name. A new directory entry has to be created in the REQ_NEW_DIR directory with the name specified by the REQ_USER_ADDR and associated to the same inode that was used by the old name. Depending on the underlying file system type, some sanity check have to be performed:

- Parent directories must be writable, searchable.

- The old inode must not be a superdirectory of the new parent directory.

FS reports success or the appropriate error value.

A.1.27 REQ_MOUNTPOINT

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number
short	REQ_UID	User ID
char	REQ_GID	Group ID

Desired FS behavior:

The FS process loads the inode and checks whether the inode is in use or not. Only directory inodes that are not in use are allowed to be a mountpoint. The FS registers that this inode is a mount point and sends back it's details.

Response message fields:

Type	Field	Description
int	RES_INODE_NR	Inode number
short	RES_MODE	File type, mode, etc.
int	RES_FILE_SIZE	File size

A.1.28 REQ_READSUPER

Request message fields:

Type	Field	Description
char	REQ_READONLY	Mount read-only?
int	REQ_BOOTIME	System boottime timestamp
int	REQ_DRIVER_E	Device driver process' endpoint (on which the file system lives)
int	REQ_DEV	Device number
char*	REQ_SLINK_STORAGE	Buffer address in the VFS address space where symbolic link's content can be copied

Desired FS behavior:

The FS process reads and checks the superblock of the partition. It loads the root inode and fills in the response message fields. It reports success or the appropriate error value.

Response message fields:

Type	Field	Description
int	RES_INODE_NR	Inode number
short	RES_MODE	File type, mode, etc.
int	RES_FILE_SIZE	File size

A.1.29 REQ_UNMOUNT

Desired FS behavior:

The FS process checks the inode table and counts the in use inodes. If there is only one – the root inode, which is hold by the mount –, the unmount can be performed. The FS process reports success or the appropriate error value.

A.1.30 REQ_SYNC

Desired FS behavior:

The FS process writes the whole buffer cache to the disk. This operation can fail only in one case, namely if the driver process dies. In this case the VFS will reissue the request after the new driver endpoint has mapped. The FS reports success.

A.1.31 REQ_LOOKUP

Request message fields:

Type	Field	Description
int	REQ_INODE_NR	Inode number of the starting directory
int	REQ_CHROOT_NR	Inode number of the process' root directory
int	REQ_FLAGS	Lookup action flag
short	REQ_UID	User ID
char	REQ_GID	Group ID
char*	REQ_PATH	Path string's address (in the VFS)
short	REQ_PATH_LEN	Path name's length
char*	REQ_USER_ADDR	Address where the last component can be stored (in the VFS)
char	REQ_SYMLOOP	Symbolic link loop counter

Desired FS behavior:

The FS process is supposed to process the path, translate it to an inode object and transfer back the inode details. Although during the lookup many things can happen:

- **Encountering a mount point:** the FS process is supposed to send back the inode number of the mount point. The response message also contains the number of characters that were processed and the current value of the symbolic link loop counter. The m_type field of the message has the EENTER_MOUNT value.

- **Leaving the partition:** the FS process is supposed to send back the number of characters that were processed and the current value of the symbolic link loop counter. The m_type field of the message has the ELEAVE_MOUNT value.

- **Symbolic link with absolute path:** the FS process is supposed to transfer back in the VFS process the new path name to the address that was specified by the REQ_SLINK_STORAGE field of the readsuper request and the new value of the symbolic link loop counter. The m_type field of the message has the ESYMLINK value.

In case the FS server encounters a mountpoint the following message is sent, (note that the message type is EENTER_MOUNT).

Response message fields:

Type	**Field**	**Description**
int	RES_INODE_NR	Inode number
int	RES_OFFSET	Number of characters processed
char	RES_SYMLOOP	Number of times symbolic links were started to be processed

In case of a successful lookup the FS process sends back the details of the inode. Otherwise the appropriate error value has to be transferred back.

