MyFS: An Enhanced File System for MINIX

A Dissertation

Submitted in partial fulfillment of the requirement for the award of the degree of

MASTER OF ENGINEERING
(COMPUTER TECHNOLOGY & APPLICATIONS)

By

ASHISH BHAWSAR
College Roll No. 05/CTA/03
Delhi University Roll No. 3005

Under the guidance of
Prof. Asok De

Department Of Computer Engineering
Delhi College Of Engineering, New Delhi-110042
(University of Delhi)

July-2005

CERTIFICATE

This is to certify that the dissertation entitled "**MyFS: An Enhanced File System for MINIX**" submitted by **Ashish Bhawsar** in the partial fulfillment of the requirement for the award of degree of **Master of Engineering** in Computer Technology and Application, Delhi College of Engineering is an account of his work carried out under my guidance and supervision.

Professor D. Roy Choudhury

Head of Department
Department of Computer Engineering
Delhi College of Engineering
Delhi

Professor Asok De

Head of Department
Department of Information Technology
Delhi College of Engineering
Delhi

ACKNOWLEDGEMENT

It is a great pleasure to have the opportunity to extent my heartiest felt gratitude to everybody who helped me throughout the course of this project.

I would like to express my heartiest felt regards to **Dr. Asok De**, Head of the Department, Department of Information Technology for the constant motivation and support during the duration of this project. It is my privilege and owner to have worked under the supervision. His invaluable guidance and helpful discussions in every stage of this thesis really helped me in materializing this project. It is indeed difficult to put his contribution in few words.

I would also like to take this opportunity to present my most sincere regards to **Dr. Goldie Gabrani**, Assistant Professor, Department of Computer Engineering, for her able guidance and support.

I would also like to take this opportunity to present my sincere regards to my teachers viz. Professor D. Roy Choudhury, Dr S. K. Saxena, Mr. Rajeev Kumar and Mrs. Rajni Jindal for their support and encouragement.

I am thankful to my friends and classmates for their unconditional support and motivation during this project.

Ashish Bhawsar

M.E. (Computer Technology & Applications)
College Roll No. 05/CTA/03
Delhi University Roll No. 3005

ABSTRACT

In this dissertation, design and implementation of an enhanced MINIX file system MyFS for MINIX operating system has been proposed and implemented. This design is for MINIX operating system and MyFS is modified version of MINIX file system with some new features, like append-only, immutable files/directories and secure deletion.

MINIX file system has some limitations. MyFS, is an effort to enhance the existing MINIX file system, and also include some new features. In MINIX file system maximum file size is restricted to nearly 65 MB. MyFS lift this restriction and is capable to handle the files of large size. The maximum file size limit is extended to 4GB nearly. MyFS uses the *triple indirect blocks* to implement support for large files, which have not been used in existing MINIX file system.

Two new types of file and directory have been added; *append-only* and *immutable* files/directories. These new type of files have are being provided in many modern operating systems, and have been proved very useful to store sensitive system and configuration information.

Another new feature of *secure file deletion* has been added. File systems just delete the references of the file and leave the file data undeleted, which can be recovered by recovery software. Moreover some special techniques can recover the data after overwriting. [5] To securely delete the user data from the disk some specific data patterns are to be write on the disk data blocks.[5] Secure deletion feature tries to delete user file data by overwriting such patterns on the disk. This dissertation is focusing on the design and implementation of these modifications. This design does not propose a new file system from the scratch but suggest modification for enhancement in the existing MINIX file system.

Contents

CHAPTER 1 PROBLEM DEFINITION AND INTRODUCTION

1.1 Literature survey

Before having problem introduction, first we summarize a few basic terms and ideas related to operating systems in this section.

1.1.1 What is an Operating System?

An **operating system** (**OS**) is the system software responsible for the direct control and management of hardware and basic system operations. Additionally, it provides a foundation upon which to run application software.

The operating system ensures that other applications are able to use memory, input and output devices and have access to the file system. If multiple applications are running, the operating system schedules these such that all processes have sufficient processor time where possible and do not interfere with each other.

In general, the operating system is the first layer of software loaded into computer memory when it starts up. As the first software layer, all other software that gets loaded after it depends on this software to provide them with various common core services. These common core services include *disk access*, *memory management, task scheduling,* and *user interfacing*. Since these basic common services are assumed to be provided by the OS.

Some examples of operating systems: UNIX like OS's (which consists of all UNIX from BSD (FreeBSD) and Linux, Mac OS, Microsoft Windows, Solaris, MS-DOS, CP/M and AmigaOS.

An operating system is conceptually broken into three sets of components: a *user interface* (which may consist of a GUI and/or a *command line interpreter* or "shell"), *low-level* system

utilities, and a *kernel*--which is the heart of the operating system. As the name implies, the *shell* is an outer wrapper to the kernel, which in turn talks directly to the hardware.

The portion of code that performs these core services is called the "***kernel***" of the operating system.

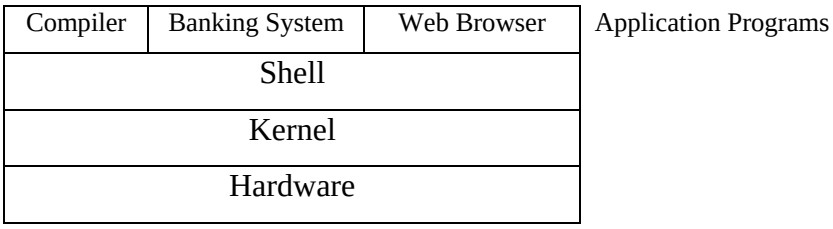

Figure 1.1 Components of Operating System

The **kernel** is the core of an operating system. It is a piece of software responsible for providing secure access to the machine's hardware and to various computer processes (a process is a computer program in a state of execution). Since there are many programs, and hardware access is limited, the kernel also decides when and how long a program should be able to make use of a piece of hardware, which is called **scheduling**. Kernels usually implement some hardware abstraction to hide the underlying complexity from the operating system and provide a clean and uniform interface to the hardware, which helps application programmers to develop programs that work with all devices of that type. The Hardware Abstraction Layer (**HAL**) then relies upon a software driver that provides the instructions specific to that device's manufacturing specifications.

1.1.2 Types of operating systems

Kernel design ideologies include those of the *monolithic kernel*, *microkernel*, and *exokernel*. Traditional commercial systems such as UNIX and Windows (including Windows NT), Linux and Mac OS X use a monolithic approach, while the trend in more modern systems are to use a microkernel (such as in AmigaOS, QNX, BeOS, etc). The microkernel approach is also very popular among research operating systems. Many embedded systems use ad hoc exokernels

On the basis of the kernel types, operating systems can be divided into four broad categories:

1. Monolithic operating system
2. Microkernel based operating system
3. Hybrid (modified microkernels) based operating system
4. Exokernel based operating system

Brief description of each of these operating systems is given below.

The *monolithic* approach defines a high-level virtual interface over the hardware, with a set of primitives or *system calls* to implement operating system services such as *process* management, *concurrency*, and *memory management* in several modules that run in supervisor mode. Even if every module servicing these operations is separate from the whole, the code integration is very tight and difficult to do correctly, and, since all the modules run in the same address space, a bug in one module can bring down the whole system. However, when the implementation is complete and trustworthy, the tight internal integration of components allows the low-level features of the underlying system to be effectively exploited, making a good monolithic kernel highly efficient. Monolithic kernels include traditional UNIX kernels, Linux kernel Windows NT.

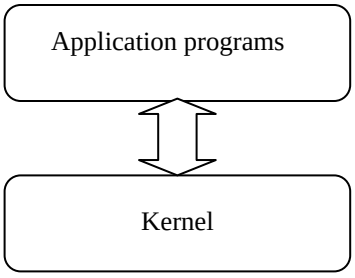

Figure 1.2 Overview of a monoltihic kernel

The *microkernel* approach consists in defining a very simple abstraction over the hardware; with a set of primitives or system calls to implement minimal OS services such as *thread management*, address spaces and interprocess communication. All other services, those normally provided by the kernel such as networking, are implemented in user-space programs referred to as *servers*. Servers are programs like any others, allowing the operating system to be modified simply by starting and stopping programs.

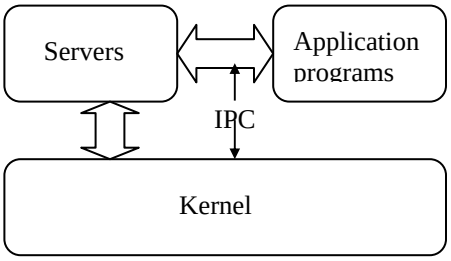

Figure 1.3 Overview of a micro kernel

Microkernels generally under perform traditional designs, sometimes dramatically. This is due in large part to the overhead of moving in and out of the kernel, a context switch, in order to move data between the various applications and servers. In more recent times newer microkernels, designed for performance first, have addressed these problems to a very large degree []. Examples of microkernels and OSs based on microkernels: AIX, Amoeba, Chorus microkernel, EROS, K42, KeyKOS (a nanokernel), The L4 microkernel family, Mach, used in GNU Hurd, NEXTSTEP, and OPENSTEP, MERT, MINIX, MorphOS, QNX, RadiOS, Windows XP, Symbian OS.

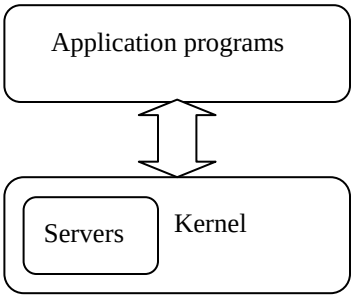

Figure 1.4 Overview of a hybrid kernel

"*Hybrid*" implies that the kernel in question shares architectural concepts or mechanisms with both monolithic and microkernel designs - specifically message passing and migration of "non-essential" code into userspace while retaining some "non-essential" code in the kernel proper for performance reasons. *Hybrid kernels* are essentially microkernels that have some "non-essential" code in kernel-space in order for that code to run more quickly than it would were it to be in user-space. Most modern operating systems today fall into this category, Microsoft

Windows NT and successors being the most popular examples. Other Hybrid kernels are ReactOS, BeOS kernel, and Netware kernel.

Exokernels, also known as *vertically structured* operating systems, are a new and rather radical approach to OS design. The idea behind exokernels is to force as few abstractions as possible on developers, enabling them to make as many decisions as possible about hardware abstractions. Exokernels are tiny, since functionality is limited to protection and multiplexing of resources. Exokernels enable low-level access to hardware; applications and abstractions may request a specific memory addresses, disk blocks etc. The kernel only ensures that the requested resource is free, and the application is allowed to access it. This low-level hardware access allows the programmer to implement custom abstractions, and omit unnecessary ones, most commonly to improve a program's performance. Exokernels are still a research effort and it was not used in any major commercial operating systems. A concept operating exokernel system is Nemesis, Citrix Systems.

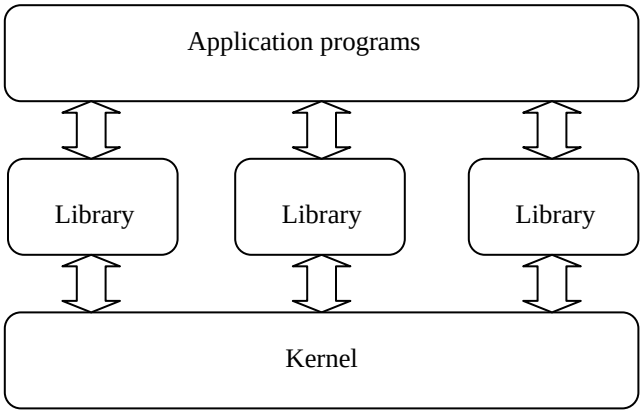

Figure 1.5 Overview of a Exokernel kernel

1.1.3 File System: Overview

A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, or they may be virtual and exist only as an access method for virtual data or for data over a network (e.g. NFS).

The most familiar file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called sectors, generally 512 bytes each. The file system software is responsible for organizing these sectors into files and directories, and keeping track of which sectors belong to which file and which are not being used. File systems typically have *directories*, which associate file names with *files*, usually by connecting the file name to an index into a *file allocation table*, such as the FAT in an MS-DOS file system, or an i-node in an UNIX-like file system. Directory structures may be *flat*, or allow *hierarchies* where directories may contain subdirectories.

In some file systems, file names are structured, with special syntax for filename extensions and version numbers. In others, file names are simple strings, and per-file metadata is stored elsewhere.

1.1.6 File Systems in Unix like OS

Most operating systems provide a file system, as a file system is an integral part of any modern operating system. Early microcomputer operating systems' only real task was *file management* - a fact reflected in their names. Some early operating systems had a separate component for handling file systems, which was called a *disk operating system*.

Unix and Unix-like operating systems assign a device name to each device, but this is not how the files on that device are accessed. Instead, Unix creates a virtual file system, which makes all the files on all the devices appear to exist under the one hierarchy. This means, in Unix, there is one root directory, and every file existing on the system is located under it somewhere. Furthermore, the Unix root directory does not have to be in any physical place. It might not be on our first hard drive - it might not even be on your computer. Unix can use a network-shared resource as its root directory. To gain access to files on another device, you must first inform the operating system where in the directory tree you would like those files to appear. This process is called mounting a file system.

1.2 Problem Description

For devlopment in file system for an operating oystem, the first, and the most obvious step is selection of a suitable operating oystem to achieve our target. Owing to time consideration, and differences in file systems of various type of operating systems, a generalized file system which could work for all the different architecture is beyond comprehension.

Also owing to various licensing restriction and factors related to cost effectiveness, it is necessary to select a generalized hardware platform with an open source operating system for this project.

There are numerous open source operating system available, which could be modified freely without much restrictions. It includes Linux which is widely available and is considered an industry standard today. linux uses VFS allows Linux to support many, often very different, file systems, each presenting a common software interface to the VFS. Linux uses ext2 and ext3 as its native file systems, which are very advanced. The **extended file system** designed to overcome certain limitations of the MINIX file system. It will be more like reinventing the wheel. A similar logic applies for most other open source operating system.

Also, present day Linux is a modular kernel based operating system designed with help and contributions of hundreds of volunteers over a time of more then a decade. To make any change to its kernel would need to recompile the whole operating system each time -which is a much massive process in comparison to working with a micro kernel based operating system. Also, as Linux is a widely used operating system, it contains a lot of modules which are not exactly needed for a project designed for research or study purpouses. This all contributes to a massive code size, managing and modifying which is a much more time taking and error prone process with no real advantage, if aims of this project is considered.

The only advantage of Line is of being monolithic, wchihc are considered more efficietn with compar to microkernels. But recent researchs have proved that the microkernl Oss can be as

efficient as monolithics. Some research papers, one aauthored by **Andrew S. Tanenbaum,** Fred Douglis, Frans Kaashoek and John Ousterhout in the Dec. 1991 gives actual performance measurements and supports Rick Rashid's conclusion that microkernel based systems are just as efficient as monolithic kernels. In words of **Tanenbaum -** *the microkernel vs. monolithic debate is essentially over. Microkernels have won. The only real argument for monolithic systems was performance, and there is now enough evidence showing that microkernel systems can be just as fast as monolithic systems.*

Also, MINIX is a micro kernel based operating system and in comparison to monolithic kernel based or modular kernel based operaing system, it is much easier to make and track changes to a MINIX kernel.

Considering these factors, it would be ideal to choose MINIX, an open source operating system by Andrew S. Tanenbaum, designed to teach students about the fundamentals of operating systems. The Minix file system was an efficient and relatively bug-free piece of software, but quite limited in features and restricted in capabilities.

There is need of a modified version of the MINIX file system that lifts various limits, such as file size and adds new features.

This is an effort to design and implement an enhanced file system MyFS for MINIX operating system that have new features and avoid maximum file size restriction. We have introduced two new types of files- immutable files and append-only files, which have been used in new operating systems and very useful storing system information, source configuration systems and other purposes.

We have included a new feature- secure deletion of files. This is a useful feature for the security point of view important user data.

We designed and implemented the enahcend file system such that it is backward compaitible. It can work witohut any problem on MyFS file systems and older version of the MINIX Os (MINIX V1 and MINIX V2 file systems.)

1.3 Dissertation organization

The organization of this dissertation is as follows.

Chapter 2: Provides introduction to file system and its concepts in details.

Chapter 3: Explains the existing MINIX file system (V2) and its structure in detail.

Chapter 4: First discussed some concepts related to enhancement to new file system and new features that are to be implemented. After that detailed designed of enhanced file system MyFS, has been discussed.

Chapter 5: Explains the implementation details of the modified and enhanced MINIX file system - MyFS. Implementation is discussed from the point of view of enhancement done and various features that have been implemented. Brief discussions of modifications for new features have been given.

Chapter 6: Presents the conclusion over the various aspects of the proposed design and discusses the further enhancements as future work.

References

Source code

CHAPTER 2 FILE SYSTEM CONCEPTS

2.1 INTRODUCTION

A ***file system*** is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk. The word is also used to refer to a partition or disk that is used to store the files or the type of the filesystem.

More formally, ***a file system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data***. []

The file system is the most visible aspects of an operating system. It provides the mechanism for an on-line storage of and access to both data and programs of the operating system and all the users of the computer program. The file system consists of two distinct parts: a collection of *files*, each storing related data, and a *directory structure*, which organizes and provides information about all the files in the system. Some file systems have a third part, *partitions*, which are used to separate physically or logically large file collections of directories.[]

The difference between a disk or partition and the filesystem it contains is important. A few programs operate directly on the raw sectors of a disk or partition; if there is an existing file system there it will be destroyed or seriously corrupted. Most programs operate on a filesystem, and therefore won't work on a partition that doesn't contain one.

Before a partition or disk can be used as a filesystem, it needs to be initialized, and the bookkeeping data structures need to be written to the disk. This process is called *making a filesystem*.

The most familiar file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called sectors, generally 512 bytes each. The file

system software is responsible for organizing these sectors into files and directories, and keeping track of which sectors belong to which file and which are not being used.

2.1.1 Various File Systems

File system types can be classified into:

1. Disk file systems,
2. Network file systems,
3. Database-based file systems and
4. Special purpose file systems.

Brief discussion of them is given below:

A *disk file system* is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer. Examples of disk file systems include FAT, NTFS, ext2, ISO 9660, ODS-5, and UDF. Some disk file systems are also journaling file systems or versioning file systems.

A *network file system* (also known as a distributed file system) is a file system where the files are accessed over a network, potentially simultaneously by several computers. Ideally, access to network file systems is user transparent. Examples include NFS, CIFS, Lustre, and Global File System.

New concepts for file management are *database-based file systems*. Instead of hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar metadata. Therefore a file search can be formulated in SQL or in natural speech. The example on the right side shows a query for "Movies that were directed by Spielberg". Examples include BFS, GNOME Storage, and WinFS.

A *special purpose* file system is basically any file system that is not a disk file system or network file system. This includes systems where the files are arranged dynamically by software, intended for such purposes as communication between computer processes or temporary file space. Special purpose file systems are most commonly used by file-centric operating systems such as Unix. Examples include the '/proc' file system used by some Unix variants, which grants access to information about processes and other operating system features.

Most modern space exploration craft like Cassini-Huygens used RTOS file systems or RTOS influenced file systems. The mars rovers are one such example of RTOS files systems, important in this case because they are implemented in flash memory.

There are some popular file systems like MINIX file system**,** ext, ext2, ext3 and FAT (MS-DOS).

MINIX file system is the oldest, presumed to be the most reliable, but quite limited in features (maximum file name 30 characters) and restricted in capabilities (at most 64 MB per file system).
A modified version of the minix filesystem that lifts the limits on the filenames and filesystem sizes but does not otherwise introduces new features.

Ext is the older version of ext2 that wasn’t upwards compatible. It is hardly ever used in new installations any more and most people have converted to ext2.

Ext2 is the most powerful of the native Linux filesystems, currently also the most popular one. It is designed to be easily upward compatible, so those new versions of the filesystem code do not require re-making the existing filesystems.

FAT (MS DOS file system) is compatibility with MS-DOS (and OS/2 and Windows NT) FAT filesystems.

2.2 File Concepts

A **file** is a stream (sequence) of bits stored as a single unit, typically in a file system on disk or magnetic tape.

Files are an ***abstraction mechanism.*** *The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file [ ].* They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work.

While a file is usually presented as a single stream, it most often is stored as multiple fragments of data at different places on a disk (or even multiple disks). One of the services operating systems usually perform for applications is that of organization of files in a file system.

File properties and other concepts are discussed in details.

2.2.1 File Naming

Files are created by processes and usually conform to a particular file format. They are almost always assigned file names by the file system, on which they are stored, so that they can be referred to at a later time.

The exact rules for file naming vary somewhat from system to system. Frequently digits and special characters are also permitted, so names like 2, urgent are valid as well. Many file systems support names as long as 255 characters. Some file systems distinguish between upper case letters and lower case letters, whereas others do not. UNIX falls in the first category; MSDOS falls in the second.

Many operating systems support two-part file names, with the two parts separated by a period, as in *prog.c*. The part following the period is called the *file extension* and usually indicates

something about the file. In MSDOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in prog.c.Z. Conventions like this are: especially useful when the same program can handle several different kinds of files.

2.2.2 File Structure

Files can be structured in any of several ways. Three common possibilities are there. The file in UNIX is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. User level programs must impose any meaning. Both UNIX and MSDOS use this approach.
Having the operating system regard files as nothing more than byte sequences provides the maximum flexibility. User programs can put anything they want in files and name them any way that is convenient.

In the second approach a file is a sequence of fixed length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation re turns one record and the write operations overwrites or appends one record. In the third kind a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key. The basic operation here is not to get the "next" record, although that is also possible, but to get the record with a specific key.

Many operating systems support several types of files. Unix and MSDOS, for example, have *regular files* and *directories*. UNIX also has *character* and *block special files*. Regular files are the ones that contain user information. Directories are system files for maintaining the structure of the file system. Character special files are related to input/output and used to model serial I/0 devices such as terminals, printers, and networks. Block special files are used to model disks. Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text. In some systems each line is terminated by a carriage return character. In others, the line feed character is used. Occasionally, both are required. Lines need not all be of the same length.

The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with an ordinary text editor. Other files are *binary files*, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of what is apparently random junk. Usually, they have some internal structure. Although technically the file is just a sequence of bytes, the operating system will only execute a file if it has the proper format. All operating systems must recognize one file type, their own executable file, but some recognize more.

2.2.3 File Access

Early operating systems provided only one kind of file access: sequential access. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files can be rewound, however, so they can be read as often as needed. Sequential files are convenient when the storage medium is magnetic tape, rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key, rather than by position. Files whose bytes or records can be read in any order are called random access files. Two methods are used for specifying where to start reading. In the first one, every READ operation gives the position in the file to start reading at. In the second one, a special operation, SEEK, is provided to set the current position. After a SEEK, the file can be read sequentially from the now current position.

2.2.4 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was created and the file's size. We will call these extra items the file's attributes. The list of attributes varies considerably from system to system.

The first four attributes relate to the file's protection and tell who may access it and who may not. All kinds of schemes are possible; in some systems the user must present a password to access a file, in which case the password must be one of the attributes.

The flags are bits or short fields control or enable some specific property. Hidden files, for example, do not appear in listings of all the files. The archive flag is a bit that keeps track of whether the file has been backed up. The backup program clears it, and the operating system sets it whenever a file is changed. In this way, the backup program can tell which files need backing up. The temporary flag allows a file to be marked for automatic deletion when the process that created it terminates.

The record length, key position, and key length fields are only present in files whose records can be looked up using a key. They provide the information required finding the keys. The various times keep track of when the file was created, most recently accessed and most recently modified. These are useful for a variety of purposes. For example, a source file that has been modified after the creation of the corresponding object file needs to be recompiled. These fields provide the necessary information.

The current size tells how big the file is at present. Some mainframe operating systems require the maximum size to be specified when the file is created, to let the operating system reserve the maximum amount of storage in advance. Workstation and personal computer operating systems are clever enough to do without this feature.

2.2.5 File Operations

Traditional file systems offer facilities to create, move and delete both files and directories. They lack facilities to create additional links to a directory, rename parent links, and create bi-directional links to files. Traditional file systems also offer facilities to truncate, append to, create, move, delete and in-place modify files.

Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.

1. CREATE: The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.

2. DELETE: When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.

3. OPEN: Before using a file, a process must open it. The purpose of the OPEN call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls,

4. CLOSE: When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing file forces writing of the file's last block, even though that block may not be entirely full yet.

5. READ: Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.

6. WRITE: Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

7. APPEND: This call is a restricted form of WRITE. It can only add data to the end of the file. Systems that provide a minimal set of sys tem calls do not generally have APPEND, but many systems provide multiple ways of doing the same thing, and these systems sometimes have APPEND.

8. SEEK: For random access files, a method is needed to specify from where to take the data. One common approach is a system call, SEEK, that repositions the pointer to the current position to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

9. GET ATTRIBUTES: Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. SET ATTRIBUTES: Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.

11. RENAME: It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

2.3 Implementation of Files

While a file is usually presented as a single stream, it most often is stored as multiple fragments of data at different places on a disk. Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. There are some approaches, which is discusses below.

2.3.1 Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous block of data on the disk. Thus on a disk with 1K blocks, a 50K file would be allocated 50 consecutive blocks. This

scheme has two significant advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced to remembering one number, the disk address of the first block. Second, the performance is excellent because the entire file can be read from the disk in a single operation.

Unfortunately, contiguous allocation also has two equally significant drawbacks. First, it is not feasible unless the maximum file size is known at the time the file is created. Without this information, the operating system does not know how much disk space to reserve. The second disadvantage is the fragmentation of the disk that results from this allocation policy. Space is wasted that might otherwise have been used.

2.3.2 Linked List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

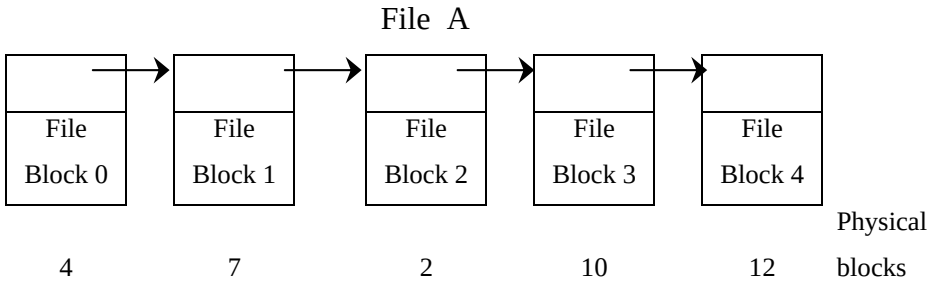

Figure 2.1 Storing a file as a linked list of disk blocks

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there. On the other hand, although reading a file sequentially is straightforward, random access is extremely slow.

2.3.3 Linked List Allocation Using an Index

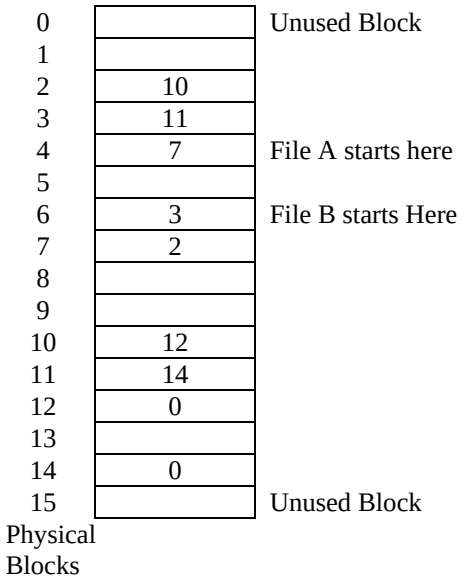

Figure 2.2 Linked List allocation using a table in memory

Taking the pointer word from each disk block and putting it in a table or index in memory can eliminate both disadvantages of the linked list allocation. Figure shows what the table looks like for the example of Fig. In both figures, we have two files. File A uses disk blocks 4, 7, 2, 10, and 12, in that order, and file B uses disk blocks 6,3, 11, and 14, in that order. Using the table of Fig., we can start with block 4 and follow the chain d l the way to the end. The same can be done starting with block 6.

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is. MSDOS uses this method for disk allocation.

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work.

2.3.4 I-NODES

Our last method for keeping track of which blocks belong to which file is to associate with each file a little table called an i-node (index node), which lists the attributes and disk addresses of the file's blocks, as shown in Figure 2.3.

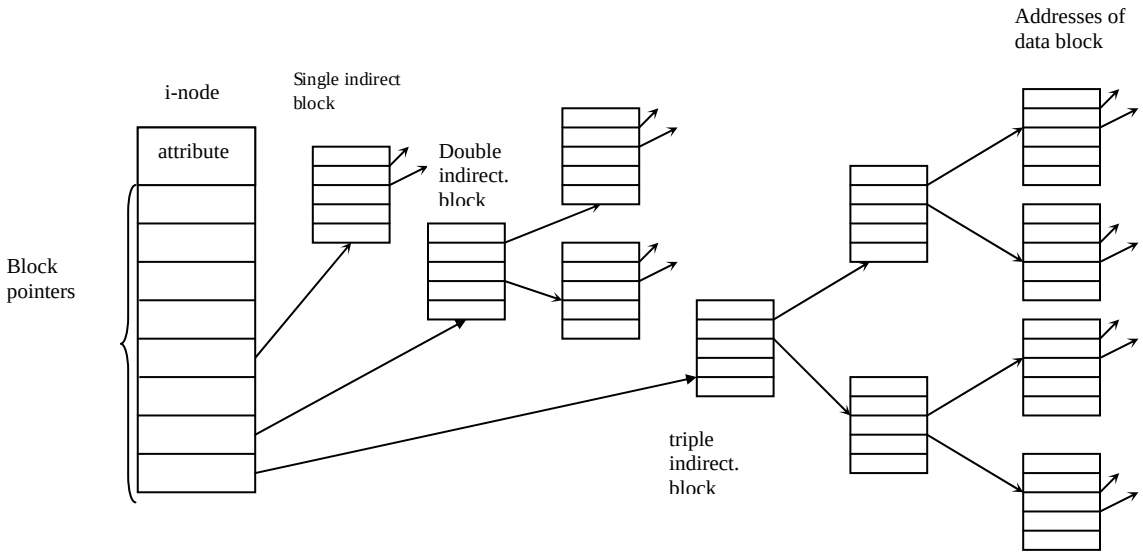

Figure 2.3 I-node structure for file

The first few disk addresses are stored in the i-node itself, so for small files, all the necessary information is right in the i-node, which is fetched from disk to main memory when the file is opened. For somewhat larger files, one of the ad dresses in the i-node is the address of a disk block called a single indirect block. This block contains additional disk addresses. If this still is not enough, another address in the i-node, called a double indirect block, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a triple indirect block can also be used.

2.4 Directories

When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry. The directory entry provides the information needed to find the disk blocks.

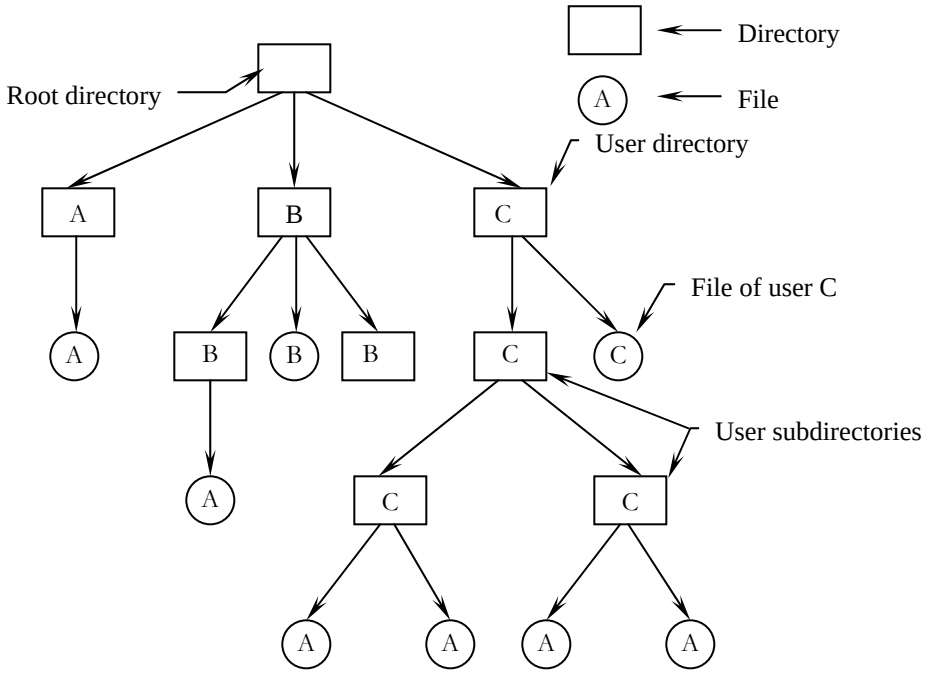

Figure 2-4 Tree structured directory

Depending on the system, this information may be the disk address of the entire file (contiguous allocation), the number of the first block (both linked list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

The number of directories varies from system to system. The simplest design is for the system to maintain a *single directory* containing all the files of all the users. If there are many users, and they choose the same file names (e.g., mail and games), conflicts and confusion will quickly make the system unworkable. This system model was used by the first microcomputer operating systems but is rarely seen any more.