Response message fields:

Type	**Field**	**Description**
int	RES_INODE_NR	Inode number
short	RES_MODE	File type, mode, etc.
int	RES_FILE_SIZE	File size
int	RES_DEV	Device number for special files

A.1.32 REQ_STIME

Request message fields:

Type	Field	Description
int	REQ_BOOTTIME	New boottime timestamp

Desired FS behavior:

The FS server stores the boottime value and uses is as a base reference in the clock_time() function.

A.1.33 REQ_BREAD

Request message fields:

Type	Field	Description
int	REQ_FD_BDEV	Device number of the block spec file
short	REQ_FD_BLOCK_SIZE	Block size
int	REQ_FD_WHO_E	User process' kernel endpoint
int	REQ_FD_POS	Position
int	REQ_FD_NBYTES	Number of bytes to be transferred
char*	REQ_FD_USER_ADDR	User's buffer address

Desired FS behavior:

The FS server transfers at most REQ_FD_NBYTES bytes from the specified position of the specified device to the user buffer. It reports success and the number of bytes transferred or sends the appropriate error code.

Response message fields:

Type	Field	Description
int	RES_FD_POS	New position
int	RES_FD_CUM_IO	Bytes transferred

A.1.34 REQ_BWRITE

Request message fields:

Type	Field	Description
int	REQ_FD_BDEV	Device number of the block spec file
short	REQ_FD_BLOCK_SIZE	Block size
int	REQ_FD_WHO_E	User process' kernel endpoint
int	REQ_FD_POS	Position
int	REQ_FD_NBYTES	Number of bytes to be transferred
char*	REQ_FD_USER_ADDR	User's buffer address

Desired FS behavior:

The FS server transfers at most REQ_FD_NBYTES bytes to the specified position of the specified device from the user buffer. It reports success and the number of bytes transferred or sends the appropriate error code.

Response message fields:

Type	Field	Description
int	RES_FD_POS	New position
int	RES_FD_CUM_IO	Bytes transferred

Appendix B

How to implement a new file system...

This Appendix aims to provide a tutorial how to write a file system implementation. Note that main goal is to show how to structure the code in order to easily cooperate with the VFS through the interface described above. Therefore, we will not provide a real on-disk file system driver here.[1] As an example, a binary tree file system is given.

B.1 The Binary Tree file system

The binary tree file system is a read-only file system that consists only of directories. Each directory has two subdirectories with the names ”0” and ”1” [2]. The root directories inode has the value 1. The ”0” subdirectory of the directory with the inode number N has the inode value $2*N$ while the subdirectory ”1” has the inode number $2*N+1$. The file system can be mounted and unmounted. The directories can be traversed, opened and read. Each directory can be stated, fstated, lstated and accessed. Thus, the implementation of the operations that have been just enumerated will be described.

B.2 Main program of the file system server

Listing B.1 shows the code of the file server's main program.

[1]For a real file system implementation please consult the MINIX file system driver source under *”/usr/src/servers/mfs”*.

[2]Actually four, because each directory has to contain the ”.” and ”..” subdirectories.

Listing B.1: Main program of the file server

```
PUBLIC int main(void)
{
 int who_e;                        /* caller */
 int error;
 message m_in, m_out;

 /* Initialize the server, then go to work. */
 init_server();

 m_in.m_type = FS_READY;
 if (sendrec(VFS_PROC_NR, &m_in) != OK) {
     return -1;
 }

 /* Check for proper reply */
 if (m_in.m_type != REQ_READSUPER) {
     return -1;
 }
 else {
     /* Read superblock and root inode */
     m_out.m_type = fs_readsuper(&m_in, &m_out);
     reply(VFS_PROC_NR, &m_out);
     if (m_out.m_type != OK) return -1;
 }

 for (;;) {
     /* Wait for request message. */
     get_work(&m_in);

     who_e = m_in.m_source;
     req_nr = m_in.m_type;

     if (req_nr < 0 || req_nr >= NREQS) {
         error = EINVAL;
     }
     else {
         /* Process request */
         error = (*fs_call_vec[req_nr])(&m_in, &m_out);
     }

     /* Send reply */
     m_out.m_type = error;
     reply(who_e, &m_out);
 }
}
```

The main program of a file system server contains the following steps:

- Initialization of the server program.
- Login to the VFS.
- Main (infinite) loop that receives a request, performs the operation and sends the reply.