An improvement on the idea of having a single directory for all files in the entire system is to have one directory per user. This design eliminates name conflicts among users but is not satisfactory far users with a large number of files. It is quite common for users to want to group their files together in logical ways. Some way is needed to group these files together in flexible ways chosen by the user.

What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so those files can be grouped together in natural ways. Files are often organized *hierarchically* by the operating system, placing them in directories.

2.4.1 Directories in UNIX

The directory structure traditionally used in UNIX is extremely simple, as shown in Fig. Each entry contains just a file name and its i-node number. All the information about the type, sizes, times, ownership, and disk blocks is contained in the i-node. Some UNIX systems have a different layout, but in all cases, a directory entry ultimately contains only an ASCII string and an i-node number.

When a file is opened, the file system must take the file name supplied and locate its disk blocks. Let us consider how the path names /usr/ast/mbox is looked up. We will use UNIX as an example, but the algorithm is basically the same for all hierarchical directory systems. First the file system locates the root directory. In UNIX its i-node is located at a fixed place on the disk.

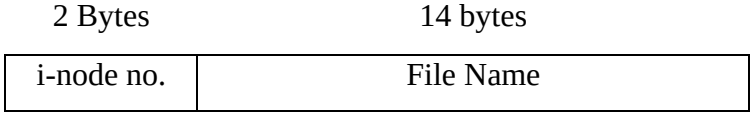

Figure 2.5 directory structure in UNIX

Then it looks up the first component of the path, usr, in the root directory to find the i-node number of the file /usr. Locating an i-node from its number is straightforward, since each one has a fixed location on the disk.

2.5 Disk Space Management

Files are normally stored on disk, so management of disk space is a major concern to file system designers. Two general strategies are possible for storing an n byte file: n consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks. The same tradeoff is present in memory management systems between pure segmentation and paging.

Storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it will probably have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed size blocks that need not be adjacent.

2.5.1 Keeping Track of Free Block

To keep track of free blocks, two methods are widely used, as shown in Figure 2.6. The first one consists of using a *linked list* of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1K block and a 32-bit disk block number, each black on the free list holds the numbers of 255 free blocks. Often free blocks are used to hold the free list.

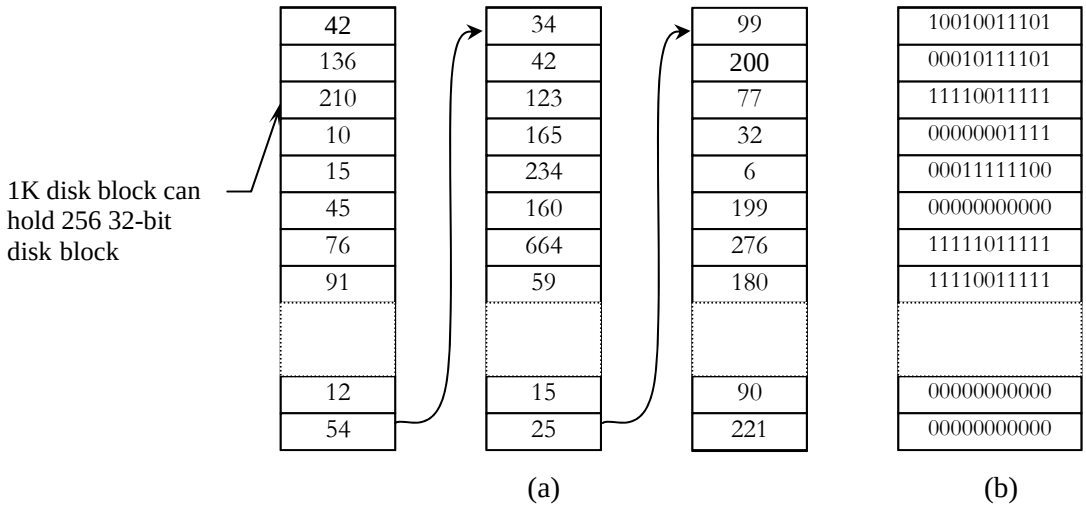

Figure 2-6 (a) Storing the free list on a linked list (b) A bit map

The other free space management technique is the *bit map*. A disk with n blocks requires a bit maps with n bits. Free blocks are represented by 0s in the map, allocated blocks by 1s (or vice versa). A 26MM disk requires 200K bits for the map, which requires only 2.5 blocks. It is not surprising that the bit map re quires less space, since it uses I bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full will the linked list scheme require fewer blocks than bit map.

If there is enough main memory to hold the bit map, that method is generally preferable. If, however, only 1 block of memory can be spared for keeping track of free disk block, and the disk is nearly full, then the linked list may be better. With only 1 block of the bit map in memory, it may turn out that no free blocks can be found on it, causing disk accesses to read the rest of the bit map. When a fresh block of the linked list is loaded into memory, 255 disk blocks can be allocated before having to go to the disk to fetch the next

2.6 Protection Mechanisms

In some systems, a program called a *reference monitor*, Every time an access to a potentially protected resource is attempted enforces protection; the system first asks the reference monitor to check its legality. The reference monitor then looks at its policy tables and makes a decision.

2.6.1 Protection Domains

A computer system contains many "objects" that need to be protected. These objects can be hardware (e.g. CPUs, memory segments, disk drives, or printers), or they can be software (e.g., processes, files, databases, or semaphores). Each object has a unique name by which it is referenced, and a finite set of operations that processes are allowed to carry out on it. READ and WRITE are operations appropriate to a file; UP and DOWN make sense on a semaphore.

In order to discuss different protection mechanisms, it is useful to introduce the concept of a domain. A domain is a set of (object, rights) pair. Each pair specifies an object and some subset

of the operations that can be performed on it. A right in this context means permission to perform one of the operations.

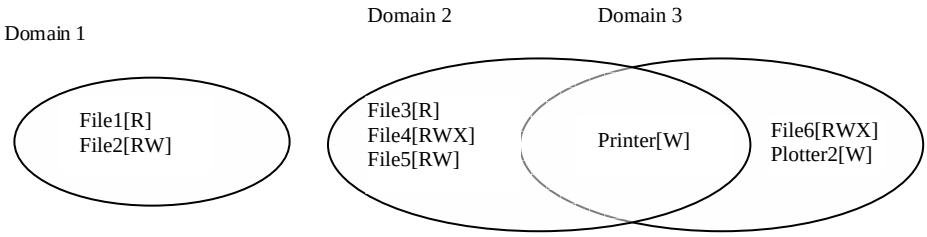

Figure 2.6 Three protection domains

Figure shows three domains, showing the objects in each domain and the rights [Read, Write, execute] available on each object. Note that Printer1 is in two domains at the same time. Although not shown in this example, it is possible for the same object to be in multiple domains, with different rights in each one.

At every instant of time, each process runs in some protection domain. In other words, there is some collection of objects it can access, and for each object it has some set of rights. Processes can also switch from domain to domain during execution. The rules for domain switching are highly system dependent.

CHAPTER 3 OVERVIEW OF THE MINIX FILE SYSTEM

Like any file system, the MINIX file system must deal with all the issues we have discussed in last chapter. It must allocate and de-allocate space for files, keep track of disk blocks and free space, provide some way to protect files against unauthorized usage, and so on.

3.1 System Calls

The file system accepts 37 system calls. MINIX file system receives system calls in form of *messages* sent by user processes and memory manager from the 37 system calls, 31 are accepted from user processes. Six system call messages are for system calls, which are handled first by the memory manager, which then calls the file system to do a part of the work. Two other messages are also processed by the file system. All the system calls are shown in table 3.1.

The structure of the file system is basically the same as that of the memory manager and all the I/O tasks. It has a main loop that waits for a message to arrive. When a message arrives, its type is extracted and used as an index into a table containing pointers to the procedures within the file system that handle all the types. Then the appropriate procedure is called, it does its work and returns a status value. The file system then sends a reply back to the caller and goes back to the top of the loop to wait for the next message.

System calls for User	Input Parameters
ACCESS	File name, access mode
CHDIR	Name of new working directory
CHMOD	File name, new mode
CHOWN	File name, new owner, group
CHROOT	Name of new root directory
CLOSE	File descriptor of file.
CREATE	Name of file to be created, mode
DUP	File descriptor (for dup2, two ids)
FCNTL	File descriptor, function code, arg
FSTAT	Name of file, buffer

IOCTL	File descriptor, function code, arg
LINK	Name of file to link to, name of link
LSEEK	File descriptor, offset, whence
MKDIR	File name, mode
MKNOD	Name of dir or special, mode, address
MOUNT	Special file, where to mount, ro flag
OPEN	Name of file to open, r/w flag
PIPE	Pointer of 2 file descriptors
READ	File descriptor, buffer, how many bytes
RENAME	File name, file name
RMDIR	File name
STAT	File name, status buffer
STIME	Pointer to current time
SYNCH	(None)
TIME	Pointer to place where current time goes
TIMES	Pointer to buffer for process and child times
UMASK	Complement of mode mask
UMOUNT	Name of special file to unmount
UNLINK	Name of file to link
UTIME	File name, file times
WRITE	File descriptor, buffer, how many bytes
System calls for MM	**Input Parameters**
EXEC	Pid
EXIT	Pid
FORK	Parent pid, child pid
SETGID	Pid, real and effective gid
SETSID	Pid
SETUID	Pid, real and effective uid

Table 3.1 List of System calls for MINIX file system

3.2 File System Structure

A MINIX file system is a logical, self-contained entity with i-nodes, directories, and data blocks. It can be stored on any block device, such as a floppy disk or a (partition of) hard disk. In all

cases, the layout of the file system has the same structure. Figure 3.1 shows this layout for a 360K floppy disk with 128 i nodes and a 1K block size. Larger file systems, or those with more or fewer i-nodes or a different block size, will have the same six components in the same order, but their relative sizes may be different.

Each file system begins with a boot block. This contains executable code. When the computer is turned on, the hardware reads the boot block from the boot device into memory, jumps to it, and begins executing its code. The boot block code begins the process of loading the operating system itself. Once the system has been booted, the boot block is not used any more. Not every disk drive can be used as a boot device, but to keep the structure uniform, every block device bas a block reserved for boot block code. At worst this strategy wastes one block. To prevent the hardware from trying to boot an unbootable device a magic number is placed at a known location in the boot block when and only when the executable code is written to the device. When booting from a device, the hardware (actually, the BIOS code) will refuse to attempt to load from a device lacking the magic number. Doing this prevents inadvertently using garbage as a boot program.

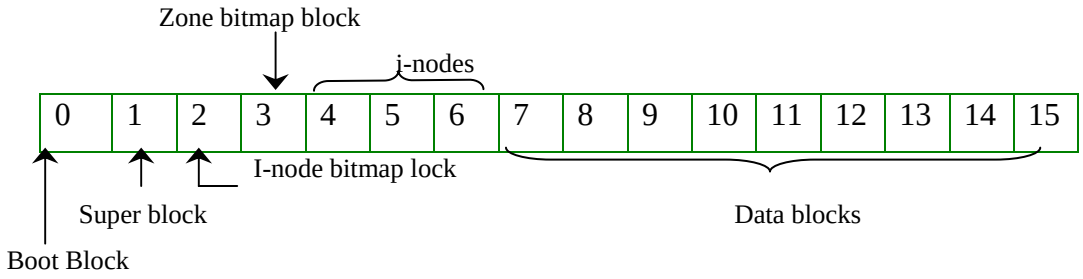

Figure 3.1 Disk layout of MINIX file system

3.2.1 Super Block

The super-block contains information describing the layout of the file system. It is illustrated in figure 3.2. The main function of the super-block is to tell the file system how big the various pieces of the file system are. Given the block size and the number of i-nodes, it is easy to calculate the size of the i-node bit map and the number of blocks of i-nodes. For example, for a

1K block, each block of the bit map has I K bytes (8K bits), and thus can keep track of the status of up to 8192 i-nodes. (Actually the first block can handle only up to 8291 i nodes, since there is no 0th i-node, but it is given a bit in the bit map, anyway). For 10,000 i-nodes, two bit map blocks are needed. Since i-nodes each occupy 64 bytes, an l K block holds up to 16 i-nodes. With 128 usable i-nodes, 8 disk blocks are needed to contain them all.

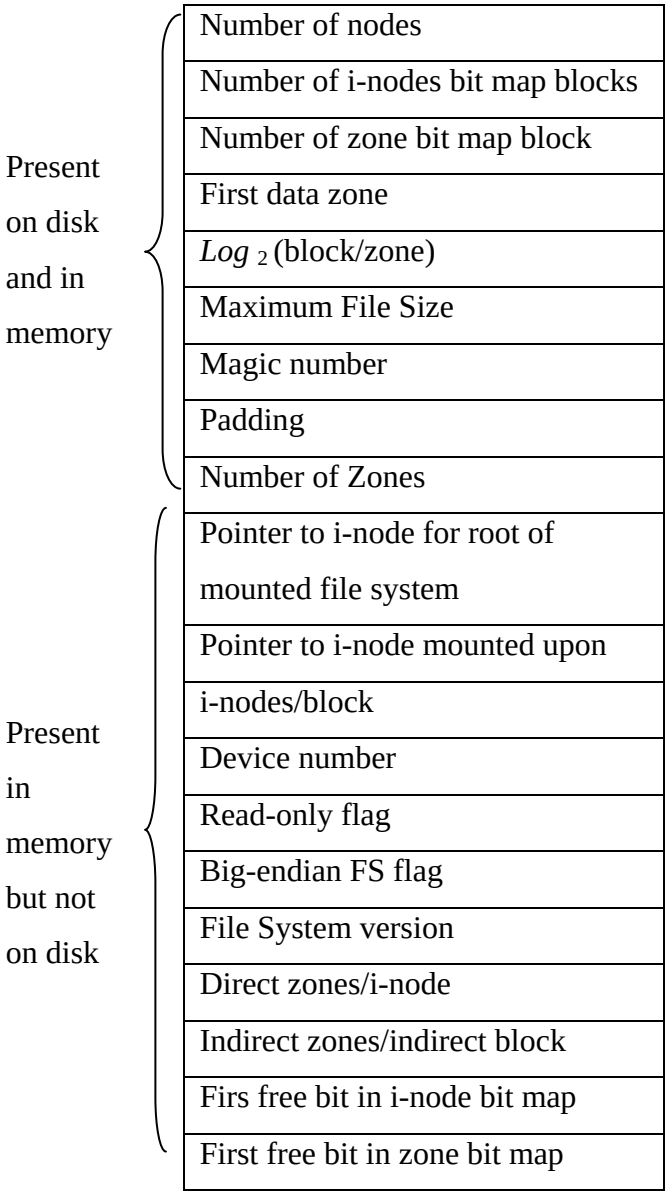

Figure 3.2 The MINIX super block structure

Disk storage can be allocated in units (zones) of 1,2,4, 8, or in general 2" blocks. The zone bit map keeps track of free storage in zones, not blocks, For all standard floppy disks used by MINIX the zone and block sizes are the same (1K), so for a first approximation a zone is the same as a block on these devices.

Note that the number of blocks per zone is not stored in the super-block, as it is never needed. And that is needed is the base 2 logarithm of the zone to block ratio, which is used as the *shift count* to convert zones to blocks and vice versa.

The *zone bit map* includes only the data zones (i.e., the blocks used for the bit maps and i-nodes are not in the map), with the first data zone designated zone I in the bit map. As with the i-node bit map, bit 0 in the map is unused, so the first block in the zone bit map can map 8191 zones and subsequent blocks can map 8192 zones each. Both the i-node and zone bit maps have 2 bits set to 1. One is for the nonexistent 0th i-node or zone; the other is for the i-node and zone used by the root directory on the device, which is placed there when the file system is created.

The information in the super-block is redundant because sometimes it is needed in one form and sometimes in another. With 1KB devoted to the super block, it makes sense to compute this information in all the forms it is needed, rather than having to re-compute it frequently during execution. The zone number of the *first data zone* on the disk, for example, can be calculated from the block size, zone size, number of i-nodes, and number of zones, but it is faster just to keep it in the super-block.

When MINIX is booted, the super-block for the root device is read into a table in memory. Similarly, as other file systems are mounted, their super-blocks are also brought into memory. The super-block table holds a number of fields not present on the disk. These include *flags* that allow a device to be specified as read-only or as following a byte-order convention opposite to the standard, and fields to speed access by indicating points in the bit maps below which all bits are marked used. In addition, there is a field describing the *device* from which the super-block came.

Before a disk can be used as a MINIX file system, shown in the structure of figure 3.2.

MINIX file system has evolved, and some aspects of the file system (for instance, the size of i-nodes) were different in earlier versions. The *magic number* identifies the version of mkfs that created the file system, so differences can be accommodated. Attempts to mount a file system not in MINIX format, such as an MSDOS diskette, will be rejected by the MOUNT sys tem call, which checks the super-block for a valid magic number and other things.

Number of Zones stores the maximum number of data blocks (zones) possible on the device. *Maximum file size* field gives the maximum file size limit in bytes on device, which is used in various file operations for error checking.

3.2.2 Bit Maps

MINIX keeps tracks of which i-nodes and zones are free by using two bit maps (in figure 2.6). When a file is removed, it is then a simple matter to calculate which block of the bit map contains the bit for the i-node being freed and to find it using the normal cache mechanism. Once the block is found, the bit corresponding to the freed i-node is set to 0. Zones are released from the zone bit map in the same way.

When a file is to be created, the file system must search through the bitmap blocks one at a time for the first free i-node. This i-node is then allocated for the new file. The in-memory copy of the super-block has a field which points to the first free i-node, so no search is necessary until after a node is used. If every i-node slot on the disk is full, the search routine returns a 0. Everything that has been said here about the i-node bit maps also applies to the zone bit map. It is searched for the first free zone when space is needed, but a pointer to the first free zone is maintained to eliminate most of the need for sequential searches through the bit map.

Most of the file system works with blocks. Disk transfers are always a block at a time, and the **buffer cache** also works with individual blocks. Only a few parts of the system that keep track of physical disk addresses (e.g., the zone bit map and the i-nodes) know about zones.

3.2.3 I-nodes

Each file is represented by a structure, called an i-node. Each i-node contains the **description** of the **file**: file *type*, *access rights*, *owners*, *timestamps*, *size*, *and pointers* to data blocks. The addresses of data blocks allocated to a file are stored in its i-node. When a user requests an I/O operation on the file, the kernel code converts the current offset to a block number, uses this number as an index in the block addresses table and reads or writes the physical block. The figure 3.3 represents the structure of an i-node.

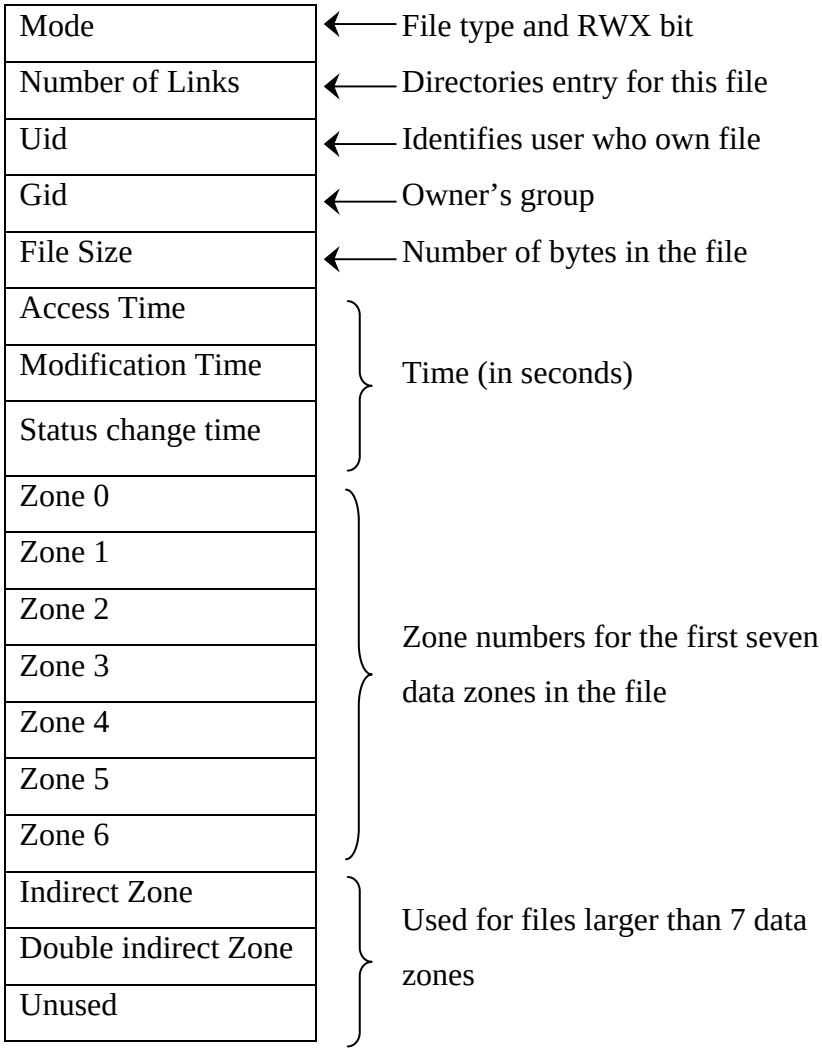

Figure 3.3 The MINIX i-node structure

It is almost the same as a standard UNIX i-node. The *disk zone pointers* are 32 bit pointers, and there are only 9 pointers, 7 direct and 2 indirect. The MINIX i-nodes occupy 64 bytes, the same as standard UNLX i-nodes, and there is space available for a 10th (triple indirect) pointer, although its use is not supported by the standard version of the FS. The MINIX i-node access, modification time and i-node change times are standard, as in UNIX. The last of these is updated for almost every file operation except a read of the file.

When a file is opened, its i-node is located and brought into the i-node table in memory, where it remains until the file is closed. The i-node table has a few additional fields not present on the disk, such as the i-node's device and number, so the file system knows where to rewrite it if it is modified while in memory. It also has a counter per i-node. If the same file is opened more than once, only one copy of the i-node is kept in memory, but the counter is incremented each time the file is opened and decremented each time the file is closed. Only when the counter finally reaches zero is the i-node removed from the table, if it has been modified since being loaded into memory, it is also rewritten to the disk.

The main function of a file's i-node is to tell where the data blocks are. The first seven zone numbers are given right in the i-node itself, with zones and blocks both l KB, files up to 7K do not need indirect blocks. Beyond 7K, *indirect zones* are needed, using the scheme of figure, except that only the *single* and *double indirect* blocks are used. With 1 K blocks and zones and 32bit zone numbers, a single indirect block holds 256 entries, representing a quarter megabyte of storage. The double indirect block points to 256 single indirect blocks, giving access to up to 64 megabytes. The maximum size of a MINIX file system is 1G.

The i-node also holds the *mode* information, which tells what *type of a file* it is (regular, directory, block special, character special, or pipe), and gives the protection and SETUID and SETGID bits.

The *link* field in the i-node records how many directory entries point to the i-node, so the file system knows when to release the file's storage.

Uid, gid field gives the owner's *user id* and its *group id*, which are used for protection mechanism. *Mode* field also stores the read, write and executes right bits for owner, owner's group and other.

3.2.4 Directories

Directories are structured in a hierarchical tree in MINIX. Each directory can contain files and subdirectories. Directories are implemented as a special type of files. Actually, a directory is a file containing a list of entries. Each entry contains an *i-node number* and a *file name*. When a process uses a pathname, the kernel code searches in the directories to find the corresponding i-node number. After the name has been converted to an i-node number, the i-node is loaded into memory and is used by subsequent requests.

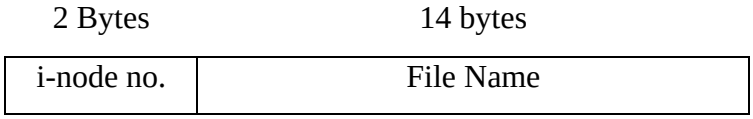

Figure 3.4 directory structure in UNIX

A MINIX directory consists of a file containing 16 byte entries as shown in figure 3.4. The first 2 bytes form a 16 bit i-node number, and the remaining 14 bytes are the file name. This is the same as the traditional UNIX.

3.2.5 File Descriptors

Once a file has been opened, a file descriptor is returned to the user process for use in subsequent READ and WRITE calls.

Like the kernel and the memory manager, the file system maintains part of the process table within its address space. Three of its fields are of particular interest. The first two are pointers to the i-nodes for the root directory and the working directory.

File mode (rw bits)
File open flags
File descriptor count
Pointer to file I-node
File offset

Figure 3.5 The MINIX file descriptor

The third interesting field in the process table is an array indexed by *file descriptor* number. This number is used to find entry in file descriptor table for that file.

File descriptor table contains the shared *file position*; it is convenient to put the *i-node pointer* there, too. File position is shared by child and parent processes. The filp entry also contains the file mode (permission bits), some flags indicating whether the file was opened in a special mode.

File descriptor Count stores the number of processes using the file descriptor entry, so the file system can tell when the last process using the entry has terminated, in order to reclaim the slot. All this information is used in subsequent file operations.

3.3 File Locking

There is yet another aspect of file system management that requires a special table. This is file locking. MINIX supports the POSIX *inter-process communication* mechanism of advisory file locking. This permits any part, or multiple parts, of a file to be marked as locked. The operating system does not enforce locking, but processes are expected to be well behaved and to look for Jocks on a file before doing anything that would conflict with another process.

The reasons for providing separate tables for locks is that a single process can have more than one lock active, and different parts of a file may be locked by more than one process (although, of course, the locks cannot overlap). So neither the process table nor the filp table is a good place to record locks. Since a file may have more than one lock placed upon it, the i-node is not a good

place either. MINIX uses another table, the *file lock table*, to record all locks. Each slot in this table has space for a lock type, indicating if the file is locked for reading or writing, the process ID holding the lock, a pointer to the i-node of the locked file, and the offsets of the first and last bytes of the locked region.

3.4 Pipes and Special Files

Pipes and special files differ from ordinary files in an important way. MINIX uses character special, block special, and terminal files. Different handling of pipes and special file are required

When a process tries to read or write from a disk file, it is certain that the operation will complete within a few hundred milliseconds at most. When reading from a pipe, the situation is different: if the pipe is empty, the reader will have to wait until some other process puts data in the pipe, which might take hours. Similarly, when reading from a terminal, a process will have to wait until somebody types something. As a consequence, the file system's normal rule of handling a request until it is finished does not work.

It is necessary to *suspend* these requests and *restart* them later. When a process tries to read or write from a pipe, the file system can check the state of the pipe immediately to see if the operation can be completed. If it can be, it is, but if it cannot be, the file system records the parameters of the system call in the process table, so it can restart the process when the time comes.

The situation with *terminals* and other *character special files* is slightly different. The i-node for each special file contains two numbers, the *major device* and the *minor device*. The major device number indicates the device class (e.g., RAM disk, floppy disk, hard disk, and terminal). It is used as an index into a file sys tem table that maps it onto the number of the corresponding task (i.e., I/O driver). In effect, the major device determines which I/O driver to call. The minor device number is passed to the driver as a parameter. It specifies which device is to be used, for example, terminal 2 or drive 1.

When a process reads from a special file, the file system extracts the major and minor device numbers from the file's i-node, and uses the major device number as an index into a file system table to map it onto the corresponding task number. Once it has the task number, the file system sends the task a message, including as parameters the minor device, the operation to be performed, the caller's process number and buffer address and the number of bytes to be transferred.

If the driver is able to carry out the work immediately (e.g., a line of input has already been typed on the terminal), it copies the data from its own internal buffers to the user and sends the file system a reply message saying that the work is done. On the other hand, if the driver is not able to carry out the work, i t records the message parameters in its internal tables, and immediately sends a reply to the file system saying that the call could not be completed. At this point, the file system is in the same situation as having discovered that someone is trying to read from an empty pipe. It records the fact that the process is suspended and waits for the next message. When the driver has acquired enough data to complete the call, it transfers them to the buffer of the still blocked user and then sends the file system a message reporting what it has done.

3.5 Mounting Operation

The usual configuration for MINIX and many other UNIX like systems is to have a small root file system containing the files needed to start the system and to do basic system maintenance, and to have the majority of the files, including user’s directories, on a separate device mounted on /usr. When the user types the command

mount /dev/hd2 /usr

on the terminal, the file system contained on hard disk partition 2 is mounted on top of /usr in the root file system. The key to the whole mounting is a *flag* set in the memory copy of the i-node of /usr after a successful mount. This flag indicates that the *i-node is mounted on*.

The MOUNT call also loads the super-block for the newly mounted file system into the super-block table and sets two pointers in it. Furthermore, it puts the root i-node of the mounted file system in the i-node table. In figure we see that super-blocks in memory contain two fields related to mounted file systems. The first of these, the i-node of the mounted file system, is set to point to the root i-node of the newly mounted file system. The second, the i-node mounted on, is set to point to the i-node mounted on, in this case, the i-node for /usr. These two pointers serve to connect the mounted file system to the root and represent the "glue" that holds the mounted file system to the root [shown as the dots in figure 3.6]. This glue is what makes mounted file systems work.

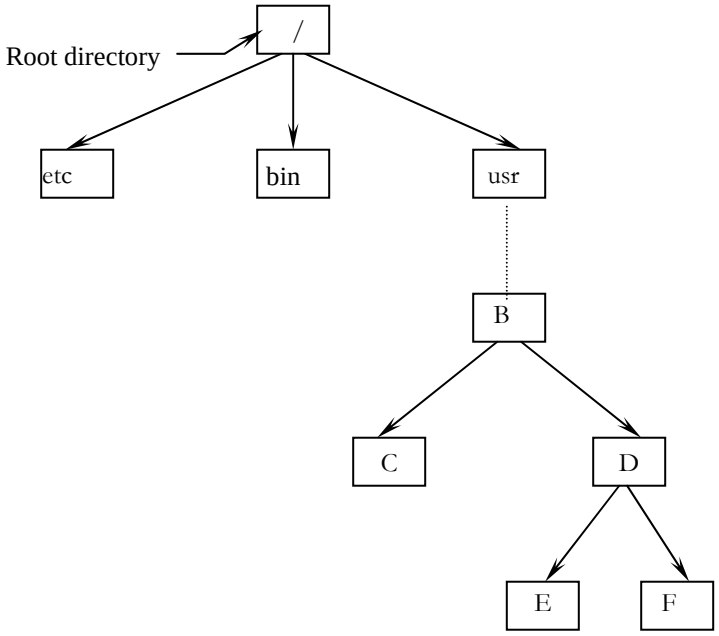

Figure 3.6 mounted file system on /usr

CHAPTER 4 MyFS: CONCEPTS AND DESIGN

To implement the enhanced MINIX file system - MyFS, some existing file system data structures have been modified. There are numerous issues related to enhanced file system and older file system, which have to be dealt. In this chapter focus on those issues and present the design for enhanced file system with new features is discussed.

4.1 Limitations of existing MINIX File System.

The MINIX file system is oldest, presumed to be the most reliable, but quite limited in features and restricted in capabilities.
The MINIX file system contains serious limitations:

1. The maximal file system size is restricted to ~65 Mega Bytes. In the MINIX V2 file System size of disk address is 32 bits, which make it's possible to deal with device sizes up to 4 terabytes in theory. But MINIX uses 9 pointers to data blocks, 7 for direct blocks and two for single indirect and double indirect block receptively. With 1-KB block size and 32-bit block numbers, a single indirect block holds 256 entries and double indirect blocks points to 256 indirect blocks, giving access to 64 MB. So maximum file size in MINIX file system is nearly 65 MB and file system size is 1 GB. So modification to use the triple indirect block or larger block sizes could both be useful to access very large files on a MINIX system.

2. Practical use of MINIX operating system results that the file system is fragile and on few system crashes, it is more likely to corrupt the file system or/and lost of data.

3. When a file is "deleted" MINIX and other file systems removes the reference to that file on the file system, but the file data still remains on the disk itself. There are many undelete programs available which can easily recover this data. This means anyone deletes his personal or important files assuming no one can access his important data, but that is not true.

4. MINIX file system uses smaller block size, 1 KB. In 1985, when MINIX was conceived, disk capacities were small and it was expected that many users would

have only floppy disks. Now a days we have very large disk space and with all aspects of computer technology, users take advantage of more abundant resources by demanding even more. So the average file size is now larger then 1 KB

4.3 Concepts

4.3.1 Large Data Blocks

Big block size can speed up I/O since fewer I/O requests, and thus fewer disk head seeks, need to be done to access a file. Big block size can speed up I/O since fewer I/O requests, and thus fewer disk head seeks, need to be done to access a file.

On the other hand smaller block size keep less internal fragmentation but reading each block normally requires a seek and a rotational delay, so reading a file consisting of many small blocks will be slow.

Now days we have very large disk space and with all aspects of computer technology, users take advantage of more abundant resources by demanding even more. So the average file size is now larger then 1 KB. *One system manager reports that the average size of files in the university system he manages has increased slowly over the years, and that in 1997 the average size of files has grown to 12K for students and 15K for faculty.*[1] So we choose large block size of 4 KB over the 1 KB block size of MINIX V2 file system.

4.3.2 Large files:

In the MINIX file system the i-node is almost the same as a standard UNIX i-node. The disk block pointers are 32-bit pointers, and there are only 9 pointers, 7 direct and 2 indirect. With 1 K blocks and blocks and 32-bit block numbers, a single indirect block holds 256 entries, representing a 256 KB of storage. The double indirect block points to 256 single indirect blocks, giving access to up to 64 megabytes. So In the MINIX file system maximum file size is restricted to 64 Mega Bytes, which imposes for the users, which need large files to store.

Increasing the number of direct blocks (12), introduce the triple indirect block and larger block size both will be useful to store very large files on a **MINIX** system. The first 12 block numbers are given right in the i-node itself. For the standard distribution, with larger blocks of **4 KB,** files up to 48KB do not need. Indirect blocks. The block size to store inodes is still have 1 KB size. With 1 **KB** blocks and 32-bit block numbers, a single indirect block holds 256 entries, representing a 256 * 4 KB (1 MB) of storage. The double indirect block points to 256 single indirect blocks, giving access up to 256 * 256 * 4 KB (256 MB) megabytes.

The use of **triple indirect** block points to 256 * 256 * 256 data blocks, giving access to up to **64 GB** (Giga Bytes) of Storage. But variable to store the file position in bytes is 32 bits long and thus limits the maximum file size up to **4 GB**. So in the MyFS maximum file system size will be nearly ~**4GB**. Table 4.1 shows the various combinations of file size and data block size, when indirect block have 1 KB size, which can hold 256 data block pointers.

Block Size	**Direct Block Pointers (12)**	**1 indirect pointers**	**2 indirect pointers**	**3 indirect pointers**
1024 Bytes	12 KB	268 KB	64 MB	16.06 GB
2048 Bytes	24 KB	513 KB	128 MB	~32 GB
4096 Bytes	48 KB	1 MB	256 MB	~65 GB

Table 4.1 File size upper limits for data block addressing.

4.3.3 Append only Files/Directory

This is a new type of file and directory, which cannot be modified, only new data, can be appended in these types of files or directories. Append-only files can be opened in write mode but data is always appended at the end of the file. This is attribute is especially useful for some system files like *log files*, which can only grow.
Some advantages and possible uses of append-only files are:

(i) The Vesta repository is a special-purpose *replicated file system*, developed as part of the Vesta software configuration management system. One of the major goals of Vesta is to make all software builds reproducible. The append-only nature of the *repository* greatly simplifies the problem of maintaining consistency among replicas.[]

(ii) In *Google file system*, files are append-only; a *stable replication* usually returns a premature end of chunk rather than outdated data. When a reader retries and contacts the master, it will immediately get current chunk locations.

(iii) To support reproducible builds, files can be designated as mutable or immutable, and directories to be designated as mutable, appendable, or immutable. In an appendable directory, new names can be created, but existing names cannot be deleted and their meanings cannot be changed. Vesta's builder reads source code and build instructions from a directory tree consisting only of appendable and immutable objects, so the meanings of names cannot change from one build to another.[SRC]

(iv) To support distributed software development, sources stored in the depository's appendable tree can be replicated.[SRC]

4.3.4 Immutable File/Directory

New types of files inspired from the *BSD file system* have been added. Immutable files can only be read: nobody can write or modify its content. This can be used to protect sensitive system and configuration files.
Many modern file systems like ext2, ext3, BSD, AMOEBA now supports the immutable files. Immutable files have many advantages, like:

(i) This can be used to enforce *access control*: the file system can take measures to protect against some root initiated attacks by declaring certain parts of a file system as read-only or immutable, or by requiring authentication before certain executables are modified.

(ii) Having files be Immutable in distributed operating systems makes it much easier to support *file caching* and *replication* because it eliminates all the problems associated with having to update all copies of a file whenever changes. The *Cedar file system* uses the immutable files and was designed to realize the following goals: high-performance remote file access by workstations [ ]

(iii) File sharing in a distributed system poses many problems. One approach to the semantic of file sharing is to make all files immutable. The **bullet server** of AMOEBA distributed operating system was designed to be very fast. AMOEBA uses the immutable file, which simplifies *automatic replication* and it also well suited for use on large-capacity, write-once optical disks.

4.3.5 Secure Deletion

Usually all the file systems do not delete the files physically from the disk. They just delete the *directory entry* of file name and mark the data blocks and i-node free. The file data remains on the disk and some any data recovery tools can recover that data after some time.

With the use of increasingly sophisticated encryption systems, an attacker wishing to gain access to sensitive data is forced to look elsewhere for information. One avenue of attack is the recovery of supposedly erased data from magnetic media or random-access memory.

Magnetic force microscopy (MFM) is a recent technique for imaging magnetization patterns with high resolution and minimal sample preparation. In the 1980's some work was done on the recovery of erased data from magnetic media [1] [2] [3], but to date the main source of information is government standards covering the destruction of data.

While methods for securely deleting data from magnetic storage exist, all fail to meet the combined requirements put forth by legislators. *Secure overwriting* [4] is a method by which

data blocks are overwritten many times with alternating patterns of 1s and 0s in order to degauss the magnetic media, making the data safe from magnetic force microscopy.[ ]

4.3.6 Erasure of Data stored on Magnetic Media *[5]*

The general concept behind an overwriting scheme is to flip each magnetic domain on the disk back and forth as much as possible without writing the same pattern twice in a row. If the data was encoded directly, we could simply choose the desired overwrite pattern of ones and zeroes and write it repeatedly.

The method proposed by Peter Gutmann of *Department of Computer Science, University of Auckland,* ensure that transitions aren't placed too closely together, or too far apart, which would mean the drive would lose track of where it was in the data.

The best we can do is to use the lowest frequency possible for overwrites, to penetrate as deeply as possible into the recording medium because very high frequency signals only "scratch the surface" of the magnetic medium.

In order to understand the theory behind the choice of data patterns to write it is necessary to take a brief look at the recording methods used in disk drives. The main limit on recording density is that as the bit density is increased, the peaks in the analog signal recorded on the media are read at a rate which may cause them to appear to overlap, creating intersymbol interference which leads to data errors. To reduce the possibility of intersymbol interference by coding data in such a way that the analog signal peaks are separated as far as possible. The separation of peaks is implemented as some form of run-length-limited, or RLL, coding.

The **RLL** encoding used in most current drives is described by pairs of run-length limits (*d*, *k*), where *d* is the minimum number of 0 symbols which must occur between each 1 symbol in the encoded data, and *k* is the maximum.

The grandfather of all RLL codes was **FM**, which wrote one user data bit followed by one clock bit, so that a 1 bit was encoded as two transitions while a 0 bit was encoded as one transition. A different approach was taken in modified FM (**MFM**), which suppresses the clock bit except between adjacent 0's (the ambiguity in the use of the term MFM is unfortunate.

These constraints help avoid intersymbol interference, but the need to separate the peaks reduces the recording density and therefore the amount of data which can be stored on a disk. To increase the recording density, MFM was gradually replaced by (2,7) RLL (the original "RLL" format), and that in turn by *(1,7) RLL*, each of which placed less constraint on the recorded signal.

The three encoding methods described above cover the vast majority of magnetic disk drives. However, each of these has several possible variants. With MFM, only one is used with any frequency, but the newest **(1,7) RLL** code has at least half a dozen variants in use. Although there can be as many as 8 bit times between transitions, the lowest sustained frequency we can have in practice is 6 bit times between transitions. This is a desirable property from the point of view of the clock-recovery circuitry, and all (1,7) RLL codes seem to have this property. We now need to find a way to write the desired pattern without knowing the particular (1,7) RLL code used. We can do this by looking at the way the drives error-correction system works. The error- correction is applied to the decoded data, even though errors generally occur in the encoded data. In order to make this work well, the data encoding should have limited error amplification, so that an erroneous encoded bit should affect only a small, finite number of decoded bits.

Decoded bits therefore depend only on nearby encoded bits, so that a repeating pattern of encoded bits will correspond to a repeating pattern of decoded bits. The repeating pattern of encoded bits is 6 bits long. Since the rate of the code is 2/3, this corresponds to a repeating pattern of 4 decoded bits. There are only 16 possibilities for this pattern, making it feasible to write all of them during the erase process. So to achieve good overwriting of (1,7) RLL disks, we write the patterns 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111. These patterns also conveniently cover two of the ones needed for

MFM overwrites, although we should add a few more iterations of the MFM-specific patterns for the reasons given above.

Finally, we have (2,7) RLL drives. Using a smaller encoding rate, an eight-bit-time signal corresponds to a repeating pattern of 4 data bits. Six-bit-time patterns can be written using 3-bit repeating patterns. The all-zero and all-one patterns overlap with the (1,7) RLL patterns, leaving six others:

0x24 0x92 0x49, 0x92 0x49 0x24 and 0x49 0x24 0x92 in hex, and
0x6D 0xB6 0xDB, 0xB6 0xDB 0x6D and 0xDB 0x6D 0xB6 in hex.

Although (1,7) is more popular in recent (post-1990) drives, some older hard drives do still use (2,7) RLL.

4.2 Design Issues:

Before discussion of detailed design of MyFS, there are some issues, which should be discussed.

1. Design should take into account the fact that enhanced file system should be able to work on older version of MINIX file systems- MINIX V1 and MINIX V2 file systems. Support for older version is not something one reads about in theoretical texts, but it is always a concern for the implementers of a new version of software.

2. Various new features should work on only the new file system, MyFS, not on the older version of file systems. This is necessary to preserve the semantics of older file system and for the consistency of the file systems. Such as file size should not exceed the restricted size on older versions of MINIX file systems that are imposed by them.

3. During secure deletion: If the device being written to supports caching or buffering of data, this should be disabled to ensure that physical disk writes are performed for each pass instead of everything but the last pass being lost in the buffering.

4.4 Design

4.4.1 Large Files

MINIX file system uses i-node of 64 bytes, which have 9 block pointers of 32 bits each. & direct pointer to access the data blocks, one to point a single indirect block which can point to 256 data block. Third pointer points double indirect blocks, which can point 256 indirect blocks, which further point the 256 data blocks. Thus we can have maximum file up to 65 MB.

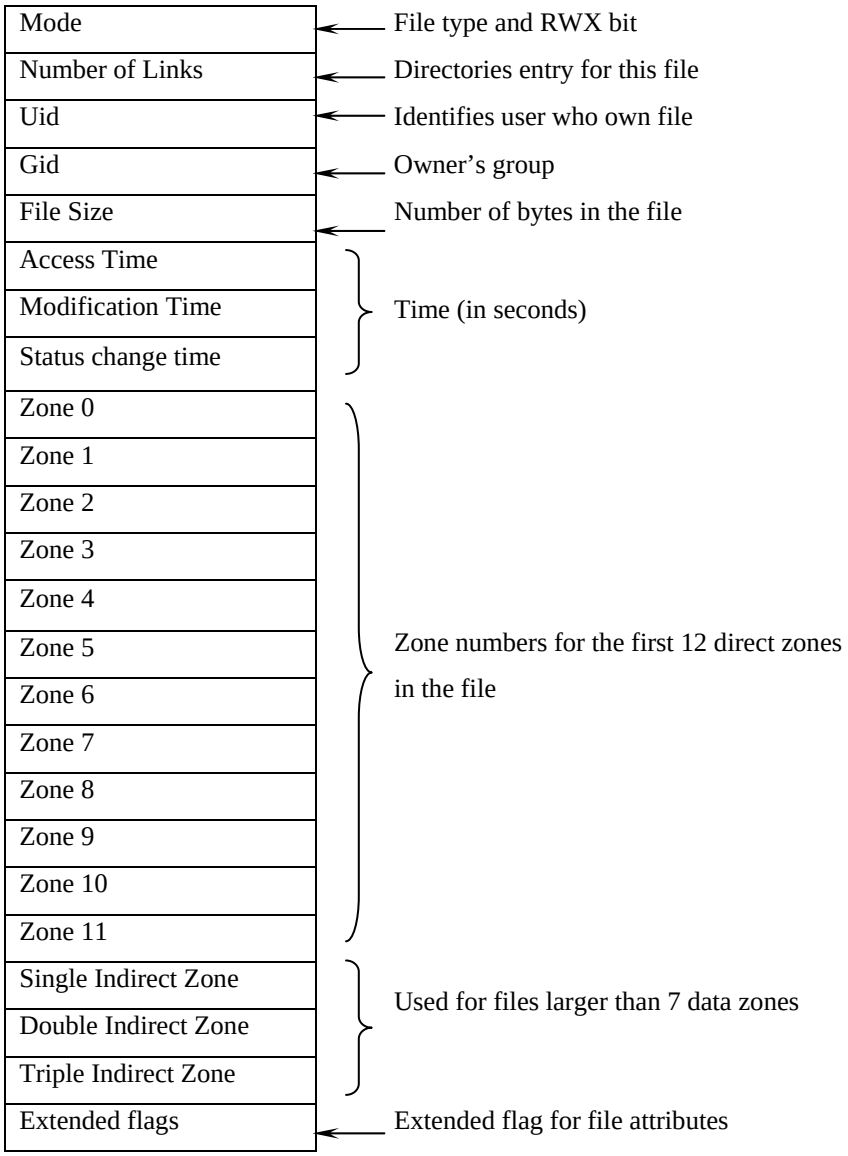

Figure 4.1 Modified structure of inode

To handle the larger files the triple indirect block have to be implemented. With larger block size (4 KB) and use of triple indirect block we can have maximum file size up to **~64 GB** theoretically, but the MINIX uses the 32 bit integer and long data type. Variable to store file position in bytes is 32 bit long and it restrict the maximum file size to 4GB. So after implementing triple indirect block we can handle the files of maximum size **4 GB**. The existing i-node structure has to be modified to accommodate these modifications. The modified i-node structure shown in figure 4.1.

4.4.2 Append-only and Immutable Files

To implement these two new types of files, a new file attribute flag: extended flags in the modified i-node structure have been introduced. In this *flag variable* user can set the file attributes to make any file or directory - append-only and/or immutable. When immutable file attribute is on, the file will stay as it is - cannot be changed, renamed, no hard links, etc, before the attribute is removed from the user. When the file attribute is set append-only user cannot modify old content of the file, only append new data on it.

31 – 5	E_COMPR 4	E_APPEND 3	E_IMMUTABLE 2	E_SYNC 1	E_SECRM 0
		Append-only file	Immutable file		secure delete file

Figure 4.2 extended flags of file

User can set or unset the file attribute to make the file immutable or append only by setting or resetting the respective *flags* in this *extended flag*. To make this possible some new request command values have been defined, to be sent to the **file control** function (**fcntl**).

```
#define F_SETETFL       8       /* set the file extended flags   */
#define F_GETETFL       9       /* get the files extended flags   */
```

There are some properties of immutable and append-only files/directories which to be considered, like:

(i) Append-only file can be opened with the append-only flag set, open call without this flag should fail.

(ii) Immutable file can be open in read-only mode; any write request should fail the operation.

(iii) When the directories given the append-only attribute, then user is not allowed to modify or rename the existing directory entries like file names, directory names. Link or unlink of the file or directory is also prohibited. User can append new entries to append only files, like creating new file or directory.

(iv) When the directories given the immutable attribute, then user can not modify or rename and delete the existing directory entries like file, directory. Link or unlink of the file or directory is also prohibited. New file or directory creation in an immutable directory is also prohibited.

4.4.3 Changes in Super Block

In the MINIX file system whenever the system crashes or improper shutdown is occurred the improper shutdown is detected by examining a system file which represents the last terminals information. There is no provision to store the file system status in the various file systems in their corresponding super block.

Number of nodes
Number of i-nodes bit map blocks
Number of zone bit map block
First data zone
Log_2 (block/zone)
Maximum File Size
Magic number
Padding
Number of Zones
Time of last file check
Maximum check interval
State

Figure 4.3 Modified super block

In order to remember the file system state, whether it is cleaned or contains error due to improper shutdown one new variable in the super block structure have been introduced. *State* variable in the super block remembers the state of the file system. The file system checking utility keeps track of the **file system state**. When a file system is mounted in read/write mode, its state is set to ``not clean''. When it is unmounted or remounted in read-only mode, its state is reset to ``Clean''. At boot time, the file system checker uses this information to decide if a file system must be checked.

Always skipping file system checks may sometimes be dangerous, to force checks at regular intervals two new variables are introduced. To force the regular file system check, two new variables have been introduced. *Last check time* and *maximal check interval* are maintained in the super block. *Time of last check* stores the time when the file system was checked for the consistency and errors last time. The *maximum check interval* is a parameter, which can be set by administrator at the file system creation, to store the time-period after that the file check to be forced. When the maximal check interval has been reached, the checker ignores the file System State and forces a file system check.

These variables make sure that the file system is checked for errors when the check interval time is elapsed. Support for Automatic consistency check on the file system status at the boot time, which may be activated after the system crash

4.4.4 Secure Deletion of Files

To implement the secure deletion on files, we have defined a new constant to be stored in the extended flags of the i-node along with immutable and append-only flags.

```
#define E_SECRM         1          /* secure deletion   */
```

User can set this file attribute to tell file system that this file to be deleted securely when removed. In the previous section we have understand the theory behind the choice of data

patterns to write and have taken a brief look at the recording methods used in disk drives on recording

From the method proposed by Peter Gutmann we now have a set of 35 overwrite patterns, which should erase everything, regardless of the raw encoding. The basic disk eraser has been improved slightly by adding random passes before and after the erase process. In this design 18 patterns of (1,7) RLL coding are choosed. To hide the secure overwriting nulls or zeroes are wrote on disk blocks before and after pattern writing. The 20 patterns have shown in bold letters in the table 4.2.

The deterministic patterns between the random writes are permuted before the write is performed, to make it more difficult for an opponent to use knowledge of the erasure data written to attempt to recover overwritten data. If the device being written to supports caching or buffering of data, this should be disabled to ensure that physical disk writes are performed for each pass instead of everything but the last pass being lost in the buffering.

Pass No	**Pattern written**
1	0x00 0x00 0x00
2	0x55 0x55 0x55
3	0xAA 0xAA 0xAA
4	0x00 0x00 0x00
5	0x11 0x11 0x11
6	0x22 0x22 0x22
7	0x33 0x33 0x33
8	0x44 0x44 0x44
9	0x55 0x55 0x55
10	0x66 0x66 0x66
11	0x77 0x77 0x77
12	0x88 0x88 0x88
13	0x99 0x99 0x99
14	0xAA 0xAA 0xAA
15	0xBB 0xBB 0xBB
16	0xCC 0xCC 0xCC
17	0xDD 0xDD 0xDD
18	0xEE 0xEE 0xEE
19	0xFF 0xFF 0xFF
20	0x00 0x00 0x00

Table 4.2 Patterns to overwrite data

CHAPTER 5 MyFS: IMPLEMENTATION

To implement the enhanced MINIX files system - MyFS, we are modifying some existing data structures and adding some new data structures. The major issue in implementation is that the MyFS file system should be able to handle different file systems, older MINIX V2 and MyFS, without any problem.

In this chapter we will discuss on those issues, which we faced during the implementation. In the implementation some places for constant names MINIX V3 been used, when referring MyFS.

5.1 Support for backward compatibility

In the implementation of MyFS, we have taken in to consideration of our design goal to keep the new file system backward compatible.

Every file system keeps a magic number in its super block structure to distinguish it from other file systems. In order to keep the file system to be able to differentiate between older MINIX file systems (V1 and V2) from MyFS, we have assign a different magic number for the MyFS file systems which to be stored in its super block.

```
/* Constant for MyFS or V3 file system */
#define SUPER_V3                    0x2244          /*magic number*/
#define SUPER_V3_REV                0x4422          /*swapped magic number */
#define V3                          3               /*version number*/

#define V3_NR_DZONES                12              /*number of direct block pointers */
#define V3_NR_TZONES                15              /*total /*number of block pointers*/*/
#define V3_I-NODE_SIZE              sizeof(d3_i-node) /*size of disk i-node*/
#define V3_I-NODES_PER_BLOCK   (BLOCK_SIZE / V3_I-NODE_SIZE)
                                                    /* number of i-nodes in a block*/
#define V3_INDIRECTS (BLOCK_SIZE /V3_ZONE_NUM_SIZE)
                                                    /* Number of block pointers in indirect block */
```

At the time of mounting the super block structure of file system is read from the disk in ***read_super*** function. The offset of *magic number* has been kept same for both file system and

new fields added to the super block are kept in the end of the super block structure. By checking the value of super magic the difference between the two file systems is handled.

```
if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
        version = V1;
  } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
        version = V2;
  } else if(magic==SUPER_V3 || magic == conv2(BYTE_SWAP, SUPER_V2)){
        version = V3;
}else{
    return (EINVAL);
}
```

There is different number of data block pointers in **MINIX V1**, **V2** and **MyFS** file systems. To handle these difference variables *s_version***,** *s_ndzones* and *s_nindirs* are used in the super block structure (which resides in memory not disk) to store the values of parameters, which are different for **V1**, **V2** and (V3) **MyFS**.

```
struct super_block {

        ino_t s_n_inodes;               /* number of usable i-nodes on the minor device */
        zone1_t  s_nzones;              /* total device size, including bit maps etc */
        short s_imap_blocks;            /* number of blocks used by i-node bit map */
        short s_zmap_blocks;            /* number of blocks used by zone bit map */
        zone1_t s_firstdatazone;        /* number of first data zone */
        short s_log_zone_size;          /* log2 of blocks/zone */
        off_t s_max_size;               /* maximum file size on this device */
        short s_magic;                  /* magic number to recognize super-blocks */
        short s_pad;                    /* try to avoid compiler-dependent padding */
        zone_t s_zones;                 /* number of zones (replaces s_nzones in V2) */
        time_t s_last_check;            /* time of last file check */
        unsigned long s_checkintervals; /* maximum allowed time interval b/w file check */
        short int s_state;              /* file system state: celan or error*/
        char s_prealloc_blocks;         /* how many blocks to preallocate  */
        off_t s_i-nodes_count;          /* total no of i-node count */

/* The following items are only used when the super_block is in memory. */

        struct i-node *s_isup;          /* i-node for root dir of mounted file sys */
        struct i-node *s_imount;        /* i-node mounted on */
        unsigned s_i-nodes_per_block;   /* precalculated from magic number */
        dev_t s_dev;                    /* whose super block is this? */
        int s_rd_only;                  /* set to 1 : file sys mounted read only */
        int s_native;                   /* set to 1 iff not byte swapping is req.*/
        int s_version;                  /* file system version */
        int s_ndzones;                  /* # direct zones in an i-node */
        int s_nindirs;                  /* # indirect zones per indirect block */
        bit_t s_isearch;                /* i-nodes below this bit no. are in use */
        bit_t s_zsearch;                /* all zones below this bit no. are in use*/
};
```

MINIX V2 file system uses the i-node of size 64 bytes and stores the 9 pointers for disk blocks, 7 to point data block directly and two for indirect pointers. **MyFS** uses the i-node of 128 Bytes, which contains 15 pointers to access disk blocks, 12 to point data block directly and 3 to point indirect blocks. It also uses the triple indirect block, which was not used in V2 file system.

All these details from other file system function and services have been separate by isolating them in the inode disk read-write function. When the i-nodes are read from the disk, in function **rw_i-node**, file system specific routines to read i-node from the disk have been called. Here various parameters in the **i-node table** have been adjusted such that the file system operations are isolated from these differences and remain in consistent for both type of file system i-node. The **old_icopy**, **new_icopy** and **new3_icopy** functions actually reads/writes i-node from/to disk and also adjust the parameters of i-node table accordingly.

```
if (sp->s_version == V1)                    /* for i-node of V1 file system*/
        old_icopy(rip, dip,  rw_flag, sp->s_native);
else if (sp->s_version == V3)               /* for i-node of V2 file system*/
    new3_icopy(rip, dip3, rw_flag, sp->s_native);
else                                        /* for i-node of V3/MyFS file system*/
        new_icopy(rip, dip2, rw_flag, sp->s_native);
```

5.2 Description of various features implemented

5.2.1 Regular file system check

MINIX file system uses a system file */usr/adm/wtmp* to store information of last shutdown request. At the time of initialization a script **rc** check if the system crashed last time and invoke the file checker for all file systems which having entry in the mount table.

In the older approach we keep track of status of the last system crash, not about the file system state. A better way is to store the file system state in to the super block, which keeps the file system state, whether it is cleaned or contains errors. Possible cause of error may be improper shutdown, system crash and improper unmounting of file system. For this purpose we have

included three new variables, *s_state, s_last_check* and *s_checkintervals* in the super block, which stores the file system state, time when file system was last checked for errors and maximum check interval allowed between two file system check.

In the new approach **shutdown** function checks the file state flag for file system-state, and invoke the file checker **fsck3.c** if the file system was not unmounted properly or the check interval is elapsed from the last file check. **Shutdown** function takes care of all the above cases and returns the *crash* value when called by **rc** script. Constants for above operations are defined in *super.h.*

```
#define VALID_FS 0x0001                 /* unmounted clearly */
#define ERROR_FS 0x0002                 /* errors detected   */

#define MAX_CHECK_INTERVAL      0x05265C07
/* Maximum time interval between file check is defined as 1 week. */
```

File system checker utility, **fsck3** checks the file system for errors and reset the file system state and set the last file check *s_last_check* to current time.

5.2.2 Large files

MyFS uses the i-node of 128 Bytes, which contains 15 pointers to access disk blocks, 12 to point data block directly and 3 to point indirect blocks. It uses the triple indirect block, which was not used in V2 file system. The structure of new i-node that is store on disk is given below.

```
typedef struct {
        mode_t d3_mode;             /* file type, protection, etc. */
        u16_t d3_nlinks;            /* how many links to this file */
        uid_t d3_uid;               /* user id of the file's owner */
        u16_t d3_gid;               /* group number */
        off_t d3_size;              /* current file size in bytes */
        time_t d3_atime;            /* time of last access (V2 only) */
        time_t d3_mtime;            /* when was file data last changed */
        time_t d3_ctime;            /* when was i-node itself changed (V2 only)*/
        zone_t d3_zone[V3_NR_TZONES];
                                    /* disk block numbers for direct, single indirect, dbl indirect
                                    and triple indirect block */
        off_t d3_blocks;            /* number disk block occupied by file */
        unsigned long d3_flags;     /*extende flags for file attributes. Used for immutable,
                                    append-only files and secure deletion  */
        unsigned long d3_file_acl;  /* for future purpose */
```

```
        unsigned long d3_dir_acl;       /* for future purpose */
        unsigned long d3_padd[7];       /*to make i-node of 128 Bytes*/

} d3_i-node;
```

Value of V3_NR_TZONES constant is 15.

Size of the i-node should be power of 2, so that we can write integral number of i-node in a block (of size 1024 Bytes.) So we have made the new i-node of size 128 Byte by adding some fields like *d3_file_acl, d3_dir_acl* and padding. Fields *d3_file_acl, d3_dir_acl* have been left for future purposes. Thus we insured that there will be integral number of i-nodes of 128 Bytes in a single block and file system can read it without any problem.

In the in the ***read.c*** and ***write.c*** files**,** read_map and write_map functions are modified to handle triple indirect blocks. **Read_map** function converts a logical file position to the physical block address by inspecting the i-node. For small files (up to 12 KB), block addresses are stored in i-node itself, but for large files one or more indirect blocks may have to be read. Function **write_map** do the opposite, write the block address in i-node or indirect block attached with that i-node. It must deal with several cases. For smaller files it just store the block address in i-node itself. But in the worst case when a file exceeds the size that can be handled by a single indirect block so a double indirect block is now required, next a single indirect block must be allocated and its address is put into the double indirect block.

We are using the triple indirect blocks in the MyFS, we have modified the **write_map** and **read_map** functions to handle this important operation of converting the logical file position the physical disk data block address.

MINIX V2 file system simply returns error for big files exceeding the size limit of 65 MB after the double indirect block is saturated. MyFS uses the tripe indirect block. **Write_map** creates a new triple indirect block, allocate a new double indirect block and further allocate a new single indirect block to store the disk address of the data block to write data on it. If the file further grows then the existing double indirect block and single indirect blocks are used to store the disk block addresses of data block. Brief functioning of the **rw_map** function is shown below.

```
/* We have pointer to i-node to be changed, file position to be mapped  and */
/* Disk data block number to be inserted.  */
/* Is 'file position' to be found in the i-node itself? Check for direct block addresses  */
/* Check whether 'file position' can be located via the single indirect block. */
/* Check whether 'file position' can be located via the double indirect block. */
/* If all above conditions are false then a new triple indirect block must be allocated */

if((z2=rip->i_zone[triple indirect block]) == NO_BLOCK )
 {       /* create a triple indirect block */
         if((z2 = alloc_zone(rip->i_dev,rip->i_zone[0])) == NO_ZONE)
              return (err_code);
        /*  store the address of triple indirect block in the i-node.*/
  }
  /* Remove offset of single indirect and double indirect blocks */
  /* calculate the index from file position to store or read block address of double indirect block */
  /* get the block number from data block . get triple indirect block of that number from the cache*/
  z = rd_indir(bp,ind_ex);
  /*read the disk address of double indirect block from that block.*/
  /* Check the double indirect block */
       if(z == NO_ZONE)
       {  /* Create the double indirect block  */
            if((z= alloc_zone(rip->i_dev,rip->i_zone[0])) == NO_ZONE)
                 return (err_code);
          /* Now update entry in triple indirect block */
          wr_indir(bp, ind_ex , z);
        }
        put_block(bp,INDIRECT_BLOCK);   /* write back the triple indirect block*/
        /* Now we have z as double indirect blocke  */
        /* calculate the index for single indirect  block address. */
        /*Get the double indirect block from the cache. */
       z1= rd_indir(bp,ind_ex);
       /* read the address of single indirect block from the double indirect block. */
       if (z1 == NO_ZONE) {    /*single indirect block is not allocated till.*/
      /* Create indirect block and store block number in i-node or double indirect block.*/
      z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);
  /* release dbl indirect blk */
   /*Get the single indirect block from the cache. */
  /*write the disk address of data block in it.*/
  /*put sinmgle indirect block bcack.*/
```

read_map function convert the file position to disk block address using indirect blocks in same way.

5.2.3 Append-only and Immutable Files

To implement these two new types of files, a new file attribute flag: extended flag *d_flags* in the modified i-node structure have been introduced.

User can set or unset the file attribute to make the file immutable or append only by setting or resetting the respective flag in this extended flag. To make this possible two new request

command values have been introduce in ***fcntl.h***, which can be set to the **file control** function (**fcntl)** to change the file attributes.

```
#define F_SETETFL               8       /* set the file extended flags   */
#define F_GETETFL               9       /* get the file extended flags    */
```

The new file attribute flags are define in ***fcntl.h***

```
#define E_IMMUTABLE             4
#define E_APPEND                8
```

Modification in ***misc.c*** is done in **fcntl** function to process the new request commands to change the file attributes. Now **fcntl** function can be called with modified constants and user can set the file attribute to make it immutable, append-only or set for secure deletion, as shown below

```
fcntl(fd, F_SETETFL, E_IMMUTABLE)               /*make file immutable    */
fcntl(fd, F_SETETFL, E_APPEND)                  /*make file append-only */
```

Along with these constants and commands some new macros to check file attribute for immutable or append-only file have been added in the ***fcntl.h*** .

```
#define IS_IMMUTABLE(rip)       rip->i_flag & E_IMMUTABLE
#define IS_APPEND(rip)          rip->i_flag & E_APPEND
```

To force the semantics of immutable and append-only files during the various file operation, file and parent directory attributes to be checked, when file to be opened, deleted, truncated and new file or directory is to be created, renamed or a new link is to be created. Whether the file system of file i-node is V3 or not have also be checked, because file attribute checks for these files are required in the MyFS file system only.

An example can describe it clearly. We want to rename a file then we have to check for the file attributes and its parent directory's attribute as well for immutable and append-only files.

```
 /* Check that the file or directory is immutable or not. We cannot
  rename an immutable file or Directory          */

 if(old_ip->i_sp->s_version ==V3 && IS_IMMUTABLE(old_ip)){
         printk("File/Directory is Immutable");
         return (EPERM); /*error for insufficient permission */
 }
```

```
/*Check whether the parent directory is immutable directory. We can not
modifies directory entries in an immutable directory. */

if(old_dirp->i_sp->s_version == V3 && IS_IMMUTABLE(old_dirp)){
        printk("Parent Directory is Immutable");
        return (EPERM); /*error for insufficient permission */
}

/* Check to see if the parent directory is APPEND only. We can
not modify the previous directory entry in a APPEND only Directory */

if(old_dirp->i_sp->s_version == V3 && IS_APPEND(old_dirp)){
        printk("Parent Directory is APPEND only");
        return (EPERM); /*error for insufficient permission */
}
```

5.2.4 Secure Deletion of Files

To implement the secure deletion of files, a new constant have to be defined and stored in the extended flags of the i-node along with immutable and append-only flags.

```
#define E_SECRM         1    /* secure deletion   */
```

User can set these file attributes to tell file system that this file to be deleted securely when removed. For this we can use the **fcntl** call with new request command as discussed in the implementation of immutable and append-only files. A new macro has been added in ***fcntl.h***, to check the secure remove property of file.

```
#define IS_SECRM(rip)               rip->i_flag & E_SECRM
```

In the design, theory behind the choice of data patterns to write has been discussed in concepts section. From the method proposed by Peter Gutmann, we now have a set of 20 overwrite patterns shown in table 4.2.