Usually file servers have similar structure with this. Minor differences are possible like handling the read ahead feature, which takes place after processing the current request.

B.3 Inode handling

The binary tree file system works as an inode based file system. Inodes are representing files, although in this particular case there are only directories. The fields of the inode structure are shown in listing B.2.

Listing B.2: The inode table

```
struct inode {
    mode_t i_mode;          /* file type, protection, etc. */
    dev_t i_dev;            /* which device is the inode on */
    off_t i_size;           /* current file size in bytes */
    int i_count;            /* usage counter of this inode */
    ino_t i_num;            /* inode number */
    int i_nlinks;           /* number of links refer this inode */
} inode[NR_INODES];
```

The server stores the in-use inodes in a static array. For managing them, a couple of functions have to be introduced. There are three important ones, loading, finding and dropping the inode. The inode loading functions first checks the inode table whether the inode to be loaded is already in use. It increases its counter, or loads the values and assigns the value one to the usage counter in case if it was not in use. The function:

Listing B.3: The inode loader function

```
PUBLIC struct inode *get_inode(dev, numb)
dev_t dev;                /* device on which inode resides */
int numb;                 /* inode number  */
{
 register struct inode *rip, *xp;

 /* Search the inode table both for (dev, numb) and a free slot. */
 xp = NIL_INODE;
 for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++) {
     /* only check used slots for (dev, numb) */
11         if (rip->i_count > 0) {
12             if (rip->i_dev == dev && rip->i_num == numb) {
13                 /* This is the inode that we are looking for. */
14                 rip->i_count++;
15                 return(rip);        /* (dev, numb) found */
16             }
17         }
18         else {
19             xp = rip;    /* remember this free slot for later */
20         }
21     }
22 
23     /* Inode we want is not currently in use.
24      * Did we find a free slot? */
25     if (xp == NIL_INODE) {     /* inode table completely full */
26         err_code = ENFILE;
27         return(NIL_INODE);
28     }
29 
30     /* A free inode slot has been located.
31      * Load the inode into it. */
32     xp->i_dev = dev;
33     xp->i_num = numb;
34     xp->i_mode = I_DIRECTORY | ALL_MODES;
35     xp->i_size = 4 * sizeof(struct _fl_direct);
36     xp->i_count = 1;
37     xp->i_nlinks = 3;
38     return(xp);
39 }
```

The binary tree file system's inodes are all directories and accessible for everyone. Properties are given between the 32*nd* and the 37*th* lines. Each file contains exactly four flexible directory entries, therefore the size of them is 4 times the structure size. Note that this values would be filled in from the disk in a real file system implementation.

In many cases the file system server has to find an in-use inode. For this purpose an inode finder function is defined:

Listing B.4: The inode finder function

```
PUBLIC struct inode *find_inode(dev, numb)
dev_t dev;                /* device on which inode resides */
int numb;                 /* inode number */
{
 struct inode *rip;

 for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++) {
     if (rip->i_count > 0) {
         /* only check used slots for (dev, numb) */
```

```
            if (rip->i_dev == dev && rip->i_num == numb) {
                /* This is the inode that we are looking for. */
                return(rip);          /* (dev, numb) found */
            }
        }
    }

    return NIL_INODE;
}
```

Note that this function does not modify the usage counter of the inode. For dropping an inode (e.g. decreasing the usage counter) the following function is defined:

Listing B.5: The inode dropper function

```
PUBLIC void put_inode(rip)
register struct inode *rip  /* pointer to inode to be released */
{
 /* checking here is easier than in caller */
 if (rip == NIL_INODE) return;

 --rip->i_count;
}
```

This routine is a simplified form of inode loader for the case where the inode pointer is already known. It increases the usage counter:

Listing B.6: The inode duplicator function

```
PUBLIC void dup_inode(rip)
register struct inode *rip  /* Inode to be duplicated */
{
/* Increase usage counter */
 ++rip->i_count;
}
```

Inode handling routines will be used in many request operations. Although, in a real file system implementation these functions can be more complicated they can follow the same structure as it was shown.