In the file **link.c** three new functions have been added: **overwrite**, **del_indirect** and **sec_delete.** In the following paragraphs the fief working of these functions are discussed.

O**verwrite** function actually overwrite the data blocks of the file with mentioned patterns. If the device being written to supports caching of data, this should be disabled to ensure that physical disk writes are performed for each pass instead of everything but the last pass being lost in the

buffering. So to achieve the desired affect of secure deletion all the patterns are required to write on the disk. **Overwrite** function bypass the cache and write the pattern on the disk block directly by calling disk input/output function, **dev_io.**

```
void overwrite(dev , z,scale)
dev_t dev;
zone_t z;
int scale;
{
static char bf[1024]={0,0};

char patt[20][3] = {{0,0,0},                    /*20 patterrns to be written on the data blocks*/
                    {0x55,0x55,0x55},
                    {0xAA,0xAA,0xAA},
                    {0x00,0x00,0x00},
                    {0x11,0x11,0x11},
                    { 0x22,0x22,0x22},
                    { 0x33,0x33,0x33},
                    { 0x44,0x44,0x44},
                    { 0x55,0x55,0x55},
                    { 0x66,0x66,0x66},
                    { 0x77,0x77,0x77},
                    { 0x88,0x88,0x88},
                    { 0x99,0x99,0x99},
                    { 0xAA,0xAA,0xAA},
                    { 0xBB,0xBB,0xBB},
                    { 0xCC,0xCC,0xCC},
                    { 0xDD,0xDD,0xDD},
                    { 0xEE,0xEE,0xEE},
                    { 0xFF,0xFF,0xFF},
                    { 0,0,0},

                };

pos = b * BLOCK_SIZE;                   /*calculate the file position for disk block */
flushall(dev);                          /*flush all the blocks of this device from cache*/

                                        /* Loop for the pattern number   */
for(p=0;p<20;p++)
 {
     /*Fill the buffer with the specific pattern    */
     /* Write the Data directly on the Disk, bypassing the Cache  */
     /* call the device task directly for block writing on disk.*/
    op = DEV_WRITE;                                 /*set the write operation/
    dev_io ( op, FALSE, dev, pos, BLOCK_SIZE, FS_PROC_NR, (char*)bf);
}
if(rip->i_nlinks == 0 && rip->i_sp->s_version == V3 && IS_SECRM(rip)){
      printk("Secure Deleting file");
      sec_delete(rip);
}
```

Files and directories are removed by unlinking them. The work of both the UNLINK and RMDIR system call is done by **do_unlink** function. For both system calls **unlink_file** function is called, which deletes the directory entry for the file or directory and decrement the link count of the i-node. Here we test whether the link count is zero and SEC_RM property is set for the file. Then we call our **sec_delete** function to delete the file's data and indirect blocks securely.
Del_sec function takes an i-node pointer for file/directory on secure deletion to be performed. It overwrites all the data blocks of file by calling **overwrite** function.

```
for (position =0; position < rip->i_size; position += block_size)
{
        if( (b = read_map(rip, position)) != NO_BLOCK) {
                z = (zone_t) b >> scale ;
                overwrite(dev, z, scale);
                free_zone(dev, z);
        }
}
```

Then it delete all the zone entries from the i-node, first direct block address from the i-node, single indirect block.

```
for( i=0; i< rip->i_ndzones; i++)                rip->i_zone[i] = 0;

if ( (z = rip->i_zone[single indirect ]) != NO_ZONE )          overwrite(dev,z,scale);
```

To delete double indirect blocks we first delete all single indirect blocks having the addresses in double indirect block by calling **del_indirect** function, and then delete the double indirect block itself.

```
if ( (z  = rip->i_zone[double indirect]) != NO_ZONE ) {
                del_indirects(dev,z,scale);          /*deletes all the single indirect blocks*/
                overwrite(dev,z,scale)                s/*delete double indirect block*/
}
```

We delete the triple indirect block in two steps. In first step we for loop for all double indirect blocks, fetching their address from triple indirect blocks. In second loop we call the **del_indirect** function for each double indirect block to delete all the single indirect blocks contained in that double indirect block.

```
if( (z = rip->i_zone[single+2]) != NO_ZONE ){
```

```
        for (i =0; i< V3_INDIRECTS; i++){
                z1 = rd_indir(bp,i);                    /*reads the address of single indirect block*/
                del_indirects(dev,z1,scale);
    }
}
```

5.2.5 Large Data Block:

MINIX V2 uses the data block of 1 KB, using data blocks of 4 KB in **MyFS.** The new file system to be backward compatible in order to handle older MINIX file systems. For that purpose a variable *s_log_zone* in the super block structure have been used, which is used to differentiate between data blocks (called zones) from simple blocks that hold i-node, directory information and indirect pointers. Variable *s_log_zone* gives base 2 logarithm of the zone to block ratio, which is used as the shift count to convert zones to block and vice versa. So we have blocks of size 1 KB to store i-node, directory entries, indirect pointers and data blocks (zones) of size 4 KB to store file data. File system check this variable when mounting the file system and sets the zone shift variable accordingly, which can be used at other file operations i.e. file read, write, etc to know the data block size.

To create a new file system – **MyFS**, a utility **mkfs3** (file mkfs3.c) have been developed. It creates MyFS file system on the specified disk partition. It writes the magic number for MyFS and zone shift for zone of 4KB in the super block of the file system. The new file system uses the new file system uses the new inode and super block structures to create the file system.

CHAPTER 6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this dissertation, design for a new File System – **MyFS** is proposed, which is an enhanced version of MINIX V2 file system and have some new features. During the course of this project, emphasis is given on various possible enhancements in the MINIX V2 file system for MINIX operating System. Kernel internal details of older file system have been studied thoroughly and the limitations of the file system are observed. Then design of an enhanced file system – **MyFS** is proposed, which lift restriction (maximum file size limit) and add new features for MINIX OS.

This project result in an enhanced MINIX file system - MyFS file system, without affecting the primary design of standard MINIX. The design also takes care backward compatibility issue. The enhanced file system can function upon older versions of MINIX file system like MINIX V1, MINIX V2 and new MyFS file system without any problem.

The design, as proposed in this dissertation, is also successfully implemented, installed and tested on the target machines in a ready to use stage. This complete implementation consists of several changes in the standard MINIX distribution.

In this dissertation first, we have break the *maximum file size limit (65MB)*, imposed by MINIX file system, using triple indirect block and larger data blocks. We have successfully created the files larger then 65MB and limit is extended to **~4 GB**.

Then, we have included tow new types of files- *immutable* and *append-only*. These new types of files and directories have been proved very useful in various applications and provided by many operating systems.

We have also included the feature of *secure file deletion*. The idea, which is inspired from a technical paper, is to overwrite the data so nobody can recover it using recovery tool or any special hardware.

At the last we have proposed and implemented some changes in super block to force *regular file system checking* for errors.

At last we have implemented a utility to create MyFS file system on any disk.

The comparison with other file system is shown in the table below:

	MINIX V1	**MINIX V2**	**ext FS**	**ext2 FS**	**MyFS**
Max FS size	64 MB	4 TB	2 GB	4 TB	**4 TB**
Max file size	64 MB	64 MB	2 GB	4 TB	**~4 GB**
3 times support	No	Yes	No	Yes	**Yes**
Secure Deletion	No	No	No	No	**Yes**
Immutable File	No	No	No	Yes	**Yes**
Append-only	No	No	No	Yes	**Yes**
FS state variable	No	No	No	Yes	**Yes**

Table 6.1 Comparison of MyFS with other file systems

This design is proposed keeping IBM PC and clones architecture, starting from 80386 onwards as target architecture.

6.2 Future work

This design has been proposed keeping IBM PC, 32-bit processor. In future, this design could be further extended to 64-bit processor, in which case file system can handle very large disks and files up to ~2TB.

In the present design, we have added some fields in file system data structure (in I-node- for compression and synchronous bits) that can be used to implement new features.

I_COMPR attribute of extended flags of i-node can be used to specify file system that this file to be stored on the disk in the compressed format. For this feature disks read and write functions can be modified to compress the file data before writing and decompress it at the time of reading. Research work is going on this feature.

I_SYNCH attribute of extended flags of i-node, can be used to specify file system that this file to be synchronized frequently as the data is modified. For this feature *cache* functions to be modified and a new type field be used to specify particular data block to be synchronized as soon as data of blocks are modified.

Many modern file systems like ext3; UNIX uses virtual file system (VFS) - an upper layer on the file system to handle file systems of different format and standards. A future possibility is to develop such a VFS layer for this file system to handle various files system like FAT16, FAT32, NTFS, ext2 and ext3.

Code of Selected Files

Include Directory files:

Fcntl.h
const.h
stat.h

files in fs directory:

const.h
type.h
super.h
inode.h
buf.h fs.h
inode.c
super.c
read.c
write.c
open.c
link.c
misc.c

utility functions files:

constant.h (for mkfs3.c)
mkfs3.c
fsck3.c

Fcntl.h

```
/* The <fcntl.h> header is needed by the open() and fcntl() system calls, which  have a variety of parameters and  flags.  They are described here.
The formats of the calls to each of these are:
 *      open(path, oflag [,mode])              open a file
 *      fcntl(fd, cmd [,arg])                  get or set file attributes  */

#ifndef _FCNTL_H
#define _FCNTL_H

/* These values are used for cmd in fcntl().  */
#define F_DUPFD                 0       /* duplicate file descriptor */
#define F_GETFD                 1       /* get file descriptor flags */
#define F_SETFD                 2       /* set file descriptor flags */
#define F_GETFL                 3       /* get file status flags */
#define F_SETFL                 4       /* set file status flags */
#define F_GETLK                 5       /* get record locking information */
#define F_SETLK                 6       /* set record locking information */
#define F_SETLKW                7       /* set record locking info; wait if blocked */

/*  Change for the New MINIX MyFS File system . Below new flags are used to set or get the inode extended flags for MyFS. */
#define F_SETETFL               8       /* set the file extended flags   */
#define F_GETETFL               9       /* get the file extended flags    */

/* This flags are for the extended file flags or the inode extended flags for the specail files in MyFS.   */
#define E_SECRM                 1       /* secure deletion   */
#define E_SYNC                  2       /* synchronous update */
#define E_IMMUTABLE             4       /* Imuutable file    */
#define E_APPEND                8       /* append only file  */
#define E_COMPR                 16      /* Compressed file   */
```

```
/* File descriptor flags used for fcntl().  */
#define FD_CLOEXEC                      1       /* close on exec flag for third arg of fcntl */

/* L_type values for record locking with fcntl().  */
#define F_RDLCK                         1       /* shared or read lock */
#define F_WRLCK                         2       /* exclusive or write lock */
#define F_UNLCK                         3       /* unlock */

/* Oflag values for open().  */
#define O_CREAT                         00100   /* creat file if it doesn't exist */
#define O_EXCL                          00200   /* exclusive use flag */
#define O_NOCTTY                        00400   /* do not assign a controlling terminal */
#define O_TRUNC                         01000   /* truncate flag */

/* File status flags for open() and fcntl().  */
#define O_APPEND                        02000   /* set append mode */
#define O_NONBLOCK                      04000   /* no delay */

/* File access modes for open() and fcntl().*/
#define O_RDONLY                        0       /* open(name, O_RDONLY) opens read only */
#define O_WRONLY                        1       /* open(name, O_WRONLY) opens write only */
#define O_RDWR                          2       /* open(name, O_RDWR) opens read/write */

/* Mask for use with file access modes.  */
#define O_ACCMODE                       03      /* mask for file access modes */

/* Struct used for locking.  */
struct flock {
        short l_type;                   /* type: F_RDLCK, F_WRLCK, or F_UNLCK */
        short l_whence;                 /* flag for starting offset */
        off_t l_start;                  /* relative offset in bytes */
        off_t l_len;                    /* size; if 0, then until EOF */
        pid_t l_pid;                    /* process id of the locks' owner */
};

/* Function Prototypes. */
#ifndef _ANSI_H
```

const.h

```
#define EXTERN                          extern  /* used in *.h files */
#define PRIVATE                         static  /* PRIVATE x limits the scope of x */
#define PUBLIC                                  /* PUBLIC is the opposite of PRIVATE */
#define FORWARD                         static  /* some compilers require this to be 'static'*/
#define TRUE                            1       /* used for turning integers into Booleans */
#define FALSE                           0       /* used for turning integers into Booleans */
#define HZ                              60      /* clock freq (software settable on IBM-PC) */
#define BLOCK_SIZE                      1024    /* # bytes in a disk block */
#define BLOCK_SIZENEW(sp)               sp->s_blocksize
#define SUPER_USER (uid_t)              0
#define MAJOR                           8       /* major device = (dev>>MAJOR) & 0377 */
#define MINOR                           0       /* minor device = (dev>>MINOR) & 0377 */
#define NULL                            ((void *)0) /* null pointer */
#define CPVEC_NR                        16      /* max # of entries in a SYS_VCOPY request */
#define NR_IOREQS                       MIN(NR_BUFS, 64)     /* maximum number of entries in an iorequest */
#define NR_SEGS                         3       /* # segments per process */
#define T                               0       /* proc[i].mem_map[T] is for text */
#define D                               1       /* proc[i].mem_map[D] is for data */
#define S                               2       /* proc[i].mem_map[S] is for stack */

/* Process numbers of some important processes. */
#define MM_PROC_NR                      0       /* process number of memory manager */
#define FS_PROC_NR                      1       /* process number of file system */
#define INET_PROC_NR                    2       /* process number of the TCP/IP server */
#define INIT_PROC_NR                    (INET_PROC_NR + ENABLE_NETWORKING)
                                        /* init -- the process that goes multiuser */
#define LOW_USER  (INET_PROC_NR + ENABLE_NETWORKING)
                                        /* first user not part of operating system */
/* Miscellaneous */
#define BYTE                    0377            /* mask for 8 bits */
```

```
#define READING                 0               /* copy data to user */
#define WRITING                 1               /* copy data from user */
#define NO_NUM                  0x8000          /* used as numerical argument to panic() */
#define NIL_PTR                 (char *) 0      /* generally useful expression */
#define HAVE_SCATTERED_IO  1                    /* scattered I/O is now standard */

/* Macros. */
#define MAX(a, b)   ((a) > (b) ? (a) : (b))
#define MIN(a, b)   ((a) < (b) ? (a) : (b))

/* Number of tasks. */
#define NR_TASKS                (9 + ENABLE_WINI + ENABLE_SCSI + ENABLE_CDROM \
                                        + ENABLE_NETWORKING + 2 * ENABLE_AUDIO)

/* Memory is allocated in clicks. */
#if (CHIP == INTEL)
#define CLICK_SIZE                      256     /* unit in which memory is allocated */
#define CLICK_SHIFT                     8       /* log2 of CLICK_SIZE */
#endif

#if (CHIP == SPARC) || (CHIP == M68000)
#define CLICK_SIZE                      4096    /* unit in which memory is alocated */
#define CLICK_SHIFT                     12      /* 2log of CLICK_SIZE */
#endif

#define click_to_round_k(n)     ((unsigned) ((((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))
#if CLICK_SIZE < 1024
#define k_to_click(n)           ((n) * (1024 / CLICK_SIZE))
#else
#define k_to_click(n)           ((n) / (CLICK_SIZE / 1024))
#endif

#define ABS                     -999    /* this process means absolute memory */

/* Flag bits for i_mode in the inode. */
#define I_TYPE                  0170000         /* this field gives inode type */
#define I_REGULAR               0100000         /* regular file, not dir or special */
#define I_BLOCK_SPECIAL         0060000         /* block special file */
#define I_DIRECTORY             0040000         /* file is a directory */
#define I_CHAR_SPECIAL          0020000         /* character special file */
#define I_NAMED_PIPE            0010000         /* named pipe (FIFO) */
#define I_SET_UID_BIT           0004000         /* set effective uid_t on exec */
#define I_SET_GID_BIT           0002000         /* set effective gid_t on exec */
#define ALL_MODES               0006777         /* all bits for user, group and others */
#define RWX_MODES               0000777         /* mode bits for RWX only */
#define R_BIT                   0000004         /* Rwx protection bit */
#define W_BIT                   0000002         /* rWx protection bit */
#define X_BIT                   0000001         /* rwX protection bit */
#define I_NOT_ALLOC             0000000         /* this inode is free */

/* Some limits. */
#define MAX_BLOCK_NR    ((block_t) 077777777)           /* largest block number */
#define HIGHEST_ZONE    ((zone_t) 077777777) /* largest zone number */
#define MAX_INODE_NR    ((ino_t) 0177777)               /* largest inode number */
#define MAX_FILE_POS            ((off_t) 037777777777)          /* largest legal file offset */

#define NO_BLOCK                ((block_t) 0)   /* absence of a block number */
#define NO_ENTRY                ((ino_t) 0)     /* absence of a dir entry */
#define NO_ZONE                 ((zone_t) 0)    /* absence of a zone number */
#define NO_DEV                  ((dev_t) 0)     /* absence of a device numb */
```

stat.h

```
/* The <sys/stat.h> header defines a struct that is used in the stat() and  fstat functions.  The information in this struct comes from the i-node of
some file.  These calls are the only approved way to inspect i-nodes. */

#ifndef _STAT_H
#define _STAT_H

struct stat {
```

```
        dev_t st_dev;           /* major/minor device number */
        ino_t st_ino;           /* i-node number */
        mode_t st_mode;         /* file mode, protection bits, etc. */
        short int st_nlink;     /* # links; TEMPORARY HACK: should be nlink_t*/
        uid_t st_uid;           /* uid of the file's owner */
        short int st_gid;       /* gid; TEMPORARY HACK: should be gid_t */
        dev_t st_rdev;
        off_t st_size;          /* file size */
        time_t st_atime;        /* time of last access */
        time_t st_mtime;        /* time of last data modification */
        time_t st_ctime;        /* time of last file status change */
};

#define S_IFMT          ((mode_t) 0170000)  /* type of file */
#define S_IFREG         ((mode_t) 0100000)  /* regular */
#define S_IFBLK         0060000             /* block special */
#define S_IFDIR         0040000             /* directory */
#define S_IFCHR         0020000             /* character special */
#define S_IFIFO         0010000             /* this is a FIFO */
#define S_ISUID         0004000             /* set user id on execution */
#define S_ISGID         0002000             /* set group id on execution */
#define S_ISVTX         01000               /* save swapped text even after use */

/* POSIX masks for st_mode. */
#define S_IRWXU         00700               /* owner:  rwx------ */
#define S_IRUSR         00400               /* owner:  r-------- */
#define S_IWUSR         00200               /* owner:  -w------- */
#define S_IXUSR         00100               /* owner:  --x------ */
#define S_IRWXG         00070               /* group:  ---rwx--- */
#define S_IRGRP         00040               /* group:  ---r----- */
#define S_IWGRP         00020               /* group:  ----w---- */
#define S_IXGRP         00010               /* group:  -----x--- */

#define S_IRWXO         00007               /* others: ------rwx */
#define S_IROTH         00004               /* others: ------r-- */
#define S_IWOTH         00002               /* others: -------w- */
#define S_IXOTH         00001               /* others: --------x */

/* The following macros test st_mode (from POSIX Sec. 5.6.1.1). */
#define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)     /* is a reg file */
#define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)     /* is a directory */
#define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)     /* is a char spec */
#define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)     /* is a block spec */
#define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)     /* is a pipe/FIFO */

/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( int chmod, (const char *_path, Mode_t _mode)        );
_PROTOTYPE( int fstat, (int _fildes, struct stat *_buf)         );
_PROTOTYPE( int mkdir, (const char *_path, Mode_t _mode)        );
_PROTOTYPE( int mkfifo, (const char *_path, Mode_t _mode)       );
_PROTOTYPE( int stat, (const char *_path, struct stat *_buf)    );
_PROTOTYPE( mode_t umask, (Mode_t _cmask)                       );
#endif /* _STAT_H */
```

files in fs directory:

const.h

```
/* Tables sizes */
#define V1_NR_DZONES    7       /* # direct zone numbers in a V1 inode */
#define V1_NR_TZONES    9       /* total # zone numbers in a V1 inode */
#define V2_NR_DZONES    7       /* # direct zone numbers in a V2 inode */
#define V2_NR_TZONES    10      /* total # zone numbers in a V2 inode */
```

```
#define NR_FILPS          128       /* # slots in filp table */
#define NR_INODES         64        /* # slots in "in core" inode table */
#define NR_SUPERS         8         /* # slots in super block table */
#define NR_LOCKS          8         /* # slots in the file locking table */

#define usizeof(t)        ((unsigned) sizeof(t))

/* File system types. */
#define SUPER_MAGIC       0x137F          /* magic number contained in super-block */
#define SUPER_REV         0x7F13          /* magic # when 68000 disk read on PC or vv */
#define SUPER_V2          0x2468          /* magic # for V2 file systems */
#define SUPER_V2_REV      0x6824          /* V2 magic written on PC, read on 68K or vv */

#define V1                1         /* version number of V1 file systems */
#define V2                2         /* version number of V2 file systems */

/* Miscellaneous constants */
#define SU_UID                      ((uid_t) 0) /* super_user's uid_t */
#define SYS_UID           ((uid_t) 0)       /* uid_t for processes MM and INIT */
#define SYS_GID           ((gid_t) 0)       /* gid_t for processes MM and INIT */
#define NORMAL            0                 /* forces get_block to do disk read */
#define NO_READ           1                 /* prevents get_block from doing disk read */
#define PREFETCH          2                 /* tells get_block not to read or mark dev */

#define XPIPE             (-NR_TASKS-1)           /* used in fp_task when susp'd on pipe */
#define XOPEN             (-NR_TASKS-2)           /* used in fp_task when susp'd on open */
#define XLOCK             (-NR_TASKS-3)           /* used in fp_task when susp'd on lock */
#define XPOPEN            (-NR_TASKS-4)           /* used in fp_task when susp'd on pipe open */

#define NO_BIT            ((bit_t) 0)       /* returned by alloc_bit() to signal failure */

#define DUP_MASK                  0100      /* mask to distinguish dup2 from dup */
#define LOOK_UP                   0         /* tells search_dir to lookup string */
#define ENTER                     1         /* tells search_dir to make dir entry */
#define DELETE                    2         /* tells search_dir to delete entry */
#define IS_EMPTY                  3         /* tells search_dir to ret. OK or ENOTEMPTY */
#define CLEAN                     0         /* disk and memory copies identical */
#define DIRTY                     1         /* disk and memory copies differ */
#define ATIME                     002       /* set if atime field needs updating */
#define CTIME                     004       /* set if ctime field needs updating */
#define MTIME                     010       /* set if mtime field needs updating */
#define BYTE_SWAP                 0         /* tells conv2/conv4 to swap bytes */
#define DONT_SWAP                 1         /* tells conv2/conv4 not to swap bytes */
#define END_OF_FILE               (-104)    /* eof detected */
#define ROOT_INODE                1         /* inode number for root directory */
#define BOOT_BLOCK                ((block_t) 0)       /* block number of boot block */
#define SUPER_BLOCK               ((block_t) 1)       /* block number of super block */

#define DIR_ENTRY_SIZE    usizeof (struct direct)                 /* # bytes/dir entry   */
#define NR_DIR_ENTRIES    (BLOCK_SIZE/DIR_ENTRY_SIZE)             /* # dir entries/blk   */
#define SUPER_SIZE                usizeof (struct super_block)    /* super_block size    */
#define PIPE_SIZE                 (V1_NR_DZONES*BLOCK_SIZE)  /* pipe size in bytes  */
#define BITMAP_CHUNKS     (BLOCK_SIZE/usizeof (bitchunk_t))       /* # map chunks/blk   */

/* Derived sizes pertaining to the V1 file system. */
#define V1_ZONE_NUM_SIZE          usizeof (zone1_t)               /* # bytes in V1 zone  */
#define V1_INODE_SIZE             usizeof (d1_inode)              /* bytes in V1 dsk ino */
#define V1_INDIRECTS              (BLOCK_SIZE/V1_ZONE_NUM_SIZE)   /* # zones/indir block */
#define V1_INODES_PER_BLOCK       (BLOCK_SIZE/V1_INODE_SIZE)      /* # V1 dsk inodes/blk */

/* Derived sizes pertaining to the V2 file system. */
#define V2_ZONE_NUM_SIZE          usizeof (zone_t)                /* # bytes in V2 zone  */
#define V2_INODE_SIZE             usizeof (d2_inode)              /* bytes in V2 dsk ino */
#define V2_INDIRECTS              (BLOCK_SIZE/V2_ZONE_NUM_SIZE)   /* # zones/indir block */
#define V2_INODES_PER_BLOCK       (BLOCK_SIZE/V2_INODE_SIZE)      /* # V2 dsk inodes/blk */
#define printf printk

/*Constants for the MINIX MyFS File System  */
#define V3_NR_DZONES              12        /*Numbe of Direct data block pointer*/
#define V3_NR_TZONES              15        /*Numbe of total data block pointer*/
```

```
#define V3_INODE_SIZE           usizeof(d3_inode)
#define V3_INODES_PER_BLOCK     (BLOCK_SIZE / V3_INODE_SIZE)
#define SUPER_V3                0x2244
#define SUPER_V3_REV            0x4422
#define V3                      3       /*MyFS version number*/
#define V3_ZONE_NUM_SIZE  usizeof(zone_t)
#define V3_INDIRECTS            (BLOCK_SIZE /V3_ZONE_NUM_SIZE)
```

type.h

```
/* Declaration of the V1 inode as it is on the disk (not in core). */
typedef struct {                        /* V1.x disk inode */
        mode_t d1_mode;                 /* file type, protection, etc. */
        uid_t d1_uid;                   /* user id of the file's owner */
        off_t d1_size;                  /* current file size in bytes */
        time_t d1_mtime;                /* when was file data last changed */
        gid_t d1_gid;                   /* group number */
        nlink_t d1_nlinks;              /* how many links to this file */
        u16_t d1_zone[V1_NR_TZONES];            /* block nums for direct, ind, and dbl ind */
} d1_inode;

/* Declaration of the V2 inode as it is on the disk (not in core). */
typedef struct {                        /* V2.x disk inode */
        mode_t d2_mode;                 /* file type, protection, etc. */
        u16_t d2_nlinks;                /* how many links to this file. HACK! */
        uid_t d2_uid;                   /* user id of the file's owner. */
        u16_t d2_gid;                   /* group number HACK! */
        off_t d2_size;                  /* current file size in bytes */
        time_t d2_atime;                /* when was file data last accessed */
        time_t d2_mtime;                /* when was file data last changed */
        time_t d2_ctime;                /* when was inode data last changed */
        zone_t d2_zone[V2_NR_TZONES];           /* block nums for direct, ind, and dbl ind */
} d2_inode;

/* Declaration of MINIX MyFS File System Disk I node entry     */
typedef struct {
        mode_t d3_mode;
        u16_t d3_nlinks;
        uid_t d3_uid;
        u16_t d3_gid;
        off_t d3_size;
        time_t d3_atime;
        time_t d3_mtime;
        time_t d3_ctime;
        zone_t d3_zone[V3_NR_TZONES];           /*15 data block pointers*/
        off_t d3_blocks;                        /*for future purpose*/
        unsigned long d3_flags;                 /*extended flags for new features*/
        unsigned long d3_file_acl;              /*for future purpose*/
        unsigned long d3_dir_acl;               /*for future purpose*/
        unsigned long d3_padd[7];               /*to complete the size of 128 bytes*/
}d3_inode;
```

super.h

/* Super block table. The root file system and every mounted file system has an entry here. The entry holds information about the sizes of the bit maps and inodes. The s_ninodes field gives the number of inodes available for files and directories, including the root directory. Inode 0 is on the disk, but not used. Thus s_ninodes = 4 means that 5 bits will be used in the bit map, bit 0, which is always 1 and not used, and bits 1-4 for files and directories. A super_block slot is free if s_dev == NO_DEV. */

```
/* These Macros are for MyFS file system  */
#define VALID_FS    0x0001              /* file system cleaned */
#define ERROR_FS    0x0002              /* file system have errors  */
```

```
EXTERN struct super_block {
        ino_t s_ninodes;            /* # usable inodes on the minor device */
        zone1_t  s_nzones;          /* total device size, including bit maps etc */
        short s_imap_blocks;        /* # of blocks used by inode bit map */
        short s_zmap_blocks;        /* # of blocks used by zone bit map */
        zone1_t s_firstdatazone;    /* number of first data zone */
        short s_log_zone_size;      /* log2 of blocks/zone */
        off_t s_max_size;           /* maximum file size on this device */
        short s_magic;              /* magic number to recognize super-blocks */
        short s_pad;                /* try to avoid compiler-dependent padding */
        zone_t s_zones;             /* number of zones (replaces s_nzones in V2) */

        /* new entries for the MyFS file system */
         time_t s_last_check;       /* last time, when file system was checked for error*/
        unsigned long s_checkintervals;  /*maximum allowable interal between two file checks*/
        short int s_state;          /*File system state variable*/
        time_t s_mtime;             /*file system mount time , for future purpose*/
        time_t s_wtime;             /*file system last write time, for future purpose*/
        short int s_mnt_count ;     /*file system mount count, for future purpose*/
        short int s_errors;         /*for future purpose*/
        char s_prealloc_blocks;
        off_t s_inodes_count;       /*file system inode count (long value), for future purpose*/

/* The following items are only used when the super_block is in memory. */
        struct inode *s_isup;       /* inode for root dir of mounted file sys */
        struct inode *s_imount;     /* inode mounted on */
        unsigned s_inodes_per_block;/* precalculated from magic number */
        dev_t s_dev;                /* whose super block is this? */
        int s_rd_only;              /* set to 1 iff file sys mounted read only */
        int s_native;               /* set to 1 iff not byte swapped file system */
        int s_version;              /* file system version, zero means bad magic */
        int s_ndzones;              /* # direct zones in an inode */
        int s_nindirs;              /* # indirect zones per indirect block */
        bit_t s_isearch;            /* inodes below this bit number are in use */
        bit_t s_zsearch;            /* all zones below this bit number are in use*/
} super_block[NR_SUPERS];

#define NIL_SUPER (struct super_block *) 0
#define IMAP                        0       /* operating on the inode bit map */
#define ZMAP                        1       /* operating on the zone bit map */
```

inode.h

```
/* Inode table.  This table holds inodes that are currently in use.  */
EXTERN struct inode {
        mode_t i_mode;                  /* file type, protection, etc. */
        nlink_t i_nlinks;               /* how many links to this file */
        uid_t i_uid;                    /* user id of the file's owner */
        gid_t i_gid;                    /* group number */
        off_t i_size;                   /* current file size in bytes */
        time_t i_atime;                 /* time of last access (V2 only) */
        time_t i_mtime;                 /* when was file data last changed */
        time_t i_ctime;                 /* when was inode itself changed (V2 only)*/
        zone_t i_zone[V3_NR_TZONES];    /* zone numbers for direct, ind, and dbl ind */
        off_t i_blocks;
        unsigned long i_flags;

  /* The following items are not present on the disk. */
        dev_t i_dev;                    /* which device is the inode on */
        ino_t i_num;                    /* inode number on its (minor) device */
        int i_count;                    /* # times inode used; 0 means slot is free */
        int i_ndzones;                  /* # direct zones (Vx_NR_DZONES) */
        int i_nindirs;                  /* # indirect zones per indirect block */
        struct super_block *i_sp;       /* pointer to super block for inode's device */
        char i_dirt;                    /* CLEAN or DIRTY */
        char i_pipe;                    /* set to I_PIPE if pipe */
        char i_mount;                   /* this bit is set if file mounted on */
        char i_seek;                    /* set on LSEEK, cleared on READ/WRITE */
        char i_update;                  /* the ATIME, CTIME, and MTIME bits are here */
```

```
} inode[NR_INODES];

#define NIL_INODE  (struct inode *) 0                   /* indicates absence of inode slot */

/* Field values.  Note that CLEAN and DIRTY are defined in "const.h" */
#define NO_PIPE                 0                       /* i_pipe is NO_PIPE if inode is not a pipe */
#define I_PIPE                  1                       /* i_pipe is I_PIPE if inode is a pipe */
#define NO_MOUNT                0                       /* i_mount is NO_MOUNT if file not mounted on*/
#define I_MOUNT                 1                       /* i_mount is I_MOUNT if file mounted on */
#define NO_SEEK                 0                       /* i_seek = NO_SEEK if last op was not SEEK */
#define ISEEK                   1                       /* i_seek = ISEEK if last op was SEEK */
```

buf.h

```
#include <sys/dir.h>                            /* need struct direct */

EXTERN struct buf {
        /* Data portion of the buffer. */
  union {
        char b__data[BLOCK_SIZE];                       /* ordinary user data */
        struct direct b__dir[NR_DIR_ENTRIES];           /* directory block */
        zone1_t b__v1_ind[V1_INDIRECTS];                /* V1 indirect block */
        zone_t  b__v2_ind[V2_INDIRECTS];                /* V2 indirect block */
        d1_inode b__v1_ino[V1_INODES_PER_BLOCK];    /* V1 inode block */
        d2_inode b__v2_ino[V2_INODES_PER_BLOCK];    /* V2 inode block */
        bitchunk_t b__bitmap[BITMAP_CHUNKS];            /* bit map block */
        zone_t b__v3_ind[V3_INDIRECTS];                 /* MyFS inode block */
        d3_inode b__v3_ino[V3_INODES_PER_BLOCK];        /* MyFS Indirect block */
  } b;

  /* Header portion of the buffer. */
  struct buf *b_next;                           /* used to link all free bufs in a chain */
  struct buf *b_prev;                           /* used to link all free bufs the other way */
  struct buf *b_hash;                           /* used to link bufs on hash chains */
  unsigned long b_blocknr;                      /* block number of its (minor) device */
  dev_t b_dev;                                  /* major | minor device where block resides */
  char b_dirt;                                  /* CLEAN or DIRTY */
  char b_count;                                 /* number of users of this buffer */
} buf[NR_BUFS];

/* A block is free if b_dev == NO_DEV. */

#define NIL_BUF ((struct buf *) 0)              /* indicates absence of a buffer */

/* These defs make it possible to use to bp->b_data instead of bp->b.b__data */
#define b_data                  b.b__data
#define b_dir                   b.b__dir
#define b_v1_ind                b.b__v1_ind
#define b_v2_ind                b.b__v2_ind
#define b_v1_ino                b.b__v1_ino
#define b_v2_ino                b.b__v2_ino
#define b_bitmap                b.b__bitmap
#define b_v3_ind                b.b__v3_ind
#define b_v3_ino                b.b__v3_ino

struct buf *buf_hash[NR_BUF_HASH];              /* the buffer hash table */

EXTERN struct buf *front;                       /* points to least recently used free block */
EXTERN struct buf *rear;                        /* points to most recently used free block */
EXTERN int bufs_in_use;                         /* # bufs currently in use (not on free list)*/

/* When a block is released, the type of usage is passed to put_block(). */
#define WRITE_IMMED             0100    /* block should be written to disk now */
#define ONE_SHOT                0200    /* set if block not likely to be needed soon */

#define INODE_BLOCK             (0 + MAYBE_WRITE_IMMED)         /* inode block */
#define DIRECTORY_BLOCK         (1 + MAYBE_WRITE_IMMED)         /* directory block */
#define INDIRECT_BLOCK          (2 + MAYBE_WRITE_IMMED)         /* pointer block */
#define MAP_BLOCK               (3 + MAYBE_WRITE_IMMED)         /* bit map */
```

```
#define ZUPER_BLOCK          (4 + WRITE_IMMED + ONE_SHOT)        /* super block */
#define FULL_DATA_BLOCK      5                                   /* data, fully used */
#define PARTIAL_DATA_BLOCK   6                                   /* data, partly used*/
#define HASH_MASK            (NR_BUF_HASH - 1)                   /* mask for hashing block numbers */
```

fs.h

```
/* This is the master header for fs.  It includes some other files  and defines the principal constants.*/
#define _POSIX_SOURCE                1          /* tell headers to include POSIX stuff */
#define _MINIX                       1          /* tell headers to include MINIX stuff */
#define _SYSTEM                      1          /* tell headers that this is the kernel */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>                       /* MUST be first */
#include <ansi.h>                               /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <limits.h>
#include <errno.h>
#include <minix/syslib.h>
#include "const.h"
#include "type.h"
#include "proto.h"
#include "glo.h"
```

inode.c

```
/* This file manages the inode table.  There are procedures to allocate and  deallocate inodes, acquire, erase, and release them, and read and write
them from the disk.
 * The entry points into this file are
 *   get_inode:          search inode table for a given inode; if not there, read it
 *   put_inode:          indicate that an inode is no longer needed in memory
 *   alloc_inode:        allocate a new, unused inode
 *   wipe_inode:         erase some fields of a newly allocated inode
 *   free_inode:         mark an inode as available for a new file
 *   update_times:       update atime, ctime, and mtime
 *   rw_inode:           read a disk block and extract an inode, or corresp. write
 *   old_icopy:          copy to/from in-core inode struct and disk inode (V1.x)
 *   new_icopy:          copy to/from in-core inode struct and disk inode (V2.x)
 *   dup_inode:          indicate that someone else is using an inode table entry */

#include "fs.h"
#include <minix/boot.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"
#include "fcntl.h"

FORWARD _PROTOTYPE( void old_icopy, (struct inode *rip, d1_inode *dip, int direction, int norm));
FORWARD _PROTOTYPE( void new_icopy, (struct inode *rip, d2_inode *dip, int direction, int norm));
FORWARD _PROTOTYPE(  void new3_icopy, (struct inode *rip, d3_inode *dip3, int direction, int norm));

PUBLIC struct inode *get_inode(dev, numb)
dev_t dev;                      /* device on which inode resides */
int numb;                       /* inode number (ANSI: may not be unshort) */
{
/* Find a slot in the inode table, load the specified inode into it, and return a pointer to the slot.  If 'dev' == NO_DEV,  just return a free slot. */
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++) {
        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. */
                        rip->i_count++;
                        return(rip);            /* (dev, numb) found */
```

```
                        }
                } else {
                        xp = rip;                       /* remember this free slot for later */
                }
 }

 /* Inode we want is not currently in use.  Did we find a free slot? */
 if (xp == NIL_INODE) {                                 /* inode table completely full */
                err_code = ENFILE;
                return(NIL_INODE);
 }

 /* A free inode slot has been located.  Load the inode into it. */
 xp->i_dev = dev;
 xp->i_num = numb;
 xp->i_count = 1;
 if (dev != NO_DEV) rw_inode(xp, READING);          /* get inode from disk */
 xp->i_update = 0;                                      /* all the times are initially up-to-date */
 return(xp);
}

PUBLIC void put_inode(rip)
register struct inode *rip;             /* pointer to inode to be released */
{
/* The caller is no longer using this inode.  If no one else is using it either  write it back to the disk immediately.  If
it has no links, truncate it and  return it to the pool of available inodes. */

  if (rip == NIL_INODE) return;                 /* checking here is easier than in caller */
  if (--rip->i_count == 0) {                    /* i_count == 0 means no one is using it now */
                if ((rip->i_nlinks & BYTE) == 0) {
                        /* i_nlinks == 0 means free the inode. */
                        truncate(rip);                  /* return all the disk blocks */
                        rip->i_mode = I_NOT_ALLOC;  /* clear I_TYPE field */
                        rip->i_dirt = DIRTY;
                        free_inode(rip->i_dev, rip->i_num);
                } else {
                        if (rip->i_pipe == I_PIPE) truncate(rip);
                }
                rip->i_pipe = NO_PIPE;              /* should always be cleared */
                if (rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
  }
}

PUBLIC struct inode *alloc_inode(dev, bits)
dev_t dev;                                      /* device on which to allocate the inode */
mode_t bits;                                    /* mode of the inode */
{
/* Allocate a free inode on 'dev', and return a pointer to it. */

  register struct inode *rip;
  register struct super_block *sp;
  int major, minor, inumb;
  bit_t b;

  sp = get_super(dev);                          /* get pointer to super_block */
  if (sp->s_rd_only) {                          /* can't allocate an inode on a read only device. */
                err_code = EROFS;
                return(NIL_INODE);
  }

  /* Acquire an inode from the bit map. */
  b = alloc_bit(sp, IMAP, sp->s_isearch);
  if (b == NO_BIT) {
                err_code = ENFILE;
                major = (int) (sp->s_dev >> MAJOR) & BYTE;
                minor = (int) (sp->s_dev >> MINOR) & BYTE;
                printf("Out of i-nodes on %sdevice %d/%d\n",
                        sp->s_dev == ROOT_DEV ? "root " : "", major, minor);
                return(NIL_INODE);
  }
```

```
  sp->s_isearch = b;                    /* next time start here */
  inumb = (int) b;                      /* be careful not to pass unshort as param */

  /* Try to acquire a slot in the inode table. */
  if ((rip = get_inode(NO_DEV, inumb)) == NIL_INODE) {
        /* No inode table slots available.  Free the inode just allocated. */
        free_bit(sp, IMAP, b);
  } else {
        /* An inode slot is available. Put the inode just allocated into it. */
        rip->i_mode = bits;                     /* set up RWX bits */
        rip->i_nlinks = (nlink_t) 0;            /* initial no links */
        rip->i_uid = fp->fp_effuid;             /* file's uid is owner's */
        rip->i_gid = fp->fp_effgid;             /* ditto group id */
        rip->i_dev = dev;                       /* mark which device it is on */
        rip->i_ndzones = sp->s_ndzones;         /* number of direct zones */
        rip->i_nindirs = sp->s_nindirs;         /* number of indirect zones per blk*/
        rip->i_sp = sp;                         /* pointer to super block */
        rip->i_flags = 0;                       /*clear the extended flag*/

/* Fields not cleared already are cleared in wipe_inode().  They have  been put there because truncate()
needs to clear the same fields if  the file happens to be open while being truncated.  It saves space not to repeat the code twice.  */
        wipe_inode(rip);
  }
  return(rip);
}

PUBLIC void wipe_inode(rip)
register struct inode *rip;             /* the inode to be erased */
{ /* Erase some fields in the inode.  This function is called from alloc_inode()  * when a new inode is to
be allocated, and from truncate(), when an existing  * inode is to be truncated.  */

  register int i;

  rip->i_size = 0;
  rip->i_update = ATIME | CTIME | MTIME;        /* update all times later */
  rip->i_dirt = DIRTY;
  for (i = 0; i < V3_NR_TZONES; i++) rip->i_zone[i] = NO_ZONE;
}

PUBLIC void rw_inode(rip, rw_flag)
register struct inode *rip;             /* pointer to inode to be read/written */
int rw_flag;                            /* READING or WRITING */
{
  register struct buf *bp;
  register struct super_block *sp;
  d1_inode *dip;
  d2_inode *dip2;
  d3_inode *dip3;
  block_t b, offset;

  /* Get the block where the inode resides. */
  sp = get_super(rip->i_dev);           /* get pointer to super block */
  rip->i_sp = sp;                       /* inode must contain super block pointer */
  offset = sp->s_imap_blocks + sp->s_zmap_blocks + 2;
  b = (block_t) (rip->i_num - 1)/sp->s_inodes_per_block + offset;
  bp = get_block(rip->i_dev, b, NORMAL);
  dip  = bp->b_v1_ino + (rip->i_num - 1) % V1_INODES_PER_BLOCK;
  dip2 = bp->b_v2_ino + (rip->i_num - 1) % V2_INODES_PER_BLOCK;
  dip3 =  bp->b_v3_ino + (rip->i_num -1) % V3_INODES_PER_BLOCK;

  /* Do the read or write. */
  if (rw_flag == WRITING) {
        if (rip->i_update) update_times(rip);       /* times need updating */
        if (sp->s_rd_only == FALSE) bp->b_dirt = DIRTY;
  }

  /* Copy the inode from the disk block to the in-core table or vice versa. If the fourth parameter below is FALSE,
the bytes are swapped. */
  if (sp->s_version == V1)
        old_icopy(rip, dip,  rw_flag, sp->s_native);
```

```
 else if (sp->s_version == V3)
      new3_icopy(rip, dip3, rw_flag, sp->s_native);
  else
            new_icopy(rip, dip2, rw_flag, sp->s_native);

 put_block(bp, INODE_BLOCK);
 rip->i_dirt = CLEAN;
}

PRIVATE void old_icopy(rip, dip, direction, norm)
register struct inode *rip;                     /* pointer to the in-core inode struct */
register d1_inode *dip;                         /* pointer to the d1_inode inode struct */
int direction;                                  /* READING (from disk) or WRITING (to disk) */
int norm;                                       /* TRUE = do not swap bytes; FALSE = swap */

{
 int i;

 if (direction == READING) {
            /* Copy V1.x inode to the in-core table, swapping bytes if need be. */
            rip->i_mode         = conv2(norm, (int) dip->d1_mode);
            rip->i_uid          = conv2(norm, (int) dip->d1_uid );
            rip->i_size         = conv4(norm, dip->d1_size);
            rip->i_mtime        = conv4(norm, dip->d1_mtime);
            rip->i_atime        = rip->i_mtime;
            rip->i_ctime        = rip->i_mtime;
            rip->i_nlinks       = (nlink_t) dip->d1_nlinks;             /* 1 char */
            rip->i_gid          = (gid_t) dip->d1_gid;                  /* 1 char */
            rip->i_ndzones      = V1_NR_DZONES;
            rip->i_nindirs      = V1_INDIRECTS;
            rip->i_blocks       = rip->i_size/BLOCK_SIZE;
            rip->i_flags        = 0;
            for (i = 0; i < V1_NR_TZONES; i++)
                        rip->i_zone[i] = conv2(norm, (int) dip->d1_zone[i]);
 } else {
            /* Copying V1.x inode to disk from the in-core table. */
            dip->d1_mode        = conv2(norm, (int) rip->i_mode);
            dip->d1_uid         = conv2(norm, (int) rip->i_uid );
            dip->d1_size        = conv4(norm, rip->i_size);
            dip->d1_mtime       = conv4(norm, rip->i_mtime);
            dip->d1_nlinks      = (nlink_t) rip->i_nlinks;      /* 1 char */
            dip->d1_gid         = (gid_t) rip->i_gid;           /* 1 char */
            for (i = 0; i < V1_NR_TZONES; i++)
                        dip->d1_zone[i] = conv2(norm, (int) rip->i_zone[i]);
 }
}

PRIVATE void new_icopy(rip, dip, direction, norm)
register struct inode *rip;                     /* pointer to the in-core inode struct */
register d2_inode *dip;                         /* pointer to the d2_inode struct */
int direction;                                  /* READING (from disk) or WRITING (to disk) */
int norm;                                       /* TRUE = do not swap bytes; FALSE = swap */
{           /* Same as old_icopy, but to/from V2 disk layout. */

 int i;

 if (direction == READING) {
            /* Copy V2.x inode to the in-core table, swapping bytes if need be. */
            rip->i_mode         = conv2(norm,dip->d2_mode);
            rip->i_uid          = conv2(norm,dip->d2_uid );
            rip->i_nlinks       = conv2(norm,(int) dip->d2_nlinks);
            rip->i_gid          = conv2(norm,(int) dip->d2_gid );
            rip->i_size         = conv4(norm,dip->d2_size);
            rip->i_atime        = conv4(norm,dip->d2_atime);
            rip->i_ctime        = conv4(norm,dip->d2_ctime);
            rip->i_mtime        = conv4(norm,dip->d2_mtime);
            rip->i_ndzones      = V2_NR_DZONES;
            rip->i_nindirs      = V2_INDIRECTS;
            rip->i_blocks       = rip->i_size / BLOCK_SIZE;
            rip->i_flags        = 0;
```

```
        for (i = 0; i < V2_NR_TZONES; i++)
                rip->i_zone[i] = conv4(norm, (long) dip->d2_zone[i]);
 } else {
        /* Copying V2.x inode to disk from the in-core table. */
        dip->d2_mode            = conv2(norm,rip->i_mode);
        dip->d2_uid             = conv2(norm,rip->i_uid );
        dip->d2_nlinks          = conv2(norm,rip->i_nlinks);
        dip->d2_gid             = conv2(norm,rip->i_gid );
        dip->d2_size            = conv4(norm,rip->i_size);
        dip->d2_atime           = conv4(norm,rip->i_atime);
        dip->d2_ctime           = conv4(norm,rip->i_ctime);
        dip->d2_mtime           = conv4(norm,rip->i_mtime);
        for (i = 0; i < V2_NR_TZONES; i++)
                dip->d2_zone[i] = conv4(norm, (long) rip->i_zone[i]);
 }
}

/**********************new 3 i copy is used for the inode copy of MyFS file system*/
/*****************************************************************************/
PRIVATE void new3_icopy(rip, dip3,direction, norm)
register struct inode * rip;
register d3_inode * dip3;
int direction;
int norm;
{ /*  handle the diferences of MyFS inode from V1 and V2 inodes  */

int i;

if(direction == READING)  {
        /*copying MyFS inode data to the in-core inode table */
        rip->i_mode = conv2(norm, dip3->d3_mode);
        rip->i_uid  = conv2(norm,dip3->d3_uid);
        rip->i_nlinks = conv2(norm,(int)dip3->d3_nlinks);
        rip->i_gid = conv2(norm,(int)dip3->d3_gid);
        rip->i_size = conv4(norm, dip3->d3_size);
        rip->i_atime = conv4(norm, dip3->d3_atime);
        rip->i_mtime = conv4(norm, dip3->d3_mtime);
        rip->i_ctime = conv4(norm, dip3->d3_ctime);
        rip->i_ndzones = V3_NR_DZONES;
        rip->i_nindirs =  V3_INDIRECTS;
        for(i =0; i< V3_NR_TZONES; i++)
                rip->i_zone[i] = conv4(norm, (long) dip3->d3_zone[i]);
        rip->i_blocks = conv4(norm,dip3->d3_blocks);
        rip->i_flags = E_SECRM; /* conv4(norm,dip3->d3_flags); */

} else {
        dip3->d3_mode = conv2(norm,rip->i_mode);
        dip3->d3_uid = conv2(norm, rip->i_uid);
        dip3->d3_nlinks = conv2(norm, rip->i_nlinks);
        dip3->d3_gid = conv2(norm, rip->i_gid);
        dip3->d3_size = conv4(norm, rip->i_size);
        dip3->d3_atime = conv4(norm, rip->i_atime);
        dip3->d3_mtime = conv4(norm, rip->i_mtime);
        dip3->d3_ctime = conv4(norm, rip->i_ctime);
         for (i=0; i< V3_NR_TZONES; i++)
                dip3->d3_zone[i] = conv4(norm, (long) rip->i_zone[i]);
        dip3->d3_blocks = conv4 (norm,rip->i_blocks);
        dip3->d3_flags =  E_SECRM;/*conv4(norm, rip->i_flags)  */
   }
}

PUBLIC void dup_inode(ip)
struct inode *ip;                       /* The inode to be duplicated. */
{
 ip->i_count++;
}
```

super.c

```
/* This file manages the super block table and the related data structures. */
#include "fs.h"
#include <string.h>
#include <minix/boot.h>
#include "buf.h"
#include "inode.h"
#include "super.h"

#define BITCHUNK_BITS                   (usizeof(bitchunk_t) * CHAR_BIT)
#define BITS_PER_BLOCK                  (BITMAP_CHUNKS * BITCHUNK_BITS)

PUBLIC bit_t alloc_bit(sp, map, origin)
struct super_block *sp;                 /* the filesystem to allocate from */
int map;                                /* IMAP (inode map) or ZMAP (zone map) */
bit_t origin;                           /* number of bit to start searching at */
{ /* Allocate a bit from a bit map and return its bit number. */

  block_t start_block;                  /* first bit block */
  bit_t map_bits;                       /* how many bits are there in the bit map? */
  unsigned bit_blocks;                  /* how many blocks are there in the bit map? */
  unsigned block, word, bcount, wcount;
  struct buf *bp;
  bitchunk_t *wptr, *wlim, k;
  bit_t i, b;

  if (sp->s_rd_only)
        panic("can't allocate bit on read-only filesys.", NO_NUM);

  if (map == IMAP) {
        start_block = SUPER_BLOCK + 1;
        map_bits = sp->s_ninodes + 1;
        bit_blocks = sp->s_imap_blocks;
  } else {
        start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
        map_bits = sp->s_zones - (sp->s_firstdatazone - 1);
        bit_blocks = sp->s_zmap_blocks;
  }
  /* Figure out where to start the bit search (depends on 'origin'). */
  if (origin >= map_bits) origin = 0;           /* for robustness */

  /* Locate the starting place. */
  block = origin / BITS_PER_BLOCK;
  word = (origin % BITS_PER_BLOCK) / BITCHUNK_BITS;

  /* Iterate over all blocks plus one, because we start in the middle. */
  bcount = bit_blocks + 1;
  do {
        bp = get_block(sp->s_dev, start_block + block, NORMAL);
        wlim = &bp->b_bitmap[BITMAP_CHUNKS];

        /* Iterate over the words in block. */
        for (wptr = &bp->b_bitmap[word]; wptr < wlim; wptr++) {

                /* Does this word contain a free bit? */
                if (*wptr == (bitchunk_t) ~0) continue;

                /* Find and allocate the free bit. */
                k = conv2(sp->s_native, (int) *wptr);
                for (i = 0; (k & (1 << i)) != 0; ++i) {}

                /* Bit number from the start of the bit map. */
                b = ((bit_t) block * BITS_PER_BLOCK)
                   + (wptr - &bp->b_bitmap[0]) * BITCHUNK_BITS
                   + i;

                /* Don't allocate bits beyond the end of the map. */
                if (b >= map_bits) break;

                /* Allocate and return bit number. */
                k |= 1 << i;
```

```
                        *wptr = conv2(sp->s_native, (int) k);
                        bp->b_dirt = DIRTY;
                        put_block(bp, MAP_BLOCK);
                        return(b);
                }
                put_block(bp, MAP_BLOCK);
                if (++block >= bit_blocks) block = 0;           /* last block, wrap around */
                word = 0;
  } while (--bcount > 0);
  return(NO_BIT);                       /* no bit could be allocated */
}

PUBLIC void free_bit(sp, map, bit_returned)
struct super_block *sp;                         /* the filesystem to operate on */
int map;                                /* IMAP (inode map) or ZMAP (zone map) */
bit_t bit_returned;                     /* number of bit to insert into the map */
{
/* Return a zone or inode by turning off its bitmap bit. */

  unsigned block, word, bit;
  struct buf *bp;
  bitchunk_t k, mask;
  block_t start_block;

  if (sp->s_rd_only)
                panic("can't free bit on read-only filesys.", NO_NUM);

  if (map == IMAP) {
                start_block = SUPER_BLOCK + 1;
  } else {
                start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
  }
  block = bit_returned / BITS_PER_BLOCK;
  word = (bit_returned % BITS_PER_BLOCK) / BITCHUNK_BITS;
  bit = bit_returned % BITCHUNK_BITS;
  mask = 1 << bit;

  bp = get_block(sp->s_dev, start_block + block, NORMAL);
  k = conv2(sp->s_native, (int) bp->b_bitmap[word]);
  if (!(k & mask)) {
                panic(map == IMAP ? "tried to free unused inode" :
                    "tried to free unused block", NO_NUM);
  }

  k &= ~mask;
  bp->b_bitmap[word] = conv2(sp->s_native, (int) k);
  bp->b_dirt = DIRTY;

  put_block(bp, MAP_BLOCK);
}

PUBLIC struct super_block *get_super(dev)
dev_t dev;                              /* device number whose super_block is sought */
{   /* Search the superblock table for this device.  It is supposed to be there. */

  register struct super_block *sp;

  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
                if (sp->s_dev == dev) return(sp);

  /* Search failed.  Something wrong. */
  panic("can't find superblock for device (in decimal)", (int) dev);

  return(NIL_SUPER);                            /* to keep the compiler and lint quiet */
}

PUBLIC int mounted(rip)
register struct inode *rip;             /* pointer to inode */
{ /* Report on whether the given inode is on a mounted (or ROOT) file system. */
```

```
  register struct super_block *sp;
  register dev_t dev;

  dev = (dev_t) rip->i_zone[0];
  if (dev == ROOT_DEV) return(TRUE);      /* inode is on root file system */

  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
                if (sp->s_dev == dev) return(TRUE);

  return(FALSE);
}

PUBLIC int read_super(sp)
register struct super_block *sp; /* pointer to a superblock */
{ /* Read a superblock. */

  register struct buf *bp;
  dev_t dev;
  int magic;
  int version, native;

  dev = sp->s_dev;                      /* save device (will be overwritten by copy) */
  bp = get_block(sp->s_dev, SUPER_BLOCK, NORMAL);
  memcpy( (char *) sp, bp->b_data, (size_t) SUPER_SIZE);
  put_block(bp, ZUPER_BLOCK);
  sp->s_dev = NO_DEV;                           /* restore later */
  magic = sp->s_magic;                          /* determines file system type */

  /* Get file system version and type. */

  if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
                version = V1;
                native  = (magic == SUPER_MAGIC);
                printf("Version v1 FS");
  } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
                version = V2;
                native  = (magic == SUPER_V2);
                printf("Version V2");
  } else if(magic==SUPER_V3 || magic == conv2(BYTE_SWAP, SUPER_V2)){
       version = V3;
       native =  (magic == SUPER_V3);
       printf("Version MyFS");
  }else{
       return (EINVAL);
  }

  /* If the super block has the wrong byte order, swap the fields; the magic
   * number doesn't need conversion. */
  sp->s_ninodes         =               conv2(native, (int) sp->s_ninodes);
  sp->s_nzones          =               conv2(native, (int) sp->s_nzones);
  sp->s_imap_blocks     =               conv2(native, (int) sp->s_imap_blocks);
  sp->s_zmap_blocks     =               conv2(native, (int) sp->s_zmap_blocks);
  sp->s_firstdatazone   =               conv2(native, (int) sp->s_firstdatazone);
  sp->s_log_zone_size =                 conv2(native, (int) sp->s_log_zone_size);
  sp->s_max_size        =               conv4(native, sp->s_max_size);
  sp->s_zones           =               conv4(native, sp->s_zones);
  if(version == V3){                    /*MyFS file system*/
                sp->s_last_check =    conv4(native, sp->s_last_check);
                sp->s_checkintervals = conv4(native, sp->s_checkintervals);
                sp->s_state  = conv2(native, sp->s_state);
                sp->s_mtime = conv4(native, sp->s_mtime);
                sp->s_mnt_count = conv2(native,sp->s_mnt_count);
                sp->s_inodes_count= conv4(native, sp->s_inodes_count);
  }
  if (version == V1) {
                sp->s_zones = sp->s_nzones;       /* only V1 needs this copy */
                sp->s_inodes_per_block = V1_INODES_PER_BLOCK;
                sp->s_ndzones = V1_NR_DZONES;
                sp->s_nindirs = V1_INDIRECTS;
  } else if(version == V2){
```

```
                sp->s_inodes_per_block = V2_INODES_PER_BLOCK;
                sp->s_ndzones = V2_NR_DZONES;
                sp->s_nindirs = V2_INDIRECTS;
        }else{
                sp->s_inodes_per_block = V3_INODES_PER_BLOCK;
                sp->s_ndzones = V3_NR_DZONES;
                sp->s_nindirs = V3_INDIRECTS;
        }
        sp->s_isearch = 0;              /* inode searches initially start at 0 */
        sp->s_zsearch = 0;              /* zone searches initially start at 0 */
        sp->s_version = version;
        sp->s_native  = native;

        /* Make a few basic checks to see if super block looks reasonable. */
        if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1 || sp->s_ninodes < 1 || sp->s_zones < 1
                                                || (unsigned) sp->s_log_zone_size > 4) {
                return(EINVAL);
        }
        sp->s_dev = dev;                /* restore device number */
        return(OK);
}
```