B.4 Request operations

This section describes the implementation of the supported requests.

B.4.1 Readsuper

First the mount request is shown:

Listing B.7: Reading superblock and root inode.

```
PUBLIC int fs_readsuper(message *m_in, message *m_out)
{
  struct inode *root_ip;

  /* Get input message values */
  fs_dev = m_in->REQ_DEV;
  fs_driver_e = m_in->REQ_DRIVER_E;
  boottime = m_in->REQ_BOOTTIME;
  vfs_slink_storage = m_in->REQ_SLINK_STORAGE;

  /* Get the root inode of the mounted file system. */
  root_ip = NIL_INODE;
  if ((root_ip = get_inode(fs_dev, ROOT_INODE)) == NIL_INODE)
    return EINVAL;

  /* Root inode properties */
  m_out->RES_INODE_NR = root_ip->i_num;
  m_out->RES_MODE = root_ip->i_mode;
  m_out->RES_FILE_SIZE = root_ip->i_size;

  /* Partition properties */
  m_out->RES_MAXSIZE = 1024;
  m_out->RES_BLOCKSIZE = 0;

  return OK;
}
```

The binary tree file system does not have a superblock. Therefore no superblock reading operation is present, although in a real file system implementation the reading of the superblock takes place here. It is worth mentioning that the FS server gets the driver process endpoint number in the readsuper request. It also receives the device number, the boottime time stamp and the buffer address in the VFS address space where the content of a symbolic link can be stored.

B.4.2 Unmount

The implementation of the unmount request:

Listing B.8: Unmounting the partition.

```
PUBLIC int fs_unmount(message *m_in, message *m_out)
{
  int count;
  register struct inode *rip;

  /* Count in use inodes */
  count = 0;
  for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++) {
        if (rip->i_count > 0 && rip->i_dev == fs_dev) {
                count += rip->i_count;
        }
  }

  /* Only the root inode should be in use with
   * the counter value 1 */
  if (count > 1) {
      return EBUSY;    /* Can't umount a busy file system */
  }

  /* Put the root inode */
  rip = find_inode(fs_dev, ROOT_INODE);
  put_inode(rip);

  return OK;
}
```

The unmount operation counts the in use inodes. The partition can be unmounted only if there is one inode in use, the root inode which is held by the corresponding virtual mount object in the VFS process. Otherwise the partition is considered to be busy. Real file system implementations usually call the sync operation after dropping the root inode so that everything gets copied back to the disk.

B.4.3 Stime

Setting the boot time is done by the following simple routine:

Listing B.9: Setting boot time stamp.

```
PUBLIC int fs_stime(message *m_in, message *m_out)
{
  boottime = m_in->REQ_BOOTTIME;
  return OK;
}
```

B.4.4 Get node

For the get node request the FS has to load (or find) the inode specified by the request message and increase its usage counter. The implementation:

Listing B.10: Getting an inode.

```
PUBLIC int fs_getnode(message *m_in, message *m_out)
{
  struct inode *rip;

  /* Get the inode */
  if ((rip = get_inode(fs_dev, m_in->REQ_INODE_NR))
                  == NIL_INODE)
        return EINVAL;

  /* Transfer back the inode's details */
  m_out->m_source = rip->i_dev;
  m_out->RES_INODE_NR = rip->i_num;
  m_out->RES_MODE = rip->i_mode;
  m_out->RES_FILE_SIZE = rip->i_size;

  return OK;
}
```

B.4.5 Put node

The put node request aims to decrease an in use inode's usage counter:

Listing B.11: Dropping an inode.

```
PUBLIC int fs_putnode(message *m_in, message *m_out)
{
  struct inode *rip;

  /* Find the inode */
  if ((rip = find_inode(fs_dev, m_in->REQ_INODE_NR))
                  == NIL_INODE)
        return EINVAL;


  put_inode(rip);
  return OK;
}
```

B.4.6 Path name lookup

In order to describe the path name lookup some additional functions have to be introduced. Although the path name lookup in the binary tree file system could be done in a very simple way, this section tries to give a common solution so that the general method of this functionality can also be shown. The whole lookup mechanism is built of four functions. The first is responsible for breaking the path name string into components:

Listing B.12: Getting the next component of a path name.

```
PRIVATE char *get_name(old_name, string)
char *old_name;           /* path name to parse */
char string[NAME_MAX];    /* component extracted from 'old_name' */
{
  register int c;
  register char *np, *rnp;

  np = string;       /* 'np' points to current position */
  rnp = old_name;    /* 'rnp' points to unparsed string */
  while ( (c = *rnp) == '/') {
      rnp++;      /* skip leading slashes */
      path_processed++; /* count characters */
  }

  /* Copy the unparsed path, 'old_name',
   * to the array, 'string'. */
  while (rnp < &old_name[PATH_MAX] && c != '/' && c != '\0') {
      if (np < &string[NAME_MAX]) *np++ = c;
      c = *++rnp;                  /* advance to next character */
      path_processed++;            /* count characters */
  }

  /* Skip trailing slashes. */
  while (c == '/' && rnp < &old_name[PATH_MAX]) {
      c = *++rnp;
      path_processed++;            /* count characters */
  }

  if (np < &string[NAME_MAX])
      *np = '\0';              /* Terminate string */

  if (rnp >= &old_name[PATH_MAX]) {
      err_code = ENAMETOOLONG;
      return((char *) 0);
  }
  return rnp;
}
```

The function skips the leading slashes, copies the next component to the character array given as the second parameter, then skips the trailing slashes and terminates the string. It also counts the characters that were processed during the operation.

The next function is used for looking up a component in a given directory:

Listing B.13: Looking up a component.

```
PUBLIC struct inode *advance(dirp, string, m_out)
struct inode *dirp;      /* inode for directory to be searched */
char string[NAME_MAX];   /* component name to look for */
message *m_out;          /* reply message */
{
  register struct inode *rip;

  /* Check for NIL_INODE. */
  if (dirp == NIL_INODE) return NIL_INODE;

  /* If 'string' is empty, yield same inode straight away. */
  if (string[0] == '\0')
      return get_inode(dirp->i_dev, dirp->i_num);

  /* "Find component" */
  if (!strcmp(string, "1"))
      rip = get_inode(dirp->i_dev, dirp->i_num * 2 + 1);
  else if (!strcmp(string, "0"))
      rip = get_inode(dirp->i_dev, dirp->i_num * 2);
  else if (!strcmp(string, "..")) {
      if (dirp->i_num == ROOT_INODE) {
          err_code = ELEAVEMOUNT;
          m_out->RES_MOUNTED = 0;
          m_out->RES_OFFSET = path_processed;
          m_out->RES_SYMLOOP = symloop;
          rip = NIL_INODE;
      }
      else
          rip = get_inode(dirp->i_dev, dirp->i_num / 2);
  }
  else if (!strcmp(string, ".")) {
      dup_inode(dirp);
      rip = dirp;
  }
  else {
      err_code = ENOENT;
      return NIL_INODE;
  }
  return(rip);  /* return pointer to inode's component */
}
```

As we mentioned before the binary tree file system has only four kind of file names. ”0”, ”1” and the two regular entry for directories ”.” and ”..”. Therefore the lookup is easy in this case. All the four strings are checked and the appropriate functionality is performed. In a real file system this function would call a low level routine that performs the search in the directory file. It is worth noting that leaving the partition is handled in this function, the reply message’s fields are filled in with the error value and a null inode pointer is returned.