read.c

```
/* This file contains the heart of the mechanism used to read (and write) files.  Read and write requests are split up
into chunks that do not cross block boundaries.  Each chunk is then processed in turn.  Reads on special files are also
detected and handled. */
#include "fs.h"
#include <fcntl.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

#define FD_MASK         077             /* max file descriptor is 63 */
PRIVATE message umess;                  /* message for asking SYSTASK for user copy */

FORWARD _PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position, unsigned off, int chunk, unsigned left, int rw_flag,
                                char *buff, int seg, int usr)                                   );

PUBLIC int do_read(){
        return(read_write(READING));
}

PUBLIC int read_write(rw_flag)
int rw_flag;                            /* READING or WRITING */
{       /* Perform read(fd, buffer, nbytes) or write(fd, buffer, nbytes) call. */

  register struct inode *rip;
  register struct filp *f;
  off_t bytes_left, f_size, position;
  unsigned int off, cum_io;
  int op, oflags, r, chunk, usr, seg, block_spec, char_spec;
  int regular, partial_pipe = 0, partial_cnt = 0;
  dev_t dev;
  mode_t mode_word;
  struct filp *wf;

  /* MM loads segments by putting funny things in upper 10 bits of 'fd'. */
  if (who == MM_PROC_NR && (fd & (~BYTE)) ) {
        usr = (fd >> 8) & BYTE;
        seg = (fd >> 6) & 03;
        fd &= FD_MASK;                  /* get rid of user and segment bits */
  } else {
        usr = who;                      /* normal case */
        seg = D;
```

```
}

/* If the file descriptor is valid, get the inode, size and mode. */
if (nbytes < 0) return(EINVAL);
if ((f = get_filp(fd)) == NIL_FILP) return(err_code);
if (((f->filp_mode) & (rw_flag == READING ? R_BIT : W_BIT)) == 0) {
        return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
}
if (nbytes == 0) return(0);          /* so char special files need not check for 0*/
position = f->filp_pos;
if (position > MAX_FILE_POS) return(EINVAL);
if (position + nbytes < position) return(EINVAL); /* unsigned overflow */
oflags = f->filp_flags;
rip = f->filp_ino;
f_size = rip->i_size;
r = OK;
if (rip->i_pipe == I_PIPE) {
        /* fp->fp_cum_io_partial is only nonzero when doing partial writes */
        cum_io = fp->fp_cum_io_partial;
} else {
        cum_io = 0;
}
op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
mode_word = rip->i_mode & I_TYPE;
regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;

char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0);
block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0);
if (block_spec) f_size = LONG_MAX;
rdwt_err = OK;                  /* set to EIO if disk error occurs */

/* Check for character special files. */
if (char_spec) {
        dev = (dev_t) rip->i_zone[0];
        r = dev_io(op, oflags & O_NONBLOCK, dev, position, nbytes, who,buffer);
        if (r >= 0) {
                cum_io = r;
                position += r;
                r = OK;
        }
} else {
        if (rw_flag == WRITING && block_spec == 0) {
                /* Check in advance to see if file will grow too big. */
                if (position > rip->i_sp->s_max_size - nbytes) return(EFBIG);

                /* Check for O_APPEND flag. */
                if (oflags & O_APPEND) position = f_size;
        /* Clear the zone containing present EOF if hole about
                 * to be created.  This is necessary because all unwritten
                 * blocks prior to the EOF must read as zeros.
                 */
                if (position > f_size) clear_zone(rip, f_size, 0);
        }

        /* Pipes are a little different.  Check. */
        if (rip->i_pipe == I_PIPE) {
             r = pipe_check(rip,rw_flag,oflags,nbytes,position,&partial_cnt);
             if (r <= 0) return(r);
        }

        if (partial_cnt > 0) partial_pipe = 1;

        /* Split the transfer into chunks that don't span two blocks. */
        while (nbytes != 0) {
                off = (unsigned int) (position % BLOCK_SIZE);/* offset in blk*/
                if (partial_pipe) {  /* pipes only */
                        chunk = MIN(partial_cnt, BLOCK_SIZE - off);
                } else
                        chunk = MIN(nbytes, BLOCK_SIZE - off);
                if (chunk < 0) chunk = BLOCK_SIZE - off;
```

```
                        if (rw_flag == READING) {
                                bytes_left = f_size - position;
                                if (position >= f_size) break;        /* we are beyond EOF */
                                if (chunk > bytes_left) chunk = (int) bytes_left;
                        }

                        /* Read or write 'chunk' bytes. */
                        r = rw_chunk(rip, position, off, chunk, (unsigned) nbytes,
                                        rw_flag, buffer, seg, usr);
                        if (r != OK) break;     /* EOF reached */
                        if (rdwt_err < 0) break;

                        /* Update counters and pointers. */
                        buffer += chunk;        /* user buffer address */
                        nbytes -= chunk;        /* bytes yet to be read */
                        cum_io += chunk;        /* bytes read so far */
                        position += chunk;      /* position within the file */

                        if (partial_pipe) {
                                partial_cnt -= chunk;
                                if (partial_cnt <= 0)  break;
                        }
                }
        }

        /* On write, update file size and access time. */
        if (rw_flag == WRITING) {
                if (regular || mode_word == I_DIRECTORY) {
                        if (position > f_size) rip->i_size = position;
                }
        } else {
                if (rip->i_pipe == I_PIPE && position >= rip->i_size) {
                        /* Reset pipe pointers. */
                        rip->i_size = 0;                        /* no data left */
                        position = 0;                           /* reset reader(s) */
                        if ( (wf = find_filp(rip, W_BIT)) != NIL_FILP) wf->filp_pos =0;
                }
        }
        f->filp_pos = position;

        /* Check to see if read-ahead is called for, and if so, set it up. */
        if (rw_flag == READING && rip->i_seek == NO_SEEK && position % BLOCK_SIZE== 0
                        && (regular || mode_word == I_DIRECTORY)) {
                rdahed_inode = rip;
                rdahedpos = position;
        }
        rip->i_seek = NO_SEEK;

        if (rdwt_err != OK) r = rdwt_err;               /* check for disk error */
        if (rdwt_err == END_OF_FILE) r = OK;
        if (r == OK) {
                if (rw_flag == READING) rip->i_update |= ATIME;
                if (rw_flag == WRITING) rip->i_update |= CTIME | MTIME;
                rip->i_dirt = DIRTY;                    /* inode is thus now dirty */
                if (partial_pipe) {
                        partial_pipe = 0;
                                /* partial write on pipe with */
                                /* O_NONBLOCK, return write count */
                        if (!(oflags & O_NONBLOCK)) {
                                fp->fp_cum_io_partial = cum_io;
                                suspend(XPIPE); /* partial write on pipe with */
                                return(0);   /* nbyte > PIPE_SIZE - non-atomic */
                        }
                }
                fp->fp_cum_io_partial = 0;
                return(cum_io);
        } else {
                return(r);
        }
```

```
}

PRIVATE int rw_chunk(rip, position, off, chunk, left, rw_flag, buff, seg, usr)
register struct inode *rip;                     /* pointer to inode for file to be rd/wr */
off_t position;                                 /* position within file to read or write */
unsigned off;                                   /* off within the current block */
int chunk;                                      /* number of bytes to read or write */
unsigned left;                                  /* max number of bytes wanted after position */
int rw_flag;                                    /* READING or WRITING */
char *buff;                                     /* virtual address of the user buffer */
int seg;                                        /* T or D segment in user space */
int usr;                                        /* which user process */
{    /* Read or write (part of) a block. */

 register struct buf *bp;
 register int r;
 int n, block_spec;
 block_t b;
 dev_t dev;

 block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
 if (block_spec) {
        b = position/BLOCK_SIZE;
        dev = (dev_t) rip->i_zone[0];
 } else {
        b = read_map(rip, position);
        dev = rip->i_dev;
 }

 if (!block_spec && b == NO_BLOCK) {
        if (rw_flag == READING) {
                /* Reading from a nonexistent block.  Must read as all zeros.*/
                bp = get_block(NO_DEV, NO_BLOCK, NORMAL);    /* get a buffer */
                zero_block(bp);
        } else {
                /* Writing to a nonexistent block. Create and enter in inode.*/
                if ((bp= new_block(rip, position)) == NIL_BUF)return(err_code);
        }
 } else if (rw_flag == READING) {
        /* Read and read ahead if convenient. */
        bp = rahead(rip, b, position, left);
} else {
        n = (chunk == BLOCK_SIZE ? NO_READ : NORMAL);
        if (!block_spec && off == 0 && position >= rip->i_size) n = NO_READ;
        bp = get_block(dev, b, n);
 }

 /* In all cases, bp now points to a valid buffer. */
 if (rw_flag == WRITING && chunk != BLOCK_SIZE && !block_spec && position >= rip->i_size && off == 0) {
        zero_block(bp);
 }
 if (rw_flag == READING) {
        /* Copy a chunk from the block buffer to user space. */
r = sys_copy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off), usr, seg, (phys_bytes) buff, (phys_bytes) chunk);
 } else {
        /* Copy a chunk from user space to the block buffer. */
r = sys_copy(usr, seg, (phys_bytes) buff, FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
                                        (phys_bytes) chunk);
        bp->b_dirt = DIRTY;
 }
 n = (off + chunk == BLOCK_SIZE ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
 put_block(bp, n);
 return(r);
}


PUBLIC block_t read_map(rip, position)
register struct inode *rip;             /* ptr to inode to map from */
off_t position;                                 /* position in file whose blk wanted */
{
```

```
register struct buf *bp;
register zone_t z,z2,z1;
int scale, boff, dzones, nr_indirects;
block_t b;
unsigned long excess, zone, block_pos, nr_dblindirects, index,ind_ex,zind,ex;

scale = rip->i_sp->s_log_zone_size;           /* for block-zone conversion */
block_pos = position/BLOCK_SIZE;          /* relative blk # in file */
zone = block_pos >> scale;        /* position's zone */
boff = (int) (block_pos - (zone << scale) ); /* relative blk # within zone */
dzones = rip->i_ndzones;
nr_indirects = rip->i_nindirs;
nr_dblindirects = (unsigned long) nr_indirects * nr_indirects;

/* Is 'position' to be found in the inode itself? */
if (zone < dzones) {
        zind = (int) zone;        /* index should be an int */
        z = rip->i_zone[zind];
        if (z == NO_ZONE) return(NO_BLOCK);
        b = ((block_t) z << scale) + boff;
        return(b);
}

/* It is not in the inode, so it must be single or double indirect. */
excess = zone - dzones;             /* first Vx_NR_DZONES don't count */

if (excess < nr_indirects) {
        /* 'position' can be located via the single indirect block. */
        z = rip->i_zone[dzones];
} else if( excess >= nr_indirects && excess < nr_dblindirects){
        /* 'position' can be located via the double indirect block. */
        if ( (z = rip->i_zone[dzones+1]) == NO_ZONE) return(NO_BLOCK);
        excess -= nr_indirects;                         /* single indir doesn't count*/
        b = (block_t) z << scale;
        bp = get_block(rip->i_dev, b, NORMAL);    /* get double indirect block */
        index = (int) (excess/nr_indirects);
        z = rd_indir(bp, index);                        /* z= zone for single*/
        put_block(bp, INDIRECT_BLOCK);          /* release double ind block */
        excess = excess % nr_indirects;                 /* index into single ind blk */
}
else{
  if(rip->i_sp->s_version != V3 ) {
      printf("Not MyFS File System");
      return (EFBIG);
   }
 if((z2 = rip->i_zone[dzones+2]) == NO_ZONE)     return (err_code);
  excess -= nr_indirects;
  excess -= nr_dblindirects;
  index = (int) excess / nr_dblindirects;
  excess = excess % nr_dblindirects;
  b = (block_t) z2 <<scale;
  bp = get_block(rip->i_dev,b , NORMAL);
  z1 = rd_indir(bp, index);
  put_block(bp, INDIRECT_BLOCK);
  if(z1== NO_ZONE) return (NO_BLOCK);
  /* Now we hav the z1 as the doble indirect block */
  ind_ex = (int) excess / nr_indirects;
  excess %= nr_indirects;
  b = (block_t) z1 << scale;
  bp= get_block(rip->i_dev, b, NORMAL);
  z= rd_indir(bp, ind_ex);
}
/* 'z' is zone num for single indirect block; 'excess' is index into it. */
if (z == NO_ZONE) return(NO_BLOCK);
b = (block_t) z << scale;                               /* b is blk # for single ind */
bp = get_block(rip->i_dev, b, NORMAL);  /* get single indirect block */
ex = (int) excess;                                      /* need an integer */
z = rd_indir(bp, ex);                                   /* get block pointed to */
put_block(bp, INDIRECT_BLOCK);                          /* release single indir blk */
if (z == NO_ZONE) {
```

```
  return(NO_BLOCK);
 }
 b = ((block_t) z << scale) + boff;
return(b);
}

PUBLIC zone_t rd_indir(bp, index)
struct buf *bp;                         /* pointer to indirect block */
int index;                      /* index into *bp */
{ /* Given a pointer to an indirect block, read one entry.  */

 struct super_block *sp;
 zone_t zone;                           /* V2 zones are longs (shorts in V1) */

 sp = get_super(bp->b_dev);      /* need super block to find file sys type */

 /* read a zone from an indirect block */
 if (sp->s_version == V1)
        zone = (zone_t) conv2(sp->s_native, (int)  bp->b_v1_ind[index]);
 else if(sp->s_version == V3)
     zone = (zone_t) conv4(sp->s_native, (long) bp->b_v3_ind[index]);
 else
        zone = (zone_t) conv4(sp->s_native, (long) bp->b_v2_ind[index]);

 if (zone != NO_ZONE && (zone < (zone_t) sp->s_firstdatazone || zone >= sp->s_zones)) {
        printf("Illegal zone number %ld in indirect block, index %d\n", (long) zone, index);
        panic("check file system", NO_NUM);
 }
 return(zone);
}

PUBLIC void read_ahead()
{ /* Read a block into the cache before it is needed. */

 register struct inode *rip;
 struct buf *bp;
 block_t b;

 rip = rdahed_inode;            /* pointer to inode to read ahead from */
 rdahed_inode = NIL_INODE;      /* turn off read ahead */
 if ( (b = read_map(rip, rdahedpos)) == NO_BLOCK) return;        /* at EOF */
 bp = rahead(rip, b, rdahedpos, BLOCK_SIZE);
 put_block(bp, PARTIAL_DATA_BLOCK);
}

PUBLIC struct buf *rahead(rip, baseblock, position, bytes_ahead)
register struct inode *rip;             /* pointer to inode for file to be read */
block_t baseblock;                      /* block at current position */
off_t position;                         /* position within file */
unsigned bytes_ahead;                   /* bytes beyond position for immediate use */
{ /* Fetch a block from the cache or the device.  If a physical read is required, prefetch as many more blocks as
   convenient into the cache. */

/* Minimum number of blocks to prefetch. */
# define BLOCKS_MINIMUM                 (NR_BUFS < 50 ? 18 : 32)

 int block_spec, scale, read_q_size;
 unsigned int blocks_ahead, fragment;
 block_t block, blocks_left;
 off_t ind1_pos;
 dev_t dev;
 struct buf *bp;
 static struct buf *read_q[NR_BUFS];

 block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
 if (block_spec) {
        dev = (dev_t) rip->i_zone[0];
 } else {
        dev = rip->i_dev;
 }
```

```
 block = baseblock;
 bp = get_block(dev, block, PREFETCH);
 if (bp->b_dev != NO_DEV) return(bp);

fragment = position % BLOCK_SIZE;
 position -= fragment;
 bytes_ahead += fragment;

 blocks_ahead = (bytes_ahead + BLOCK_SIZE - 1) / BLOCK_SIZE;

 if (block_spec && rip->i_size == 0) {
        blocks_left = NR_IOREQS;
 } else {
        blocks_left = (rip->i_size - position + BLOCK_SIZE - 1) / BLOCK_SIZE;
        /* Go for the first indirect block if we are in its neighborhood. */
        if (!block_spec) {
                scale = rip->i_sp->s_log_zone_size;
                ind1_pos = (off_t) rip->i_ndzones * (BLOCK_SIZE << scale);
                if (position <= ind1_pos && rip->i_size > ind1_pos) {
                        blocks_ahead++;
                        blocks_left++;
                }
        }
 }

 /* No more than the maximum request. */
 if (blocks_ahead > NR_IOREQS) blocks_ahead = NR_IOREQS;

 /* Read at least the minimum number of blocks, but not after a seek. */
 if (blocks_ahead < BLOCKS_MINIMUM && rip->i_seek == NO_SEEK)
        blocks_ahead = BLOCKS_MINIMUM;

 /* Can't go past end of file. */
 if (blocks_ahead > blocks_left) blocks_ahead = blocks_left;

 read_q_size = 0;

 /* Acquire block buffers. */
 for (;;) {
        read_q[read_q_size++] = bp;
        if (--blocks_ahead == 0) break;
        /* Don't trash the cache, leave 4 free. */
        if (bufs_in_use >= NR_BUFS - 4) break;
        block++;
        bp = get_block(dev, block, PREFETCH);
        if (bp->b_dev != NO_DEV) {
                /* Oops, block already in the cache, get out. */
                put_block(bp, FULL_DATA_BLOCK);
                break;
        }
 }
 rw_scattered(dev, read_q, read_q_size, READING);
 return(get_block(dev, baseblock, NORMAL));
}
```

write.c

```
/* This file is the counterpart of "read.c".  It contains the code for writing in sofar as this is not contained in read_write(). */

#include "fs.h"
#include <string.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

FORWARD _PROTOTYPE( int write_map, (struct inode *rip, off_t position, zone_t new_zone)       );
```

```
FORWARD _PROTOTYPE( void wr_indir, (struct buf *bp, int index, zone_t zone) );

PUBLIC int do_write()		{ /* Perform the write(fd, buffer, nbytes) system call. */
	return(read_write(WRITING));
}

PRIVATE int write_map(rip, position, new_zone)
register struct inode *rip;		/* pointer to inode to be changed */
off_t position;				/* file address to be mapped */
zone_t new_zone;			/* zone # to be inserted */
{ /* Write a new zone into an inode. */
  int new_trp , doubl;
  int scale, new_ind, new_dbl, zones, nr_indirects, single;
  zone_t z, z1;
  zone_t z2;
  register block_t b;
  unsigned long excess, zone, nr_dblindirects,ind_ex , zindex, ex;
  struct buf *bp;

  rip->i_dirt = DIRTY;			/* inode will be changed */
  bp = NIL_BUF;
  scale = rip->i_sp->s_log_zone_size;		/* for zone-block conversion */
  zone = (position/BLOCK_SIZE) >> scale;  /* relative zone # to insert */
  zones = rip->i_ndzones;		/* # direct zones in the inode */
  nr_indirects = rip->i_nindirs;/* # indirect zones per indirect block */

  /* Is 'position' to be found in the inode itself? */
  if (zone < zones) {
	zindex = (int) zone;	/* we need an integer here */
	rip->i_zone[zindex] = new_zone;
	return(OK);
  }

  /* It is not in the inode, so it must be single or double indirect. */
  excess = (unsigned long) zone - zones;	/* first Vx_NR_DZONES don't count */
  new_ind = FALSE;
  new_dbl = FALSE;
  new_trp = FALSE;
  nr_dblindirects = (unsigned long )nr_indirects * nr_indirects ;

  if (excess < nr_indirects) {
	/* 'position' can be located via the single indirect block. */
	z1 = rip->i_zone[zones];		/* single indirect zone */
	single = TRUE;
   }else{ if(excess >= nr_indirects && excess < (unsigned long) nr_dblindirects)
     {
	/* 'position' can be located via the double indirect block. */
	if ( (z = rip->i_zone[zones+1]) == NO_ZONE) {
		/* Create the double indirect block. */
		if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0])) == NO_ZONE)
			return(err_code);
		rip->i_zone[zones+1] = z;
		new_dbl = TRUE;	/* set flag for later */
	}

	/* Either way, 'z' is zone number for double indirect block. */
	excess -= (unsigned long) nr_indirects;		/* single indirect doesn't count */
	ind_ex = (int) (excess / nr_indirects);
	excess = (unsigned long) (excess % nr_indirects);
	if (ind_ex >= nr_indirects) return(EFBIG);
	b = (block_t) z << scale;
	bp = get_block(rip->i_dev, b, (new_dbl ? NO_READ : NORMAL));
	if (new_dbl) zero_block(bp);
	z1 = rd_indir(bp, ind_ex);
	single = FALSE;
  }
  else {
     /*position' can be located via the triple indirect block */
   if(rip->i_sp->s_version != V3 ) {
        printf("Not MyFS File System");
```

```
        return (EFBIG);
    }

     if((z2=rip->i_zone[zones+2]) == NO_ZONE )
     {
      /*create a triple indirect block */
      if((z2 = alloc_zone(rip->i_dev,rip->i_zone[0])) == NO_ZONE)
          return (err_code);
      rip->i_zone[zones+2] = z2;
      new_trp = TRUE;
     }
     /* Now we have triple in z2 .Remove counts of single indirect block */
     excess -= nr_indirects;
     excess -= nr_dblindirects;
     ind_ex = (int) excess / nr_dblindirects;
     excess %= nr_dblindirects;

     if(ind_ex >= nr_indirects)
        return (EFBIG);
     b= (block_t) z2 << scale;
     bp = get_block(rip->i_dev, b , (new_trp ? NO_READ : NORMAL));
     if (new_trp)  zero_block(bp);
     z = rd_indir(bp,ind_ex);
     single = FALSE;
     doubl = FALSE;

    /* Check the double indirect block */
     if(z == NO_ZONE){
             new_dbl= TRUE;
             /* Create the double indirect block  */
             if((z= alloc_zone(rip->i_dev,rip->i_zone[0])) == NO_ZONE)
                     return (err_code);
        /* Now update entry in triple indirect block */
        wr_indir(bp, ind_ex , z);
        if(bp!=NIL_BUF)  bp->b_dirt = DIRTY;
        if(z == NO_ZONE)          return (err_code);
    }
       put_block(bp,INDIRECT_BLOCK);
       /* Now we have z as double zone  */
       ind_ex = (int)( excess/nr_indirects);
       excess = excess % nr_indirects;
       if(ind_ex >= nr_indirects)   return (EFBIG);
       b = (block_t) z <<scale;
       bp = get_block(rip->i_dev, b , (new_dbl ? NO_READ : NORMAL));
       if(new_dbl)      zero_block(bp);
       z1= rd_indir(bp,ind_ex);
       single =  FALSE;
}
}
/* z1 is now single indirect zone; 'excess' is index. */
if (z1 == NO_ZONE) {
             /* Create indirect block and store zone # in inode or dbl indir blk. */
             z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);
             if(single)
                     rip->i_zone[zones] = z1;             /* update inode */
             else
                     wr_indir(bp, ind_ex, z1);            /* update dbl indir */
             new_ind = TRUE;
             if(bp != NIL_BUF) bp->b_dirt = DIRTY;     /* if double ind, it is dirty*/
             if(z1 == NO_ZONE) {
                     put_block(bp, INDIRECT_BLOCK);          /* release dbl indirect blk */
                     return(err_code);        /* couldn't create single ind */
             }
}
put_block(bp, INDIRECT_BLOCK);          /* release double indirect blk */

/* z1 is indirect block's zone number. */
b = (block_t) z1 << scale;
bp = get_block(rip->i_dev, b, (new_ind ? NO_READ : NORMAL) );
if (new_ind) zero_block(bp);
```

```
  ex = (int) excess;                    /* we need an int here */
  wr_indir(bp, ex, new_zone);
  bp->b_dirt = DIRTY;
  put_block(bp, INDIRECT_BLOCK);
 /* printf("End wmap");   */
  return(OK);
}

PRIVATE void wr_indir(bp, index, zone)
struct buf *bp;                         /* pointer to indirect block */
int index;                      /* index into *bp */
zone_t zone;                            /* zone to write */
{ /* Given a pointer to an indirect block, write one entry. */

  struct super_block *sp;
  sp = get_super(bp->b_dev);        /* need super block to find file sys type */

  /* write a zone into an indirect block */
  if (sp->s_version == V1)
              bp->b_v1_ind[index] = (zone1_t) conv2(sp->s_native, (int)  zone);
  else if (sp->s_version == V3)
        bp->b_v3_ind[index] = (zone_t)  conv4(sp->s_native, (long) zone);
  else
              bp->b_v2_ind[index] = (zone_t)  conv4(sp->s_native, (long) zone);
}

PUBLIC void clear_zone(rip, pos, flag)

register struct inode *rip;             /* inode to clear */
off_t pos;                      /* points to block to clear */
int flag;                       /* 0 if called by read_write, 1 by new_block */
{ /* Zero a zone, possibly starting in the middle.  The parameter 'pos' gives a byte in the first block to be zeroed.
Clearzone() is called from  read_write and new_block().*/

  register struct buf *bp;
  register block_t b, blo, bhi;
  register off_t next;
  register int scale;
  register zone_t zone_size;

  /* If the block size and zone size are the same, clear_zone() not needed. */
  scale = rip->i_sp->s_log_zone_size;
  if (scale == 0) return;

  zone_size = (zone_t) BLOCK_SIZE << scale;
  if (flag == 1) pos = (pos/zone_size) * zone_size;
  next = pos + BLOCK_SIZE - 1;

  /* If 'pos' is in the last block of a zone, do not clear the zone. */
  if (next/zone_size != pos/zone_size) return;
  if ( (blo = read_map(rip, next)) == NO_BLOCK) return;
  bhi = (  ((blo>>scale)+1) << scale)   - 1;

  /* Clear all the blocks between 'blo' and 'bhi'. */
  for (b = blo; b <= bhi; b++) {
              bp = get_block(rip->i_dev, b, NO_READ);
              zero_block(bp);
              put_block(bp, FULL_DATA_BLOCK);
  }
}


PUBLIC struct buf *new_block(rip, position)
register struct inode *rip;             /* pointer to inode */
off_t position;                         /* file pointer */
{   /* Acquire a new block and return a pointer to it.  Doing so may require allocating a complete zone, and then returning the initial block. On the
other hand, the current zone may still have some unused blocks.*/

  register struct buf *bp;
  block_t b, base_block;
```

```
  zone_t z;
  zone_t zone_size;
  int scale, r;
  struct super_block *sp;

  /* Is another block available in the current zone? */
  if ( (b = read_map(rip, position)) == NO_BLOCK) {
                /* Choose first zone if possible. */
                /* Lose if the file is nonempty but the first zone number is NO_ZONE
                 * corresponding to a zone full of zeros.  It would be better to
                 * search near the last real zone.
                 */
                if (rip->i_zone[0] == NO_ZONE) {
                        sp = rip->i_sp;
                        z = sp->s_firstdatazone;
                } else {
                        z = rip->i_zone[0];     /* hunt near first zone */
                }
                if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);
                if ( (r = write_map(rip, position, z)) != OK) {
                        free_zone(rip->i_dev, z);
                        err_code = r;
                        return(NIL_BUF);
                }

                /* If we are not writing at EOF, clear the zone, just to be safe. */
                if ( position != rip->i_size)
                clear_zone(rip, position, 1);

                scale = rip->i_sp->s_log_zone_size;
                base_block = (block_t) z << scale;
                zone_size = (zone_t) BLOCK_SIZE << scale;
                b = base_block + (block_t)((position % zone_size)/BLOCK_SIZE);
  }

  bp = get_block(rip->i_dev, b, NO_READ);
  zero_block(bp);
  return(bp);
}


PUBLIC void zero_block(bp)
register struct buf *bp;                /* pointer to buffer to zero */
{ /* Zero a block. */

  memset(bp->b_data, 0, BLOCK_SIZE);
  bp->b_dirt = DIRTY;
}
```

open.c

```
/* This file contains the procedures for creating, opening, closing, and
 * seeking on files. The entry points into this file are
 *   do_creat:          perform the CREAT system call
 *   do_open:           perform the OPEN system call
 *   do_mknod:          perform the MKNOD system call
 *   do_mkdir:          perform the MKDIR system call
 *   do_close:          perform the CLOSE system call
 *   do_lseek:  perform the LSEEK system call */

#include "fs.h"
#include <sys/stat.h>
#include <fcntl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
```

```
#include "inode.h"
#include "lock.h"
#include "param.h"
#include "super.h"

PRIVATE message dev_mess;
PRIVATE char mode_map[] = {R_BIT, W_BIT, R_BIT|W_BIT, 0};

FORWARD _PROTOTYPE( int common_open, (int oflags, Mode_t omode)                  );
FORWARD _PROTOTYPE( int pipe_open, (struct inode *rip,Mode_t bits,int oflags)    );
FORWARD _PROTOTYPE( struct inode *new_node, (char *path, Mode_t bits, zone_t z0)          );

PUBLIC int do_creat()
{ /* Perform the creat(name, mode) system call. */
 int r;

 if (fetch_name(name, name_length, M3) != OK) return(err_code);
 r = common_open(O_WRONLY | O_CREAT | O_TRUNC, (mode_t) mode);
 return(r);
}

PUBLIC int do_open()
{ /* Perform the open(name, flags,...) system call. */

 int create_mode = 0;                   /* is really mode_t but this gives problems */
 int r;

 /* If O_CREAT is set, open has three parameters, otherwise two. */
 if (mode & O_CREAT) {
            create_mode = c_mode;
            r = fetch_name(c_name, name1_length, M1);
 } else {
            r = fetch_name(name, name_length, M3);
 }

 if (r != OK) return(err_code); /* name was bad */
 r = common_open(mode, create_mode);
 return(r);
}

PRIVATE int common_open(oflags, omode)
register int oflags;
mode_t omode;
{ /* Common code from do_creat and do_open. */

 register struct inode *rip;
 int r, b, major, task, exist = TRUE;
 dev_t dev;
 mode_t bits;
 off_t pos;
 struct filp *fil_ptr, *filp2;

 /* Remap the bottom two bits of oflags. */
 bits = (mode_t) mode_map[oflags & O_ACCMODE];

 /* See if file descriptor and filp slots are available. */
 if ( (r = get_fd(0, bits, &fd, &fil_ptr)) != OK) return(r);

 /* If O_CREATE is set, try to make the file. */
 if (oflags & O_CREAT) {
            /* Create a new inode by calling new_node(). */
     omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
            rip = new_node(user_path, omode, NO_ZONE);
            r = err_code;
            if (r == OK) exist = FALSE;      /* we just created the file */
            else if (r != EEXIST) return(r); /* other error */
            else exist = !(oflags & O_EXCL); /* file exists, if the O_EXCL
                                                        flag is set this is an error */
 } else {
             /* Scan path name. */
```

```
        if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
}

/* Claim the file descriptor and filp slot and fill them in. */
fp->fp_filp[fd] = fil_ptr;
fil_ptr->filp_count = 1;
fil_ptr->filp_ino = rip;
fil_ptr->filp_flags = oflags;

/* Only do the normal open code if we didn't just create the file. */
if (exist) {
        /* Check protections. */
        if ((r = forbidden(rip, bits)) == OK) {
            if(rip->i_sp->s_version == V3 && IS_APPEND(rip)){
                if(~(oflags & O_APPEND)){
                    printk("File is append only");
                    return (EPERM);
                }
            }
                /* Opening reg. files directories and special files differ. */
                switch (rip->i_mode & I_TYPE) {
                  case I_REGULAR:
                        /* Truncate regular file if O_TRUNC. */
                        if (oflags & O_TRUNC) {
                                if ((r = forbidden(rip, W_BIT)) !=OK) break;
                                truncate(rip);
                                wipe_inode(rip);
                                /* Send the inode from the inode cache to the
                                 * block cache, so it gets written on the next
                                 * cache flush.
                                 */
                                rw_inode(rip, WRITING);
                        }
                        break;

                  case I_DIRECTORY:
                        /* Directories may be read but not written. */
                        r = (bits & W_BIT ? EISDIR : OK);
                        break;

                  case I_CHAR_SPECIAL:
                  case I_BLOCK_SPECIAL:
                        /* Invoke the driver for special processing. */
                        dev_mess.m_type = DEV_OPEN;
                        dev = (dev_t) rip->i_zone[0];
                        dev_mess.DEVICE = dev;
                        dev_mess.COUNT = bits | (oflags & ~O_ACCMODE);
                        major = (dev >> MAJOR) & BYTE;          /* major device nr */
                        if (major <= 0 || major >= max_major) {
                                r = ENODEV;
                                break;
                        }
                        task = dmap[major].dmap_task;   /* device task nr */
                        (*dmap[major].dmap_open)(task, &dev_mess);
                        r = dev_mess.REP_STATUS;
                        break;

                  case I_NAMED_PIPE:
                        oflags |= O_APPEND;             /* force append mode */
                        fil_ptr->filp_flags = oflags;
                        r = pipe_open(rip, bits, oflags);
                        if (r == OK) {
                                /* See if someone else is doing a rd or wt on
                                 * the FIFO.  If so, use its filp entry so the
                                 * file position will be automatically shared.
                                 */
                                b = (bits & R_BIT ? R_BIT : W_BIT);
                                fil_ptr->filp_count = 0; /* don't find self */
                                if ((filp2 = find_filp(rip, b)) != NIL_FILP) {
                                        /* Co-reader or writer found. Use it.*/
```

```
                                        fp->fp_filp[fd] = filp2;
                                        filp2->filp_count++;
                                        filp2->filp_ino = rip;
                                        filp2->filp_flags = oflags;

                                        /* i_count was incremented incorrectly
                                         * by eatpath above, not knowing that
                                         * we were going to use an existing
                                         * filp entry.  Correct this error.
                                         */
                                        rip->i_count--;
                                } else {
                                        /* Nobody else found.  Restore filp. */
                                        fil_ptr->filp_count = 1;
                                        if (b == R_BIT)
                                            pos = rip->i_zone[V2_NR_DZONES+1];
                                        else
                                            pos = rip->i_zone[V2_NR_DZONES+2];
                                        fil_ptr->filp_pos = pos;
                                }
                        }
                        break;
                }
        }
 }

 /* If error, release inode. */
 if (r != OK) {
        fp->fp_filp[fd] = NIL_FILP;
        fil_ptr->filp_count= 0;
        put_inode(rip);
        return(r);
 }

 return(fd);
}

PRIVATE struct inode *new_node(path, bits, z0)
char *path;                     /* pointer to path name */
mode_t bits;                            /* mode of the new inode */
zone_t z0;                      /* zone number 0 for new inode */
{ /* New_node() is called by common_open(), do_mknod(), and do_mkdir().  In all cases it allocates a new inode,
   makes a directory entry for it on the path 'path', and initializes it.  It returns a pointer to the inode if it can do
this; otherwise it returns NIL_INODE.  It always sets 'err_code'  to an appropriate value (OK or an error code).*/

 register struct inode *rlast_dir_ptr, *rip;
 register int r;
 char string[NAME_MAX];

 /* See if the path can be opened down to the last directory. */
 if ((rlast_dir_ptr = last_dir(path, string)) == NIL_INODE) return(NIL_INODE);
 if(rlast_dir_ptr->i_sp->s_version==V3 && IS_IMMUTABLE(rlast_dir_ptr)){
     printk("Parent Directory is Immutable");
     return (NIL_INODE);
 }
 /* The final directory is accessible. Get final component of the path. */
 rip = advance(rlast_dir_ptr, string);
 if ( rip == NIL_INODE && err_code == ENOENT) {
        /* Last path component does not exist.  Make new directory entry. */
        if ( (rip = alloc_inode(rlast_dir_ptr->i_dev, bits)) == NIL_INODE) {
                /* Can't creat new inode: out of inodes. */
                put_inode(rlast_dir_ptr);
                return(NIL_INODE);
        }

/* Force inode to the disk before making directory entry to make the system more robust in the face of a crash:
an inode with no directory entry is much better than the opposite. */
        rip->i_nlinks++;
        rip->i_zone[0] = z0;            /* major/minor device numbers */
        rw_inode(rip, WRITING);                 /* force inode to disk now */
```

```
		/* New inode acquired.  Try to make directory entry. */
		if ((r = search_dir(rlast_dir_ptr, string, &rip->i_num,ENTER)) != OK) {
			put_inode(rlast_dir_ptr);
			rip->i_nlinks--;	/* pity, have to free disk inode */
			rip->i_dirt = DIRTY;	/* dirty inodes are written out */
			put_inode(rip);	/* this call frees the inode */
			err_code = r;
			return(NIL_INODE);
		}

	} else {
		/* Either last component exists, or there is some problem. */
		if (rip != NIL_INODE)
			r = EEXIST;
		else
			r = err_code;
	}

	/* Return the directory inode and exit. */
	put_inode(rlast_dir_ptr);
	err_code = r;
	return(rip);
}

PRIVATE int pipe_open(rip, bits, oflags)
register struct inode *rip;
register mode_t bits;
register int oflags;
{ /*  This function is called from common_open. It checks if there is at least one reader/writer pair for the pipe, if not
   *  it suspends the caller, otherwise it revives all other blocked processes hanging on the pipe. */

	if (find_filp(rip, bits & W_BIT ? R_BIT : W_BIT) == NIL_FILP) {
		if (oflags & O_NONBLOCK) {
			if (bits & W_BIT) return(ENXIO);
		} else
			suspend(XPOPEN);	/* suspend caller */
	} else if (susp_count > 0) {/* revive blocked processes */
		release(rip, OPEN, susp_count);
		release(rip, CREAT, susp_count);
	}
	rip->i_pipe = I_PIPE;

	return(OK);
}

PUBLIC int do_mknod()
{ /* Perform the mknod(name, mode, addr) system call. */

	register mode_t bits, mode_bits;
	struct inode *ip;

	/* Only the super_user may make nodes other than fifos. */
	mode_bits = (mode_t) m.m1_i2;		/* mode of the inode */
	if (!super_user && ((mode_bits & I_TYPE) != I_NAMED_PIPE)) return(EPERM);
	if (fetch_name(m.m1_p1, m.m1_i1, M1) != OK) return(err_code);
	bits = (mode_bits & I_TYPE) | (mode_bits & ALL_MODES & fp->fp_umask);
	ip = new_node(user_path, bits, (zone_t) m.m1_i3);
	put_inode(ip);
	return(err_code);
}

PUBLIC int do_mkdir()
{ /* Perform the mkdir(name, mode) system call. */

	int r1, r2;			/* status codes */
	ino_t dot, dotdot;		/* inode numbers for . and .. */
	mode_t bits;			/* mode bits for the new inode */
	char string[NAME_MAX];	/* last component of the new dir's path name */
	register struct inode *rip, *ldirp;
```

```
  /* Check to see if it is possible to make another link in the parent dir. */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  ldirp = last_dir(user_path, string);              /* pointer to new dir's parent */
  if (ldirp == NIL_INODE) return(err_code);
  if ( (ldirp->i_nlinks & BYTE) >= LINK_MAX) {
               put_inode(ldirp);        /* return parent */
               return(EMLINK);
  }
  if(ldirp->i_sp->s_version == V3 && IS_IMMUTABLE(ldirp)){
       printk("Parent Directory is Immutable");
       return (EPERM);
  }
  /* Next make the inode. If that fails, return error code. */
  bits = I_DIRECTORY | (mode & RWX_MODES & fp->fp_umask);
  rip = new_node(user_path, bits, (zone_t) 0);
  if (rip == NIL_INODE || err_code == EEXIST) {
               put_inode(rip);                      /* can't make dir: it already exists */
               put_inode(ldirp);        /* return parent too */
               return(err_code);
  }

  /* Get the inode numbers for . and .. to enter in the directory. */
  dotdot = ldirp->i_num;            /* parent's inode number */
  dot = rip->i_num;                 /* inode number of the new dir itself */

  /* Now make dir entries for . and .. unless the disk is completely full. */
  /* Use dot1 and dot2, so the mode of the directory isn't important. */
  rip->i_mode = bits;   /* set mode */
  r1 = search_dir(rip, dot1, &dot, ENTER);   /* enter . in the new dir */
  r2 = search_dir(rip, dot2, &dotdot, ENTER);            /* enter .. in the new dir */

  /* If both . and .. were successfully entered, increment the link counts. */
  if (r1 == OK && r2 == OK) {
               /* Normal case.  It was possible to enter . and .. in the new dir. */
               rip->i_nlinks++;         /* this accounts for . */
               ldirp->i_nlinks++;       /* this accounts for .. */
               ldirp->i_dirt = DIRTY;           /* mark parent's inode as dirty */
  } else {
               /* It was not possible to enter . or .. probably disk was full. */
               (void) search_dir(ldirp, string, (ino_t *) 0, DELETE);
               rip->i_nlinks--;         /* undo the increment done in new_node() */
  }
  rip->i_dirt = DIRTY;                          /* either way, i_nlinks has changed */

  put_inode(ldirp);                 /* return the inode of the parent dir */
  put_inode(rip);                   /* return the inode of the newly made dir */
  return(err_code);                 /* new_node() always sets 'err_code' */
}

PUBLIC int do_close()
{ /* Perform the close(fd) system call. */

  register struct filp *rfilp;
  register struct inode *rip;
  struct file_lock *flp;
  int rw, mode_word, major, task, lock_count;
  dev_t dev;

  /* First locate the inode that belongs to the file descriptor. */
  if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);
  rip = rfilp->filp_ino; /* 'rip' points to the inode */

  if (rfilp->filp_count - 1 == 0 && rfilp->filp_mode != FILP_CLOSED) {
               /* Check to see if the file is special. */
               mode_word = rip->i_mode & I_TYPE;
               if (mode_word == I_CHAR_SPECIAL || mode_word == I_BLOCK_SPECIAL) {
                            dev = (dev_t) rip->i_zone[0];
                            if (mode_word == I_BLOCK_SPECIAL)  {
                                         /* Invalidate cache entries unless special is mounted
```

```
                                        * or ROOT
                                        */
                                       if (!mounted(rip)) {
                                              (void) do_sync();/* purge cache */
                                                     invalidate(dev);
                                       }
                               }
                               /* Use the dmap_close entry to do any special processing
                                * required.
                                */
                               dev_mess.m_type = DEV_CLOSE;
                               dev_mess.DEVICE = dev;
                               major = (dev >> MAJOR) & BYTE;          /* major device nr */
                               task = dmap[major].dmap_task;   /* device task nr */
                               (*dmap[major].dmap_close)(task, &dev_mess);
                       }
               }

               /* If the inode being closed is a pipe, release everyone hanging on it. */
               if (rip->i_pipe == I_PIPE) {
                       rw = (rfilp->filp_mode & R_BIT ? WRITE : READ);
                       release(rip, rw, NR_PROCS);
               }

               /* If a write has been done, the inode is already marked as DIRTY. */
               if (--rfilp->filp_count == 0) {
                       if (rip->i_pipe == I_PIPE && rip->i_count > 1) {
                               /* Save the file position in the i-node in case needed later.
                                * The read and write positions are saved separately.  The
                                * last 3 zones in the i-node are not used for (named) pipes.
                                */
                               if (rfilp->filp_mode == R_BIT)
                                       rip->i_zone[V2_NR_DZONES+1] = (zone_t) rfilp->filp_pos;
                               else
                                       rip->i_zone[V2_NR_DZONES+2] = (zone_t) rfilp->filp_pos;
                       }
                       put_inode(rip);
               }

               fp->fp_cloexec &= ~(1L << fd);          /* turn off close-on-exec bit */
               fp->fp_filp[fd] = NIL_FILP;

               /* Check to see if the file is locked.  If so, release all locks. */
               if (nr_locks == 0) return(OK);
               lock_count = nr_locks;          /* save count of locks */
               for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
                       if (flp->lock_type == 0) continue;              /* slot not in use */
                       if (flp->lock_inode == rip && flp->lock_pid == fp->fp_pid) {
                               flp->lock_type = 0;
                               nr_locks--;
                       }
               }
               if (nr_locks < lock_count) lock_revive();    /* lock released */
               return(OK);
       }

       PUBLIC int do_lseek()
       { /* Perform the lseek(ls_fd, offset, whence) system call. */

               register struct filp *rfilp;
               register off_t pos;

               /* Check to see if the file descriptor is valid. */
               if ( (rfilp = get_filp(ls_fd)) == NIL_FILP) return(err_code);

               /* No lseek on pipes. */
               if (rfilp->filp_ino->i_pipe == I_PIPE) return(ESPIPE);

               /* The value of 'whence' determines the start position to use. */
               switch(whence) {
```

```
            case 0:     pos = 0;    break;
            case 1: pos = rfilp->filp_pos;      break;
            case 2: pos = rfilp->filp_ino->i_size;          break;
            default: return(EINVAL);
 }

 /* Check for overflow. */
 if (((long)offset > 0) && ((long)(pos + offset) < (long)pos)) return(EINVAL);
 if (((long)offset < 0) && ((long)(pos + offset) > (long)pos)) return(EINVAL);
 pos = pos + offset;

 if (pos != rfilp->filp_pos)
            rfilp->filp_ino->i_seek = ISEEK;/* inhibit read ahead */
 rfilp->filp_pos = pos;
 reply_l1 = pos;                    /* insert the long into the output message */
 return(OK);
}
```