The next function controls the whole lookup mechanism:

Listing B.14: Parsing the path.

```
PUBLIC struct inode *parse_path(path, string, action, m_in, m_out)
char *path;                    /* the path name to be parsed */
char string[NAME_MAX];         /* the final component is returned here */
int action;                    /* action on last part of path */
message *m_in;                 /* request message */
message *m_out;                /* reply message */
{
 struct inode *rip, *dir_ip;
 struct inode *ver_rip;
 char *new_name;
 char lstring[NAME_MAX];

 /* Find starting inode inode according to the request message */
 if ((rip = find_inode(fs_dev, m_in->REQ_INODE_NR)) == NIL_INODE) {
     err_code = ENOENT;
     return NIL_INODE;
 }

 /* Find chroot inode according to the request message */
 if (m_in->REQ_CHROOT_NR != 0) {
     if ((chroot_dir = find_inode(fs_dev, m_in->REQ_CHROOT_NR))
             == NIL_INODE) {
         err_code = ENOENT;
         return NIL_INODE;
     }
 }
 else
     chroot_dir = NIL_INODE;

 /* No characters were processed yet */
 path_processed = 0;

 dup_inode(rip);   /* inode will be returned with put_inode */

 /* Looking for the starting directory?
  * Note: this happens after EENTERMOUNT or ELEAVEMOUNT
  * without more path component */
  if (*path == '\0') {
       return rip;
  }

  if (string == (char *) 0) string = lstring;

  /* Scan the path component by component. */
  while (TRUE) {
      /* Extract one component. */
      if ((new_name = get_name(path, string)) == (char*)0) {
          put_inode(rip);          /* bad path in user space */
          return(NIL_INODE);
      }
      if (*new_name == '\0' && (action & PATH_PENULTIMATE)) {
          if ((rip->i_mode & I_TYPE) == I_DIRECTORY) {
              return(rip);         /* normal exit */
          }
          else {
              /* last file of path prefix is not a directory */
              put_inode(rip);
              err_code = ENOTDIR;
              return(NIL_INODE);
          }
      }

      /* There is more path.  Keep parsing. */
      dir_ip = rip;
      rip = advance(dir_ip, string, m_out);

      if (*new_name != '\0') {
          put_inode(dir_ip);
          path = new_name;
          continue;
      }

      /* Either last name reached or symbolic link is opaque */
      if ((action & PATH_NONSYMBOLIC) != 0) {
          put_inode(rip);
          return(dir_ip);
      }
      else {
          put_inode(dir_ip);
          return(rip);
      }
  }
}
```

The path parser finds the inode which represents the starting directory of the lookup and the chroot directory if it is specified in the request message. After a couple of sanity checks it starts breaking the path name into components and looking them up. It returns the inode with increased usage counter. The function that handles the actual lookup request:

Listing B.15: Path name lookup.

```
PUBLIC int lookup(message *m_in, message *m_out)
{
  char string[NAME_MAX];
  struct inode *rip;
  int s_error;

  string[0] = '\0';

  /* Copy the path name and set up caller's user and group ID */
  err_code = sys_datacopy(VFS_PROC_NR, (vir_bytes) m_in->REQ_PATH,
    SELF, (vir_bytes) user_path, (phys_bytes) m_in->REQ_PATH_LEN);

  if (err_code != OK) return err_code;

  caller_uid = m_in->REQ_UID;
  caller_gid = m_in->REQ_GID;

  /* Lookup inode */
  rip = parse_path(user_path, string, m_in->REQ_FLAGS, m_in, m_out);

  /* Copy back the last name if it is required */
  if ((m_in->REQ_FLAGS & LAST_DIR || m_in->REQ_FLAGS & LAST_DIR_EATSYM)
                  && err_code != ENAMETOOLONG) {
        s_error = sys_datacopy(SELF_E, (vir_bytes) string, VFS_PROC_NR,
                (vir_bytes) m_in->REQ_USER_ADDR, (phys_bytes)
                MIN(strlen(string)+1, NAME_MAX));
      if (s_error != OK) return s_error;
  }

  /* Error or mount point encountered */
  if (rip == NIL_INODE)
      return err_code;

  m_out->RES_INODE_NR = rip->i_num;
  m_out->RES_MODE = rip->i_mode;
  m_out->RES_FILE_SIZE = rip->i_size;

  /* Drop inode (path parse increased the counter) */
  put_inode(rip);
  return OK;
}
```