link.c

/* This file handles the LINK and UNLINK system calls. It also deals with deallocating the storage used by a file when the last UNLINK is done to a file and the blocks must be returned to the free block pool.*/

```
#include "fs.h"
#include <sys/stat.h>
#include <string.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"
#include <fcntl.h>

#define SAME 1000

FORWARD _PROTOTYPE( int remove_dir, (struct inode *rldirp, struct inode *rip,
                                     char dir_name[NAME_MAX])                   );
FORWARD _PROTOTYPE( int unlink_file, (struct inode *dirp, struct inode *rip,
                                     char file_name[NAME_MAX])                  );

PUBLIC int do_link()
{ /* Perform the link(name1, name2) system call. */

  register struct inode *ip, *rip;
  register int r;
  char string[NAME_MAX];
  struct inode *new_ip;

  /* See if 'name' (file to be linked) exists. */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Check to see if the file or directory is immutable . We can not create links
  on immutable file or directory        */
  if(rip->i_sp->s_version == V3 && IS_IMMUTABLE(rip)){
     printk("File is Immutable ");
     return (EPERM);
   }
  /* Check to see if the file has maximum number of links already. */
  r = OK;
  if ( (rip->i_nlinks & BYTE) >= LINK_MAX) r = EMLINK;

  /* Only super_user may link to directories. */
  if (r == OK)
```

```
        if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;

  /* If error with 'name', return the inode. */
  if (r != OK) {
        put_inode(rip);
        return(r);
  }

  /* Does the final directory of 'name2' exist? */
  if (fetch_name(name2, name2_length, M1) != OK) {
        put_inode(rip);
        return(err_code);
  }
  if ( (ip = last_dir(user_path, string)) == NIL_INODE) r = err_code;

  /* If 'name2' exists in full (even if no space) set 'r' to error. */
  if (r == OK) {
        if ( (new_ip = advance(ip, string)) == NIL_INODE) {
                r = err_code;
                if (r == ENOENT) r = OK;
        } else {
                put_inode(new_ip);
                r = EEXIST;
        }
  }

  /* Check for links across devices. */
  if (r == OK)
        if (rip->i_dev != ip->i_dev) r = EXDEV;

  /* Try to link. */
  if (r == OK)
        r = search_dir(ip, string, &rip->i_num, ENTER);

  /* If success, register the linking. */
  if (r == OK) {
        rip->i_nlinks++;
        rip->i_update |= CTIME;
        rip->i_dirt = DIRTY;
  }
  /* Done.  Release both inodes. */
  put_inode(rip);
  put_inode(ip);
  return(r);
}

PUBLIC int do_unlink()
{ /* Perform the unlink(name) or rmdir(name) system call. The code for these two is almost the same.
They differ only in some condition testing.  Unlink() may be used by the superuser to do dangerous
things; rmdir() may not. */

  register struct inode *rip;
  struct inode *rldirp;
  int r;
  char string[NAME_MAX];

  /* Get the last directory in the path. */
  if (fetch_name(name, name_length, M3) != OK) return(err_code);
  if ( (rldirp = last_dir(user_path, string)) == NIL_INODE)
        return(err_code);

  /* The last directory exists.  Does the file also exist? */
  r = OK;
  if ( (rip = advance(rldirp, string)) == NIL_INODE) r = err_code;

  /* If error, return inode. */
  if (r != OK) {
        put_inode(rldirp);
        return(r);
  }
```

```
/* Do not remove a mount point. */
  if (rip->i_num == ROOT_INODE) {
                put_inode(rldirp);
                put_inode(rip);
                return(EBUSY);
  }

  /* Now test if the call is allowed, separately for unlink() and rmdir(). */
  if (fs_call == UNLINK) {
                /* Only the su may unlink directories, but the su can unlink any dir.*/
                if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;

                /* Don't unlink a file if it is the root of a mounted file system. */
                if (rip->i_num == ROOT_INODE) r = EBUSY;

                /* Actually try to unlink the file; fails if parent is mode 0 etc. */
                if (r == OK) r = unlink_file(rldirp, rip, string);
  } else {
                r = remove_dir(rldirp, rip, string); /* call is RMDIR */
  }

  /* If unlink was possible, it has been done, otherwise it has not. */
  put_inode(rip);
  put_inode(rldirp);
  return(r);
}

PUBLIC int do_rename()
{  /* Perform the rename(name1, name2) system call. */

  struct inode *old_dirp, *old_ip; /* ptrs to old dir, file inodes */
  struct inode *new_dirp, *new_ip;              /* ptrs to new dir, file inodes */
  struct inode *new_superdirp, *next_new_superdirp;
  int r = OK;                                                   /* error flag; initially no error */
  int odir, ndir;                               /* TRUE iff {old|new} file is dir */
  int same_pdir;                                /* TRUE iff parent dirs are the same */
  char old_name[NAME_MAX], new_name[NAME_MAX];
  ino_t numb;
  int r1;

  /* See if 'name1' (existing file) exists.  Get dir and file inodes. */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  if ( (old_dirp = last_dir(user_path, old_name))==NIL_INODE) return(err_code);

  if ( (old_ip = advance(old_dirp, old_name)) == NIL_INODE) r = err_code;
  /* Check that the file or directory is Immutbale or not. We can not rename a
  Immutable file or Directory            */
  if(old_ip->i_sp->s_version ==V3 && IS_IMMUTABLE(old_ip)){
     printk("File/Directory is Immutable");
     return (EPERM);
  }
 /* Check to see if the parent directory is APPEND only .
 we can not modify the previous directory entry in a APPEND only Directory */
 if(old_dirp->i_sp->s_version == V3 && IS_APPEND(old_dirp)){
    printk("Parent Directory is APPEND only");
    return (EPERM);
 }
 /*check wheather the parent directory is immutable directory. We can not
 modifies directory entries in a immutable directory. */
 if(old_dirp->i_sp->s_version == V3 && IS_IMMUTABLE(old_dirp)){
     printk("Parent Directory is Immutable");
     return (EPERM);

  }
   /* See if 'name2' (new name) exists.  Get dir and file inodes. */
  if (fetch_name(name2, name2_length, M1) != OK) r = err_code;
  if ( (new_dirp = last_dir(user_path, new_name)) == NIL_INODE) r = err_code;
  new_ip = advance(new_dirp, new_name);   /* not required to exist */

  if (old_ip != NIL_INODE)
```

```
        odir = ((old_ip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */

/* If it is ok, check for a variety of possible errors. */
if (r == OK) {
        same_pdir = (old_dirp == new_dirp);

        /* The old inode must not be a superdirectory of the new last dir. */
        if (odir && !same_pdir) {
                dup_inode(new_superdirp = new_dirp);
                while (TRUE) {                  /* may hang in a file system loop */
                        if (new_superdirp == old_ip) {
                                r = EINVAL;
                                break;
                        }
                        next_new_superdirp = advance(new_superdirp, dot2);
                        put_inode(new_superdirp);
                        if (next_new_superdirp == new_superdirp)
                                break;  /* back at system root directory */
                        new_superdirp = next_new_superdirp;
                        if (new_superdirp == NIL_INODE) {
                                /* Missing ".." entry.  Assume the worst. */
                                r = EINVAL;
                                break;
                        }
                }
                put_inode(new_superdirp);
        }

        /* The old or new name must not be . or .. */
        if (strcmp(old_name, ".")==0 || strcmp(old_name, "..")==0 ||
           strcmp(new_name, ".")==0 || strcmp(new_name, "..")==0) r = EINVAL;

        /* Both parent directories must be on the same device. */
        if (old_dirp->i_dev != new_dirp->i_dev) r = EXDEV;

        /* Parent dirs must be writable, searchable and on a writable device */
        if ((r1 = forbidden(old_dirp, W_BIT | X_BIT)) != OK ||
           (r1 = forbidden(new_dirp, W_BIT | X_BIT)) != OK) r = r1;

        /* Some tests apply only if the new path exists. */
        if (new_ip == NIL_INODE) {
                /* don't rename a file with a file system mounted on it. */
                if (old_ip->i_dev != old_dirp->i_dev) r = EXDEV;
                if (odir && (new_dirp->i_nlinks & BYTE) >= LINK_MAX &&
                   !same_pdir && r == OK) r = EMLINK;
        } else {
                if (old_ip == new_ip) r = SAME; /* old=new */

                /* has the old file or new file a file system mounted on it? */
                if (old_ip->i_dev != new_ip->i_dev) r = EXDEV;

                ndir = ((new_ip->i_mode & I_TYPE) == I_DIRECTORY); /* dir ? */
                if (odir == TRUE && ndir == FALSE) r = ENOTDIR;
                if (odir == FALSE && ndir == TRUE) r = EISDIR;
        }
}
if (r == OK) {
        if (new_ip != NIL_INODE) {
                 /* There is already an entry for 'new'. Try to remove it. */
                if (odir)
                        r = remove_dir(new_dirp, new_ip, new_name);
                else
                        r = unlink_file(new_dirp, new_ip, new_name);
        }
        /* if r is OK, the rename will succeed, while there is now an unused entry in the new parent directory.  */
}

 if (r == OK) {
/* If the new name will be in the same parent directory as the old one, first remove the old name to free an entry for
the new name, otherwise first try to create the new name entry to make sure the rename will succeed.*/
```

```
                numb = old_ip->i_num;                           /* inode number of old file */
                if (same_pdir) {
                        r = search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);         /* shouldn't go wrong. */
                        if (r==OK) (void) search_dir(old_dirp, new_name, &numb, ENTER);
                } else {
                        r = search_dir(new_dirp, new_name, &numb, ENTER);
                        if (r == OK)
                           (void) search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
                }
 }
/* If r is OK, the ctime and mtime of old_dirp and new_dirp have been marked for update in search_dir.*/

 if (r == OK && odir && !same_pdir) {
                /* Update the .. entry in the directory (still points to old_dirp). */
                numb = new_dirp->i_num;
                (void) unlink_file(old_ip, NIL_INODE, dot2);
                if (search_dir(old_ip, dot2, &numb, ENTER) == OK) {
                        /* New link created. */
                        new_dirp->i_nlinks++;
                        new_dirp->i_dirt = DIRTY;
                }
 }

 /* Release the inodes. */
 put_inode(old_dirp);
 put_inode(old_ip);
 put_inode(new_dirp);
 put_inode(new_ip);
 return(r == SAME ? OK : r);
}

PUBLIC void truncate(rip)
register struct inode *rip;             /* pointer to inode to be truncated */
{    /* Remove all the zones from the inode 'rip' and mark it dirty. */

  register block_t b;
  zone_t z, zone_size, z1;
  off_t position;
  int i, scale, file_type, waspipe, single, nr_indirects;
  struct buf *bp;
  dev_t dev;

  /* Check to see if the file or directory is IMMUTABLE . we can not truncate the
  IMMUTABLE file/directory        */
  if(rip->i_sp->s_version == V3 && IS_IMMUTABLE(rip)){
    printk("File is IMMUTABLE ");
    return ;
  }
  file_type = rip->i_mode & I_TYPE;           /* check to see if file is special */
  if (file_type == I_CHAR_SPECIAL || file_type == I_BLOCK_SPECIAL) return;
  dev = rip->i_dev;                     /* device on which inode resides */
  scale = rip->i_sp->s_log_zone_size;
  zone_size = (zone_t) BLOCK_SIZE << scale;
  nr_indirects = rip->i_nindirs;

  /* Pipes can shrink, so adjust size to make sure all zones are removed. */
  waspipe = rip->i_pipe == I_PIPE;              /* TRUE is this was a pipe */
  if (waspipe) rip->i_size = PIPE_SIZE;

  /* Step through the file a zone at a time, finding and freeing the zones. */
  for (position = 0; position < rip->i_size; position += zone_size) {
                if ( (b = read_map(rip, position)) != NO_BLOCK) {
                        z = (zone_t) b >> scale;
                        free_zone(dev, z);
                }
  }

  /* All the data zones have been freed.  Now free the indirect zones. */
  rip->i_dirt = DIRTY;
  if (waspipe) {
```

```
                wipe_inode(rip);        /* clear out inode for pipes */
                return;                 /* indirect slots contain file positions */
        }
        single = rip->i_ndzones;
        free_zone(dev, rip->i_zone[single]);    /* single indirect zone */
        if ( (z = rip->i_zone[single+1]) != NO_ZONE) {
                /* Free all the single indirect zones pointed to by the double. */
                b = (block_t) z << scale;
                bp = get_block(dev, b, NORMAL);         /* get double indirect zone */
                for (i = 0; i < nr_indirects; i++) {
                        z1 = rd_indir(bp, i);
                        free_zone(dev, z1);
                }

                /* Now free the double indirect zone itself. */
                put_block(bp, INDIRECT_BLOCK);
                free_zone(dev, z);
        }
}

PRIVATE int remove_dir(rldirp, rip, dir_name)
struct inode *rldirp;                   /* parent directory */
struct inode *rip;                      /* directory to be removed */
char dir_name[NAME_MAX];                /* name of directory to be removed */
{  /* A directory file has to be removed. Five conditions have to met:
    *           - The file must be a directory
    *           - The directory must be empty (except for . and ..)
    *           - The final component of the path must not be . or ..
    *           - The directory must not be the root of a mounted file system
    *           - The parent directory must not be immutable/append only.
- The directory must not be anybody's root/working directory  */

  int r;
  register struct fproc *rfp;
  /* Check for the IMMUTABLE and append only aprent directory parent directory */
  if(rip->i_sp->s_version == V3 && (IS_IMMUTABLE(rldirp) || IS_APPEND(rldirp))){
    printf("Directory is IMMUTABLE or APPEND only ");
    return (EPERM);
  }
  /* search_dir checks that rip is a directory too. */
  if ((r = search_dir(rip, "", (ino_t *) 0, IS_EMPTY)) != OK) return r;

  if (strcmp(dir_name, ".") == 0 || strcmp(dir_name, "..") == 0)return(EINVAL);
  if (rip->i_num == ROOT_INODE) return(EBUSY); /* can't remove 'root' */

  for (rfp = &fproc[INIT_PROC_NR + 1]; rfp < &fproc[NR_PROCS]; rfp++)
                if (rfp->fp_workdir == rip || rfp->fp_rootdir == rip) return(EBUSY);
/* can't remove anybody's working dir */

  /* Actually try to unlink the file; fails if parent is mode 0 etc. */
  if ((r = unlink_file(rldirp, rip, dir_name)) != OK) return r;

  /* Unlink . and .. from the dir. The super user can link and unlink any dir,
   * so don't make too many assumptions about them.  */
  (void) unlink_file(rip, NIL_INODE, dot1);
  (void) unlink_file(rip, NIL_INODE, dot2);
  return(OK);
}

PRIVATE int unlink_file(dirp, rip, file_name)
struct inode *dirp;                     /* parent directory of file */
struct inode *rip;                      /* inode of file, may be NIL_INODE too. */
char file_name[NAME_MAX];   /* name of file to be removed */
{ /* Unlink 'file_name'; rip must be the inode of 'file_name' or NIL_INODE. */

  ino_t numb;                           /* inode number */
  int           r;

  /* If rip is not NIL_INODE, it is used to get faster access to the inode. */
  if (rip == NIL_INODE) {
```

```
                /* Search for file in directory and try to get its inode. */
                err_code = search_dir(dirp, file_name, &numb, LOOK_UP);
                if (err_code == OK) rip = get_inode(dirp->i_dev, (int) numb);
                if (err_code != OK || rip == NIL_INODE) return(err_code);
 } else {
                dup_inode(rip);                         /* inode will be returned with put_inode */
 }

 r = search_dir(dirp, file_name, (ino_t *) 0, DELETE);

 if (r == OK) {
                rip->i_nlinks--;          /* entry deleted from parent's dir */
                rip->i_update |= CTIME;
                rip->i_dirt = DIRTY;

 if(rip->i_nlinks == 0 && rip->i_sp->s_version == V3 && IS_SECRM(rip)){
      printk("Secure Deleting file");
      sec_delete(rip);
  }
}
 put_inode(rip);
 return(r);
}
/*****************************************************************************/
/******************  Secure Delete ***************************************/
/*****************************************************************************/

void sec_delete(rip)
register struct inode*rip;
{
register block_t b;
zone_t z, zone_size,z1;
off_t position;
int i, scale, file_type, single, nr_indirects;
struct buf * bp;
dev_t dev;

if ( rip->i_sp->s_version !=V3)
{
    printk("Not MyFS File System"); return;
}
file_type = rip->i_mode & I_TYPE;
if (file_type == I_CHAR_SPECIAL || file_type == I_BLOCK_SPECIAL || rip->i_pipe == I_PIPE)
     return;

dev = rip->i_dev;
scale = rip->i_sp->s_log_zone_size;
zone_size = (zone_t) BLOCK_SIZE <<scale;
nr_indirects = rip->i_nindirs;

/*    find and delete all the data zones   */

for (position =0; position < rip->i_size; position += zone_size)
{
   if( (b = read_map(rip, position)) != NO_BLOCK) {
        z = (zone_t) b >> scale ;
        overwrite(dev, z, scale);
        free_zone(dev, z);
   }

}
/* Now Delete all the zone entries from the inode                */
 for( i=0; i< rip->i_ndzones; i++){
     overwrite(dev,rip->i_zone[i],scale);
     rip->i_zone[i] = 0;
 }

 single = rip->i_ndzones;
if ( (z = rip->i_zone[single]) != NO_ZONE )  overwrite(dev,z,scale);
if ( (z  = rip->i_zone[single]) != NO_ZONE )  del_indirects(dev,z,scale);
```

```
if( (z = rip->i_zone[single+2]) != NO_ZONE ){
     b = (block_t) z << scale;
     bp =get_block(dev, b, NORMAL);
     for (i =0; i< V3_INDIRECTS; i++){
          z1 = rd_indir(bp,i);
          del_indirects(dev,z1,scale);
     }
 }
 bp->b_dirt = DIRTY;
 put_block(bp, INDIRECT_BLOCK);
 overwrite(dev,z,scale);

 free_zone(dev,z);
}
/*************************delete indirect zones     ************************/
void del_indirects(dev,z,scale)
dev_t dev;
zone_t z;
int scale;
{
 block_t b;
 struct buf * bp;
 int i;
 zone_t z1;
 b = (block_t)z << scale;
 bp = get_block(dev,b, NORMAL);
 for (i=0; i< V3_INDIRECTS ; i++) {
     z1 = rd_indir(bp,i);
     overwrite(dev,z1,scale);
     free_zone(dev,z1);
 }
bp->b_dirt = DIRTY;
put_block(bp, INDIRECT_BLOCK);
overwrite(dev,z,scale);

}
/*****************************************************************************/
/***********************      OverWrite Data     ***************************/
/*****************************************************************************/
void overwrite(dev , z,scale)
dev_t dev;
zone_t z;
int scale;
{
static char bf[1024]={0,0};

char patt[20][3] = {{0,0,0},
                    {0x55,0x55,0x55},
                    {0xAA,0xAA,0xAA},
                    {0x00,0x00,0x00},
                    {0x11,0x11,0x11},
                    { 0x22,0x22,0x22},
                    { 0x33,0x33,0x33},
                    { 0x44,0x44,0x44},
                    { 0x55,0x55,0x55},
                    { 0x66,0x66,0x66},
                    { 0x77,0x77,0x77},
                    { 0x88,0x88,0x88},
                    { 0x99,0x99,0x99},
                    { 0xAA,0xAA,0xAA},
                    { 0xBB,0xBB,0xBB},
                    { 0xCC,0xCC,0xCC},
                    { 0xDD,0xDD,0xDD},
                    { 0xEE,0xEE,0xEE},
                    { 0xFF,0xFF,0xFF},
                    { 0,0,0},
                   };

int p,i,zonesize=1,op,j,res=0;
off_t pos;
```

```
block_t b=0;

b = z << scale;
pos = b * BLOCK_SIZE;
op = DEV_WRITE;
flushall(dev);
for(i=1;i<scale;i++)     zonesize *= 2;

/* Loop for the pattern number   */
for(p=23;p<24;p++){
/*Fill the buffer with the specific pattern    */
        for(i=0;i<BLOCK_SIZE;i+=3){
         bf[i] = patt[p][0];
         bf[i+1] = patt[p][1];
         bf[i+2] = patt[p][2];
     }
     /* Write the Data directly on the Disk     */
     /*    Bypass the Cache                    */
     pos = b*BLOCK_SIZE;
    for(j=1;j<=zonesize;j++){
         res = 0;

         dev_io(op,FALSE,dev,pos,BLOCK_SIZE,FS_PROC_NR,(char*)bf);
         if(res != BLOCK_SIZE)   {
                printk("Write Error");
                return;
          }
         pos+= BLOCK_SIZE;
     }
}
}
```

misc.c

```
/* This file contains a collection of miscellaneous procedures.  Some of them  perform simple system calls.  Some
others do a little part of system calls that are mostly performed by the Memory Manager.
*   do_fcntl:                perform the FCNTL system call   */

#include "fs.h"
#include <fcntl.h>
#include <unistd.h>    /* cc runs out of memory with unistd.h :-( */
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/boot.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "dev.h"
#include "param.h"


PUBLIC int do_dup()
{ /* Perform the dup(fd) or dup2(fd,fd2) system call. These system calls are obsolete.  In fact, it is not
even possible to invoke them using the current library because the library routines call fcntl().  They
are provided to permit old binary programs to continue to run.*/

  register int rfd;
  register struct filp *f;
  struct filp *dummy;
  int r;

  /* Is the file descriptor valid? */
  rfd = fd & ~DUP_MASK;                         /* kill off dup2 bit, if on */
  if ((f = get_filp(rfd)) == NIL_FILP) return(err_code);

  /* Distinguish between dup and dup2. */
  if (fd == rfd) {                              /* bit not on */
                /* dup(fd) */
```

```
                if ( (r = get_fd(0, 0, &fd2, &dummy)) != OK) return(r);
  } else {
                /* dup2(fd, fd2) */
                if (fd2 < 0 || fd2 >= OPEN_MAX) return(EBADF);
                if (rfd == fd2) return(fd2);            /* ignore the call: dup2(x, x) */
                fd = fd2;                       /* prepare to close fd2 */
                (void) do_close();      /* cannot fail */
  }

  /* Success. Set up new file descriptors. */
  f->filp_count++;
  fp->fp_filp[fd2] = f;
  return(fd2);
}

PUBLIC int do_fcntl()
{ /* Perform the fcntl(fd, request, ...) system call. */

  register struct filp *f;
  int new_fd, r, fl,flags=0;
  long cloexec_mask;                    /* bit map for the FD_CLOEXEC flag */
  long clo_value;                       /* FD_CLOEXEC flag in proper position */
  struct filp *dummy;
  struct inode * file_ino;
  unsigned int oldflags=0;

  /* Is the file descriptor valid? */
  if ((f = get_filp(fd)) == NIL_FILP) return(err_code);
  file_ino = f->filp_ino;

  switch (request) {
    case F_DUPFD:
                /* This replaces the old dup() system call. */
                if (addr < 0 || addr >= OPEN_MAX) return(EINVAL);
                if ((r = get_fd(addr, 0, &new_fd, &dummy)) != OK) return(r);
                f->filp_count++;
                fp->fp_filp[new_fd] = f;
                return(new_fd);

    case F_GETFD:
                /* Get close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
                return( ((fp->fp_cloexec >> fd) & 01) ? FD_CLOEXEC : 0);

    case F_SETFD:
                /* Set close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
                cloexec_mask = 1L << fd;            /* singleton set position ok */
                clo_value = (addr & FD_CLOEXEC ? cloexec_mask : 0L);
                fp->fp_cloexec = (fp->fp_cloexec & ~cloexec_mask) | clo_value;
                return(OK);

    case F_GETFL:
                /* Get file status flags (O_NONBLOCK and O_APPEND). */
                fl = f->filp_flags & (O_NONBLOCK | O_APPEND | O_ACCMODE);
                return(fl);

    case F_SETFL:
                /* Set file status flags (O_NONBLOCK and O_APPEND). */
                fl = O_NONBLOCK | O_APPEND;
                f->filp_flags = (f->filp_flags & ~fl) | (addr & fl);
                return(OK);

    case F_GETLK:
    case F_SETLK:
    case F_SETLKW:
                /* Set or clear a file lock. */
                r = lock_op(f, request);
                return(r);
    case F_GETETFL:
    flags = file_ino->i_flags;
    return (flags);
```

```
   case F_SETETFL:
    {       /* set or clear extended flags   */
      flags = addr;
      if((fp->fp_realuid != file_ino->i_uid))
      {       printk("Protection Problem ");
              return -5;
       }
      oldflags = file_ino->i_flags;
      file_ino->i_flags |= flags;
       /* addr have the new flags  */
      if(flags & E_SYNC )
              file_ino->i_flags |= E_SYNC;
      else
              file_ino->i_flags &= ~E_SYNC;

      if(flags & E_SECRM )
        file_ino->i_flags |= E_SECRM;
      else
        file_ino->i_flags &= ~E_SECRM;

      if( flags & E_APPEND )
        file_ino->i_flags |= E_APPEND;
      else
        file_ino->i_flags &= ~E_APPEND;

      if( flags & E_IMMUTABLE )
        file_ino->i_flags |= E_IMMUTABLE;
      else
        file_ino->i_flags &= ~E_IMMUTABLE;

       file_ino->i_update |= CTIME;
       file_ino->i_dirt =  DIRTY;
       put_inode(file_ino);
     return 0;
    }
   default:
              return(EINVAL);
  }
}
```