It copies the path name from the VFS' address space and sets up the caller's user and group ID according to the request, then it issues the path name parse and fills in the response message respectively. Finally the inode is dropped and success is reported.

B.4.7 Stat

As it was shown the binary tree file system is able to handle path name lookup. One of the most frequently called system call is the statistics about a file. In order to show how the stat request is implemented, an internal stat function has to be introduced, which is called by the stat and fstat requests.

The internal stat function is the following:

Listing B.16: Inode stat.

```
PRIVATE int stat_inode(rip, user_addr, who_e)
register struct inode *rip;  /* pointer to inode to stat */
char *user_addr;             /* user space address  */
int who_e;                   /* endpoint of the caller */
{
/* Common code for stat and fstat requests. */
  struct stat statbuf;
  int r;

  statbuf.st_dev = rip->i_dev;
  statbuf.st_ino = rip->i_num;
  statbuf.st_mode = rip->i_mode;
  statbuf.st_nlink = rip->i_nlinks;
  statbuf.st_uid = caller_uid;
  statbuf.st_gid = caller_gid;
  statbuf.st_rdev = NO_DEV;
  statbuf.st_size = rip->i_size;
  statbuf.st_atime = statbuf.st_mtime =
      statbuf.st_ctime = boottime;

  /* Copy the struct to user space. */
  r = sys_datacopy(SELF, (vir_bytes) &statbuf,
        who_e, (vir_bytes) user_addr, (phys_bytes) sizeof(statbuf));

  return r;
}
```

The routine fills in a stat structure and transfers it back to the address in user space. It is worth noting that in the binary tree file system each file is owned by the current caller and all the time stamps related to the files are the boot time stamp of the system. This values would be filled in by the inode data in case of a real file system.

The function that handles the stat request from the VFS is the following:

Listing B.17: Stat.

```
PUBLIC int fs_stat(message *m_in, message *m_out)
{
  register struct inode *rip;
  register int r;

  /* Load inode to be stated */
  if ( (rip = get_inode(fs_dev, m_in->REQ_INODE_NR))
                             == NIL_INODE)
        return(EINVAL);

  /* Call the internal stat function */
  r = stat_inode(rip, 0, m_in->REQ_USER_ADDR, m_in->REQ_WHO_E);
  put_inode(rip);                /* release the inode */
  return(r);
}
```

It loads the inode and calls the internal inode stat function.

B.4.8 Open

Although there are no regular files on the binary tree file system, the open service has to be provided by the FS in order to browse directories. In the binary tree case the open code simply increases the usage counter of the inode and transfers back the details. The implementation:

Listing B.18: Opening a file.

```
PUBLIC int fs_open(message *m_in, message *m_out)
{
  int r;
  struct inode *rip;

  /* Get file inode. */
  if ( (rip = get_inode(fs_dev, m_in->REQ_INODE_NR)) == NIL_INODE) {
      return ENOENT;
  }

  /* Reply message */
  m_out->RES_INODE_NR = rip->i_num;
  m_out->RES_MODE = rip->i_mode;
  m_out->RES_FILE_SIZE = rip->i_size;

  return OK;
}
```

B.4.9 Getdir

The getdir request is used when a process is about changing its working or chroot directory. In the binary tree file system case the FS has to load the inode and increase the usage counter, since each file is a directory and each directory is browsable on the file system. Although, in a real file system driver it has to be checked whether the caller has the right permissions in order to perform the change. The implementation is the following:

Listing B.19: Getting a directory.

```
PUBLIC int fs_getdir(message *m_in, message *m_out)
{
  register struct inode *rip;

  /* Try to open the new directory. */
  if ((rip = get_inode(fs_dev, m_in->REQ_INODE_NR))
                == NIL_INODE) {
        return(EINVAL);
  }

  /* If OK send back inode details */
  m_out->RES_INODE_NR = rip->i_num;
  m_out->RES_MODE = rip->i_mode;
  m_out->RES_FILE_SIZE = rip->i_size;

  return OK;
}
```

B.4.10 Access

In the binary tree file system case everyone is allowed to do anything. The access function is therefore simply returns OK:

Listing B.20: Checking permissions.

```
PUBLIC int fs_access(message *m_in, message *m_out)
{
  return OK;
}
```

B.4.11 Read

Since there are only directories on the binary tree file system and each directory contains the same directories – except that the inode numbers are different – reading from them equals with creating a pseudo content and copy it back to the user space address. The implementation:

Listing B.21: Reading from a file.

```
PUBLIC int fs_readwrite(message *m_in, message *m_out)
{
  int r, usr, seg, chunk;
  off_t position, nrbytes;
  unsigned int cum_io;
  char *user_addr;
  struct inode *rip;

  /* Four directory entries are in each directory */
  struct _fl_direct dirents[4];

  /* Find the inode referred */
  if ((rip = find_inode(fs_dev, m_in->REQ_FD_INODE_NR))
              == NIL_INODE) {
      return EINVAL;
  }

  /* Build pseudo directory data */
  dirents[0].d_ino = rip->i_num;           /* "." entry */
  dirents[0].d_extent = 0;
  dirents[0].d_name[0] = '.';
  dirents[0].d_name[1] = 0;

  dirents[1].d_ino = (rip->i_num != 1) ?
        (rip->i_num >> 1) : 1;             /* ".." entry */
  dirents[1].d_extent = 0;
  dirents[1].d_name[0] = '.';
  dirents[1].d_name[1] = '.';
  dirents[1].d_name[2] = 0;

  dirents[2].d_ino = rip->i_num * 2;       /* "0" entry */
  dirents[2].d_extent = 0;
  dirents[2].d_name[0] = '0';
  dirents[2].d_name[1] = 0;

  dirents[3].d_ino = rip->i_num * 2 + 1;   /* "1" entry */
  dirents[3].d_extent = 0;
  dirents[3].d_name[0] = '1';
  dirents[3].d_name[1] = 0;

  /* Get message fields */
  usr = m_in->REQ_FD_WHO_E;
  seg = m_in->REQ_FD_SEG;
  position = m_in->REQ_FD_POS;
  nrbytes = (unsigned) m_in->REQ_FD_NBYTES;
  user_addr = m_in->REQ_FD_USER_ADDR;

  /* Determine the number of bytes to copy */
  chunk = MIN(nrbytes, (sizeof(dirents) - position));

  /* Copy the chunk to user space. */
  r = sys_vircopy(SELF_E, D, (phys_bytes) ((char*)dirents + position),
        usr, seg, (phys_bytes) user_addr, (phys_bytes) chunk);

  position += chunk;
  cum_io = chunk;

  m_out->RES_FD_POS = position;
  m_out->RES_FD_CUM_IO = cum_io;
  m_out->RES_FD_SIZE = rip->i_size;
  return r;
}
```

The function finds the inode specified in the request message. It creates the pseudo file data with the four directory entries. Note that the inode numbers are computed according to the directory being read. It computes the number of bytes to be read. Since the user can request less than the size of the data buffer it has to be checked. It sends back the new file position and the number of bytes successfully transfered.

In a real file system implementation this function most probably calls a low level function that access the buffer cache to perform the data transfer.