mount.c

```
/* This file performs the MOUNT and UMOUNT system calls.
  * The entry points into this file are
 *   do_mount:          perform the MOUNT system call
 *   do_umount:         perform the UMOUNT system call  */

#include "fs.h"
#include <fcntl.h>
#include <minix/com.h>
#include <sys/stat.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

PRIVATE message dev_mess;
FORWARD _PROTOTYPE( dev_t name_to_dev, (char *path)                              );

PUBLIC int do_mount()
{ /* Perform the mount(name, mfile, rd_only) system call. */

  register struct inode *rip, *root_ip;
  struct super_block *xp, *sp;
  dev_t dev;
```

```
mode_t bits;
int rdir, mdir;                 /* TRUE iff {root|mount} file is dir */
int r, found, major, task;

/* Only the super-user may do MOUNT. */
if (!super_user) return(EPERM);

/* If 'name' is not for a block special file, return error. */
if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);

/* Scan super block table to see if dev already mounted & find a free slot.*/
sp = NIL_SUPER;
found = FALSE;
for (xp = &super_block[0]; xp < &super_block[NR_SUPERS]; xp++) {
        if (xp->s_dev == dev) found = TRUE;     /* is it mounted already? */
        if (xp->s_dev == NO_DEV) sp = xp;       /* record free slot */
}
if (found) return(EBUSY);       /* already mounted */
if (sp == NIL_SUPER) return(ENFILE);    /* no super block available */

dev_mess.m_type = DEV_OPEN;                     /* distinguish from close */
dev_mess.DEVICE = dev;                  /* Touch the device. */
if (rd_only) dev_mess.COUNT = R_BIT;
else  dev_mess.COUNT = R_BIT|W_BIT;

major = (dev >> MAJOR) & BYTE;
if (major <= 0 || major >= max_major) return(ENODEV);
task = dmap[major].dmap_task;           /* device task nr */
(*dmap[major].dmap_open)(task, &dev_mess);
if (dev_mess.REP_STATUS != OK) return(EINVAL);

/* Fill in the super block. */
sp->s_dev = dev;                /* read_super() needs to know which dev */
r = read_super(sp);

/* Is it recognized as a Minix filesystem? */
if (r != OK) {
        dev_mess.m_type = DEV_CLOSE;
        dev_mess.DEVICE = dev;
        (*dmap[major].dmap_close)(task, &dev_mess);
        return(r);
}

/* Now get the inode of the file to be mounted on. */
if (fetch_name(name2, name2_length, M1) != OK) {
        sp->s_dev = NO_DEV;
        dev_mess.m_type = DEV_CLOSE;
        dev_mess.DEVICE = dev;
        (*dmap[major].dmap_close)(task, &dev_mess);
        return(err_code);
}
if ( (rip = eat_path(user_path)) == NIL_INODE) {
        sp->s_dev = NO_DEV;
        dev_mess.m_type = DEV_CLOSE;
        dev_mess.DEVICE = dev;
        (*dmap[major].dmap_close)(task, &dev_mess);
        return(err_code);
}

/* It may not be busy. */
r = OK;
if (rip->i_count > 1) r = EBUSY;

/* It may not be special. */
bits = rip->i_mode & I_TYPE;
if (bits == I_BLOCK_SPECIAL || bits == I_CHAR_SPECIAL) r = ENOTDIR;

/* Get the root inode of the mounted file system. */
root_ip = NIL_INODE;                    /* if 'r' not OK, make sure this is defined */
```

```
  if (r == OK) {
        if ( (root_ip = get_inode(dev, ROOT_INODE)) == NIL_INODE) r = err_code;
  }
  if (root_ip != NIL_INODE && root_ip->i_mode == 0) r = EINVAL;

  /* File types of 'rip' and 'root_ip' may not conflict. */
  if (r == OK) {
        mdir = ((rip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */
        rdir = ((root_ip->i_mode & I_TYPE) == I_DIRECTORY);
        if (!mdir && rdir) r = EISDIR;
  }

  /* If error, return the super block and both inodes; release the maps. */
  if (r != OK) {
        put_inode(rip);
        put_inode(root_ip);
        (void) do_sync();
        invalidate(dev);

        sp->s_dev = NO_DEV;
        dev_mess.m_type = DEV_CLOSE;
        dev_mess.DEVICE = dev;
        (*dmap[major].dmap_close)(task, &dev_mess);
        return(r);
  }

  /* Nothing else can go wrong.  Perform the mount. */
  rip->i_mount = I_MOUNT;       /* this bit says the inode is mounted on */
  sp->s_imount = rip;
  sp->s_isup = root_ip;
  sp->s_rd_only = rd_only;
  sp->s_state = ERROR_FS;
  return(OK);
}

PUBLIC int do_umount()
{ /* Perform the umount(name) system call. */

  register struct inode *rip;
  struct super_block *sp, *sp1;
  dev_t dev;
  int count;
  int major, task;

  /* Only the super-user may do UMOUNT. */
  if (!super_user) return(EPERM);

  /* If 'name' is not for a block special file, return error. */
  if (fetch_name(name, name_length, M3) != OK) return(err_code);
  if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);

  /* See if the mounted device is busy.  Only 1 inode using it should be
   * open -- the root inode -- and that inode only 1 time.
   */
  count = 0;
  for (rip = &inode[0]; rip< &inode[NR_INODES]; rip++)
        if (rip->i_count > 0 && rip->i_dev == dev) count += rip->i_count;
  if (count > 1) return(EBUSY);   /* can't umount a busy file system */

  /* Find the super block. */
  sp = NIL_SUPER;
  for (sp1 = &super_block[0]; sp1 < &super_block[NR_SUPERS]; sp1++) {
        if (sp1->s_dev == dev) {
                sp = sp1;
                break;
        }
  }

  /* Sync the disk, and invalidate cache. */
  (void) do_sync();                     /* force any cached blocks out of memory */
```

```
  invalidate(dev);                      /* invalidate cache entries for this dev */
  if (sp == NIL_SUPER) return(EINVAL);

  major = (dev >> MAJOR) & BYTE;        /* major device nr */
  task = dmap[major].dmap_task; /* device task nr */
  dev_mess.m_type = DEV_CLOSE;                  /* distinguish from open */
  dev_mess.DEVICE = dev;
  (*dmap[major].dmap_close)(task, &dev_mess);

  /* Finish off the unmount. */
  sp->s_imount->i_mount = NO_MOUNT;   /* inode returns to normal */
  put_inode(sp->s_imount);      /* release the inode mounted on */
  put_inode(sp->s_isup);        /* release the root inode of the mounted fs */
  sp->s_imount = NIL_INODE;
  sp->s_dev = NO_DEV;
  sp->s_state = VALID_FS;
  return(OK);
}

PRIVATE dev_t name_to_dev(path)
char *path;                             /* pointer to path name */
{ /* Convert the block special file 'path' to a device number.  If 'path' is not a block special file, return error code in 'err_code'.*/

  register struct inode *rip;
  register dev_t dev;

  /* If 'path' can't be opened, give up immediately. */
  if ( (rip = eat_path(path)) == NIL_INODE) return(NO_DEV);

  /* If 'path' is not a block special file, return error. */
  if ( (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) {
                err_code = ENOTBLK;
                put_inode(rip);
                return(NO_DEV);
  }

  /* Extract the device number. */
  dev = (dev_t) rip->i_zone[0];
  put_inode(rip);
  return(dev);
}
```

utility functions files:

mkfs3.c

```
/* mkfs  -  make the MINIX filesystem         Authors: Tanenbaum et al. */
/*              Authors: Andy Tanenbaum, Paul Ogilvie, Frans Meulenbroeks, Bruce Evans
  Modified by Ashish Bhawsar for MyFS file system
 * This program can make version 3 (MyFS) file systems, as follows:
     mkfs -3 /dev/fd0 360    #version 3 MyFS with 360 blocks    */

#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <minix/config.h>
#include <minix/const.h>
#include <minix/type.h>
#include <minix/minlib.h>
#include "../../fs/const.h"
#if (MACHINE == IBM_PC)
```

```
#include <minix/partition.h>
#include <sys/ioctl.h>
#endif

#undef EXTERN
#define EXTERN                          /* get rid of EXTERN by making it null */
#include "../../fs/type.h"
#include "../../fs/super.h"
#include <minix/fslib.h>

#include "constant.h"

#ifndef DOS
#ifndef UNIX
#undef printf                           /* printf is a macro for printk */
#define UNIX
#endif
#endif

#ifndef PATH_MAX
#define PATH_MAX 255
#endif

#include <stdio.h>

#define INODE_MAP               2
#define MAX_TOKENS              10
#define LINE_LEN                200
#define BIN                     2
#define BINGRP                  2
#define BIT_MAP_SHIFT           13
#define N_BLOCKS                (1024L * 1024)
#define N_BLOCKS16               (128L * 1024)
#define INODE_MAX               ((unsigned) 65535)

/* You can make a really large file system on a 16-bit system, but the array of bits that get_block()/putblock() needs gets
a bit big, so we can only prefill MAX_INIT blocks.  (16-bit fsck can't check a file system larger than N_BLOCKS16
anyway.) */

#define MAX_INIT      (sizeof(char *) == 2 ? N_BLOCKS16 : N_BLOCKS)

extern char *optarg;
extern int optind;

int next_zone, next_inode, zone_size, zone_shift = 0, zoff;
block_t nrblocks;
int inode_offset, lct = 0, disk, fd, print = 0, file = 0;
unsigned int nrinodes;
int override = 0, simple = 0, dflag;
int donttest;                           /* skip test if it fits on medium */
char *progname;

long current_time, bin_time;
char zero[BLOCK_SIZE], *lastp;
char umap[MAX_INIT / 8];        /* bit map tells if block read yet */
block_t zone_map;               /* where is zone map? (depends on # inodes) */
int inodes_per_block;
int fs_version=3;               /*here we are making MyFS or version 3 File System*/
block_t max_nrblocks = N_BLOCKS;

FILE *proto;

_PROTOTYPE(int main, (int argc, char **argv));
_PROTOTYPE(block_t sizeup, (char *device));
_PROTOTYPE(void super, (zone_t zones, Ino_t inodes));
_PROTOTYPE(void rootdir, (Ino_t inode));
_PROTOTYPE(void eat_dir, (Ino_t parent));
_PROTOTYPE(void eat_file, (Ino_t inode, int f));
_PROTOTYPE(void enter_dir, (Ino_t parent, char *name, Ino_t child));
_PROTOTYPE(void incr_size, (Ino_t n, long count));
```

```
_PROTOTYPE(PRIVATE ino_t alloc_inode, (int mode, int usrid, int grpid));
_PROTOTYPE(PRIVATE zone_t alloc_zone, (void));
_PROTOTYPE(void add_zone, (Ino_t n, zone_t z, long bytes, long cur_time));
_PROTOTYPE(void add_z_3, (Ino_t n, zone_t z, long bytes, long cur_time));
_PROTOTYPE(void incr_link, (Ino_t n));
_PROTOTYPE(void insert_bit, (block_t block, int bit));
_PROTOTYPE(int mode_con, (char *p));
_PROTOTYPE(void getline, (char line[LINE_LEN], char *parse[MAX_TOKENS]));
_PROTOTYPE(void check_mtab, (char *devname));
_PROTOTYPE(long file_time, (int f));
_PROTOTYPE(void pexit, (char *s));
_PROTOTYPE(void copy, (char *from, char *to, int count));
_PROTOTYPE(void print_fs, (void));
_PROTOTYPE(int read_and_set, (block_t n));
_PROTOTYPE(void special, (char *string));
_PROTOTYPE(void get_block, (block_t n, char buf[BLOCK_SIZE]));
_PROTOTYPE(void put_block, (block_t n, char buf[BLOCK_SIZE]));
_PROTOTYPE(void cache_init, (void));
_PROTOTYPE(void flush, (void));
_PROTOTYPE(void mx_read, (int blocknr, char buf[BLOCK_SIZE]));
_PROTOTYPE(void mx_write, (int blocknr, char buf[BLOCK_SIZE]));
_PROTOTYPE(void dexit, (char *s, int sectnum, int err));
_PROTOTYPE(void usage, (void));

int main(argc, argv)
int argc;
char *argv[];
{
  int nread, mode, usrid, grpid, ch;
  block_t blocks;
  block_t i;
  ino_t root_inum;
  ino_t inodes;
  zone_t zones;
  char *token[MAX_TOKENS], line[LINE_LEN];
  struct stat statbuf;

  /* Get two times, the current time and the mod time of the binary of mkfs itself.  When the -d flag is used, the later
     time is put into the i_mtimes of all the files.  This feature is useful when producing a set of file systems, and one
     wants all the times to be identical. First you set the time of the mkfs binary to what you want, then go.  */

  current_time = time((time_t *) 0);              /* time mkfs is being run */
  stat(argv[0], &statbuf);
  bin_time = statbuf.st_mtime;  /* time when mkfs binary was last modified */

  /* Process switches. */
  progname = argv[0];
  blocks = 0;
  i = 0;
  inodes_per_block = V3_INODES_PER_BLOCK;
  max_nrblocks = N_BLOCKS;
  while ((ch = getopt(argc, argv, "1b:di:lot")) != EOF)
                switch (ch) {
                    case 'b':
                                blocks = strtoul(optarg, (char **) NULL, 0);
                                break;
                    case 'd':
                                dflag = 1;
                                current_time = bin_time;
                                break;
                    case 'i':
                                i = strtoul(optarg, (char **) NULL, 0);
                                break;
                    case 'l':  print = 1;   break;
                    case 'o':  override = 1;              break;
                    case 't':  donttest = 1;              break;
                    default:  usage();
                }

          inodes_per_block = V3_INODES_PER_BLOCK;              /* fs_version = 3;  */
```

```
/* Determine the size of the device if not specified as -b or proto. */
  if (argc - optind == 1 && blocks == 0) blocks = sizeup(argv[optind]);

  /* The remaining args must be 'special proto', or just 'special' if the block size has already been specified. */
  if (argc - optind != 2 && (argc - optind != 1 || blocks == 0)) usage();

  /* Check special. */
  check_mtab(argv[optind]);

  /* Check and start processing proto. */
  optarg = argv[++optind];
  if (optind < argc && (proto = fopen(optarg, "r")) != NULL) {
                /* Prototype file is readable. */
                lct = 1;
                getline(line, token);     /* skip boot block info */

                /* Read the line with the block and inode counts. */
                getline(line, token);
                blocks = atol(token[0]);
                if (blocks > max_nrblocks) pexit("Block count too large");
                if (sizeof(char *) == 2 && blocks > N_BLOCKS16) {
                    fprintf(stderr,
                "%s: warning: FS is larger than the %dM that fsck can check!\n",
                                progname, (int) (N_BLOCKS16 / (1024L * 1024)));
                }
                inodes = atoi(token[1]);

                /* Process mode line for root directory. */
                getline(line, token);
                mode = mode_con(token[0]);
                usrid = atoi(token[1]);
                grpid = atoi(token[2]);
  } else {
                lct = 0;
                if (optind < argc) {
                                /* Maybe the prototype file is just a size.  Check. */
                                blocks = strtoul(optarg, (char **) NULL, 0);
                                if (blocks == 0) pexit("Can't open prototype file");
                }
                if (i == 0) {
                /* The default for inodes is 3 blocks per inode, rounded up to fill an inode block.  Above 20M, the average files are
                 * sure to be larger because it is hard to fill up 20M with tiny files, so reduce the default number of inodes.  This
                 * default can always be overridden by using the -i option. */
                                i = blocks / 3;
                                if (blocks >= 20000) i = blocks / 4;
                                if (blocks >= 40000) i = blocks / 5;
                                if (blocks >= 60000) i = blocks / 6;
                                if (blocks >= 80000) i = blocks / 7;
                                if (blocks >= 100000) i = blocks / 8;
                                i += inodes_per_block - 1;
                                i = i / inodes_per_block * inodes_per_block;
                                if (i > INODE_MAX) i = INODE_MAX;
                }
                if (blocks < 5) pexit("Block count too small");
                printf("Blocks = %lu",blocks);  if (blocks > max_nrblocks) pexit("Block count too large");
                if (i < 1) pexit("Inode count too small");
                if (i > INODE_MAX) pexit("Inode count too large");
                inodes = (ino_t) i;

                /* Make simple file system of the given size, using defaults. */
                mode = 040777;
                usrid = BIN;
                grpid = BINGRP;
                simple = 1;
  }
  nrblocks = blocks;
  nrinodes = inodes;

  /* Open special. */
```

```
 special(argv[--optind]);

 if (!donttest) {
        static short testb[BLOCK_SIZE / sizeof(short)];

        /* Try writing the last block of partition or diskette. */
        lseek(fd, (off_t) (blocks - 1) * BLOCK_SIZE, SEEK_SET);
        testb[0] = 0x3245;
        testb[1] = 0x11FF;
        if (write(fd, (char *) testb, BLOCK_SIZE) != BLOCK_SIZE)
                pexit("File system is too big for minor device");
        sync();                         /* flush write, so if error next read fails */
        lseek(fd, (off_t) (blocks - 1) * BLOCK_SIZE, SEEK_SET);
        testb[0] = 0;
        testb[1] = 0;
        nread = read(fd, (char *) testb, BLOCK_SIZE);
        if (nread != BLOCK_SIZE || testb[0] != 0x3245 || testb[1] != 0x11FF)
                pexit("File system is too big for minor device");
        lseek(fd, (off_t) (blocks - 1) * BLOCK_SIZE, SEEK_SET);
        testb[0] = 0;
        testb[1] = 0;
        if (write(fd, (char *) testb, BLOCK_SIZE) != BLOCK_SIZE)
                pexit("File system is too big for minor device");
        lseek(fd, 0L, SEEK_SET);
 }


  /* Make the file-system */

cache_init();

put_block((block_t) 0, zero);       /* Write a null boot block. */

  zone_shift = 0;                   /* for future use */
  zones = nrblocks >> zone_shift;

  super(zones, inodes);

  root_inum = alloc_inode(mode, usrid, grpid);
  rootdir(root_inum);
  if (simple == 0) eat_dir(root_inum);

  print_fs();
  flush();
  return(0);
}                                   /* end main */

block_t sizeup(device)
char *device;
{
  int fd;
  struct partition entry;

  if ((fd = open(device, O_RDONLY)) == -1) return 0;
  if (ioctl(fd, DIOCGETP, &entry) == -1) entry.size = 0;
  close(fd);
  return entry.size / BLOCK_SIZE;
}

void super(zones, inodes)
zone_t zones;
ino_t inodes;
{
  unsigned int i;
  int inodeblks;
  int initblks;

  zone_t initzones, nrzones, v3sq,triple;
  zone_t zo;
  struct super_block *sup;
```

```
  char buf[BLOCK_SIZE], *cp;

  for (cp = buf; cp < &buf[BLOCK_SIZE]; cp++) *cp = 0;
  sup = (struct super_block *) buf;
  /* lint - might use a union */

  sup->s_ninodes = inodes; /* fs version is 3  */
                sup->s_nzones = 0;     /* not used in MyFS - 0 forces errors early */
                sup->s_zones = zones;

  sup->s_imap_blocks = bitmapsize((bit_t) (1 + inodes));
  sup->s_zmap_blocks = bitmapsize((bit_t) zones);
  inode_offset = sup->s_imap_blocks + sup->s_zmap_blocks + 2;
  inodeblks = (inodes + inodes_per_block - 1) / inodes_per_block;
  initblks = inode_offset + inodeblks;
  initzones = (initblks + (1 << zone_shift) - 1) >> zone_shift;
  nrzones = nrblocks >> zone_shift;
  sup->s_firstdatazone = (initblks + (1 << zone_shift) - 1) >> zone_shift;
  zoff = sup->s_firstdatazone - 1;
  sup->s_log_zone_size = zone_shift;

  /*  fs_version == 3       */
  sup->s_magic = SUPER_V3; /* identify super blocks */
  v3sq = (zone_t) V3_INDIRECTS * V3_INDIRECTS;
  triple = (zone_t) V3_INDIRECTS * V3_INDIRECTS * V3_INDIRECTS;
  zo = V3_NR_DZONES + (zone_t) V3_INDIRECTS + v3sq + triple;
  /*
  sup->s_max_size = zo * BLOCK_SIZE;
  printf("zo =%lu max size %lu",(unsigned long)zo,(unsigned long )sup->s_max_size);
  */
  sup->s_max_size = MAX_FILE_POS;

  zone_size = 1 << zone_shift;      /* nr of blocks per zone */

  put_block((block_t) 1, buf);

  /* Clear maps and inodes. */
  for (i = 2; i < initblks; i++) put_block((block_t) i, zero);

  next_zone = sup->s_firstdatazone;
  next_inode = 1;

  zone_map = INODE_MAP + sup->s_imap_blocks;

  insert_bit(zone_map, 0);            /* bit zero must always be allocated */
  insert_bit((block_t) INODE_MAP, 0);         /* inode zero not used but
                                                        * must be allocated */
}

void rootdir(inode)
ino_t inode;
{
  zone_t z;

  z = alloc_zone();
  add_zone(inode, z, 32L, current_time);
  enter_dir(inode, ".", inode);
  enter_dir(inode, "..", inode);
  incr_link(inode);
  incr_link(inode);
}

void eat_dir(parent)
ino_t parent;
{    /* Read prototype lines and set up directory. Recurse if need be. */
  char *token[MAX_TOKENS], *p;
  char line[LINE_LEN];
  int mode, usrid, grpid, maj, min, f;
  ino_t n;
  zone_t z;
```

```
  long size;

  while (1) {
        getline(line, token);
        p = token[0];
        if (*p == '$') return;
        p = token[1];
        mode = mode_con(p);
        usrid = atoi(token[2]);
        grpid = atoi(token[3]);
        if (grpid & 0200) fprintf(stderr, "A.S.Tanenbaum\n");
        n = alloc_inode(mode, usrid, grpid);

        /* Enter name in directory and update directory's size. */
        enter_dir(parent, token[0], n);
        incr_size(parent, 16L);

        /* Check to see if file is directory or special. */
        incr_link(n);
        if (*p == 'd') {
                /* This is a directory. */
                z = alloc_zone();       /* zone for new directory */
                add_zone(n, z, 32L, current_time);
                enter_dir(n, ".", n);
                enter_dir(n, "..", parent);
                incr_link(parent);
                incr_link(n);
                eat_dir(n);
        } else if (*p == 'b' || *p == 'c') {
                /* Special file. */
                maj = atoi(token[4]);
                min = atoi(token[5]);
                size = 0;
                if (token[6]) size = atoi(token[6]);
                size = BLOCK_SIZE * size;
                add_zone(n, (zone_t) ((maj << 8) | min), size, current_time);
        } else {
                /* Regular file. Go read it. */
                if ((f = open(token[4], O_RDONLY)) < 0) {
                        fprintf(stderr, "%s: Can't open %s: %s\n",
                                progname, token[4], strerror(errno));
                } else
                        eat_file(n, f);
        }
  }

}

/* Zonesize >= blocksize */
void eat_file(inode, f)
ino_t inode;
int f;
{
  int ct, i, j, k;
  zone_t z;
  char buf[BLOCK_SIZE];
  long timeval;

  do {
        for (i = 0, j = 0; i < zone_size; i++, j += ct) {
                for (k = 0; k < BLOCK_SIZE; k++) buf[k] = 0;
                if ((ct = read(f, buf, BLOCK_SIZE)) > 0) {
                        if (i == 0) z = alloc_zone();
                        put_block((z << zone_shift) + i, buf);
                }
        }
        timeval = (dflag ? current_time : file_time(f));
        if (ct) add_zone(inode, z, (long) j, timeval);
  } while (ct == BLOCK_SIZE);
  close(f);
```

```
}

void enter_dir(parent, name, child)
ino_t parent, child;
char *name;
{  /* Enter child in parent directory  Works for dir > 1 block and zone > block */
  int i, j, k, l, off;
  block_t b;
  zone_t z;
  char *p1, *p2;
  struct direct dir_entry[NR_DIR_ENTRIES];

  d3_inode ino3[V3_INODES_PER_BLOCK];

  int nr_dzones;

  b = ((parent - 1) / inodes_per_block) + inode_offset;
  off = (parent - 1) % inodes_per_block;

  get_block(b, (char *) ino3);
  nr_dzones = V3_NR_DZONES;

  for (k = 0; k < nr_dzones; k++) {
                /*                      */
                    z = ino3[off].d3_zone[k];
                            if (z == 0) {
                                    z = alloc_zone();
                                    ino3[off].d3_zone[k] = z;
     }

 for (l = 0; l < zone_size; l++) {
                            get_block((z << zone_shift) + l, (char *) dir_entry);
                            for (i = 0; i < NR_DIR_ENTRIES; i++) {
                                    if (dir_entry[i].d_ino == 0) {
                                            dir_entry[i].d_ino = child;
                                            p1 = name;
                                            p2 = dir_entry[i].d_name;
                                            j = 14;
                                            while (j--) {
                                                    *p2++ = *p1;
                                                    if (*p1 != 0) p1++;
                                            }
                                    put_block((z << zone_shift) + l, (char *) dir_entry);
                                            put_block(b, (char *) ino3);
                                            return;
                                    }
                            }
                }
  }

  printf("Directory-inode %d beyond direct blocks.  Could not enter %s\n", parent, name);
  pexit("Halt");
}


void add_zone(n, z, bytes, cur_time)
ino_t n;
zone_t z;
long bytes, cur_time;{
                add_z_3(n, z, bytes, cur_time);
}

void add_z_3(n, z, bytes, cur_time)
ino_t n;
zone_t z;
long bytes, cur_time;
{               /* Add zone z to inode n. The file has grown by 'bytes' bytes. */

  int off, i;
  block_t b;
```

```
  zone_t indir;
  zone_t blk[V3_INDIRECTS];
  d3_inode *p;
  d3_inode inode[V3_INODES_PER_BLOCK];

  b = ((n - 1) / V3_INODES_PER_BLOCK) + inode_offset;
  off = (n - 1) % V3_INODES_PER_BLOCK;
  get_block(b, (char *) inode);
  p = &inode[off];
  p->d3_size += bytes;
  p->d3_mtime = cur_time;
  for (i = 0; i < V3_NR_DZONES; i++)
        if (p->d3_zone[i] == 0) {
                p->d3_zone[i] = z;
                put_block(b, (char *) inode);
                return;
        }
  put_block(b, (char *) inode);

  /* File has grown beyond a small file. */
  if (p->d3_zone[V3_NR_DZONES] == 0) p->d3_zone[V3_NR_DZONES] = alloc_zone();
  indir = p->d3_zone[V3_NR_DZONES];
  put_block(b, (char *) inode);
  b = indir << zone_shift;
  get_block(b, (char *) blk);
  for (i = 0; i < V3_INDIRECTS; i++)
        if (blk[i] == 0) {
                blk[i] = z;
                put_block(b, (char *) blk);
                return;
        }
  pexit("File has grown beyond single indirect");
}


void incr_link(n)
ino_t n;
{
  /* Increment the link count to inode n */
  int off;
  block_t b;
  d3_inode inode3[V3_INODES_PER_BLOCK];
  b = ((n - 1) / inodes_per_block) + inode_offset;
  off = (n - 1) % inodes_per_block;

        get_block(b, (char *) inode3);
        inode3[off].d3_nlinks++;
        put_block(b, (char *) inode3);
}


void incr_size(n, count)
ino_t n;
long count;
{
  /* Increment the file-size in inode n */
  block_t b;
  int off;
  d3_inode inode3[V3_INODES_PER_BLOCK];
  b = ((n - 1) / inodes_per_block) + inode_offset;
  off = (n - 1) % inodes_per_block;

        get_block(b, (char *) inode3);
        inode3[off].d3_size += count;
        put_block(b, (char *) inode3);

}


PRIVATE ino_t alloc_inode(mode, usrid, grpid)
```

```
int mode, usrid, grpid;
{
  ino_t num;
  int off;
  block_t b;
  d3_inode inode3[V3_INODES_PER_BLOCK];

  num = next_inode++;
  if (num > nrinodes) pexit("File system does not have enough inodes");
  b = ((num - 1) / inodes_per_block) + inode_offset;
  off = (num - 1) % inodes_per_block;

  /*                        */
 get_block(b, (char *) inode3);
 inode3[off].d3_mode = mode;
 inode3[off].d3_uid = usrid;
 inode3[off].d3_gid = grpid;
 put_block(b, (char *) inode3);


  /* Set the bit in the bit map. */
  /* DEBUG FIXME.  This assumes the bit is in the first inode map block. */
  insert_bit((block_t) INODE_MAP, (int) num);
  return(num);
}

PRIVATE zone_t alloc_zone()
{               /* Allocate a new zone , Works for zone > block */
  block_t b;
  int i;
  zone_t z;

  z = next_zone++;
  b = z << zone_shift;
  if ((b + zone_size) > nrblocks)
                pexit("File system not big enough for all the files");
  for (i = 0; i < zone_size; i++)
                put_block(b + i, zero);                 /* give an empty zone */
  /* DEBUG FIXME.  This assumes the bit is in the first zone map block. */
  insert_bit(zone_map, (int) (z - zoff));
/* lint, NOT OK because * z hasn't been broken up into block + offset yet. */
  return(z);
}

void insert_bit(block, bit)
block_t block;
int bit;
{               /* Insert 'count' bits in the bitmap */
  int w, s;
  short buf[BLOCK_SIZE / sizeof(short)];

  if (block < 0) pexit("insert_bit called with negative argument");
  get_block(block, (char *) buf);
  w = bit / (8 * sizeof(short));
  s = bit % (8 * sizeof(short));
  buf[w] |= (1 << s);
  put_block(block, (char *) buf);
}

int mode_con(p)
char *p;
{               /* Convert string to mode */
  int o1, o2, o3, mode;
  char c1, c2, c3;

  c1 = *p++;
  c2 = *p++;
  c3 = *p++;
  o1 = *p++ - '0';
  o2 = *p++ - '0';
```

```
  o3 = *p++ - '0';
  mode = (o1 << 6) | (o2 << 3) | o3;
  if (c1 == 'd') mode += I_DIRECTORY;
  if (c1 == 'b') mode += I_BLOCK_SPECIAL;
  if (c1 == 'c') mode += I_CHAR_SPECIAL;
  if (c1 == '-') mode += I_REGULAR;
  if (c2 == 'u') mode += I_SET_UID_BIT;
  if (c3 == 'g') mode += I_SET_GID_BIT;
  return(mode);
}

void getline(line, parse)
char *parse[MAX_TOKENS];
char line[LINE_LEN];
{               /* Read a line and break it up in tokens */
  int k;
  char c, *p;
  int d;

  for (k = 0; k < MAX_TOKENS; k++) parse[k] = 0;
  for (k = 0; k < LINE_LEN; k++) line[k] = 0;
  k = 0;
  parse[0] = 0;
  p = line;
  while (1) {
                if (++k > LINE_LEN) pexit("Line too long");
                d = fgetc(proto);
                if (d == EOF) pexit("Unexpected end-of-file");
                *p = d;
                if (*p == '\n') lct++;
                if (*p == ' ' || *p == '\t') *p = 0;
                if (*p == '\n') {
                                *p++ = 0;
                                *p = '\n';
                                break;
                }
                p++;
  }

  k = 0;
  p = line;
  lastp = line;
  while (1) {
                c = *p++;
                if (c == '\n') return;
                if (c == 0) continue;
                parse[k++] = p - 1;
                do {
                                c = *p++;
                } while (c != 0 && c != '\n');
  }
}

void check_mtab(devname)
char *devname;                                  /* /dev/hd1 or whatever */
{               /* Check to see if the special file named in s is mounted. */

  int n;
  char special[PATH_MAX + 1], mounted_on[PATH_MAX + 1], version[10], rw_flag[10];

  if (load_mtab("mkfs") < 0) return;
  while (1) {
                n = get_mtab_entry(special, mounted_on, version, rw_flag);
                if (n < 0) return;
                if (strcmp(devname, special) == 0) {
                                /* Can't mkfs on top of a mounted file system. */
                                fprintf(stderr, "%s: %s is mounted on %s\n",
                                                progname, devname, mounted_on);
                                exit(1);
                }
```

```
  }
}

long file_time(f)
int f;
{
  struct stat statbuf;
  fstat(f, &statbuf);
  return(statbuf.st_mtime);
}


void pexit(s)
char *s;
{
  fprintf(stderr, "%s: %s\n", progname, s);
  if (lct != 0)
                fprintf(stderr, "Line %d being processed when error detected.\n", lct);
  flush();
  exit(2);
}

void copy(from, to, count)
char *from, *to;
int count;
{
  while (count--) *to++ = *from++;
}


void print_fs()
{
  int i, j;
  ino_t k;
  d3_inode inode3[V3_INODES_PER_BLOCK];

  unsigned short usbuf[BLOCK_SIZE / sizeof(unsigned short)];
  block_t b, inode_limit;
  struct direct dir[NR_DIR_ENTRIES];

  get_block((block_t) 1, (char *) usbuf);
  printf("\nSuperblock: ");
  for (i = 0; i < 8; i++) printf("%06o ", usbuf[i]);
  get_block((block_t) 2, (char *) usbuf);
  printf("...\nInode map:  ");
  for (i = 0; i < 9; i++) printf("%06o ", usbuf[i]);
  get_block((block_t) 3, (char *) usbuf);
  printf("...\nZone  map:  ");
  for (i = 0; i < 9; i++) printf("%06o ", usbuf[i]);
  printf("...\n");

  k = 0;
  for (b = inode_offset; k < nrinodes; b++) {
                        get_block(b, (char *) inode3);

                for (i = 0; i < inodes_per_block; i++) {
                        k = inodes_per_block * (int) (b - inode_offset) + i + 1;
                        /* Lint but OK */
                        if (k > nrinodes) break;

                                if (inode3[i].d3_mode != 0) {
                                        printf("Inode %2d:  mode=", k);
                                        printf("%06o", inode3[i].d3_mode);
                                        printf("  uid=%2d  gid=%2d  size=",
                                        inode3[i].d3_uid, inode3[i].d3_gid);
                                        printf("%6ld", inode3[i].d3_size);
                                        printf("  zone[0]=%ld\n", inode3[i].d3_zone[0]);
                                }
                                if ((inode3[i].d3_mode & I_TYPE) == I_DIRECTORY) {
                                        /* This is a directory */
```

```
                                        get_block(inode3[i].d3_zone[0], (char *) dir);
                                        for (j = 0; j < NR_DIR_ENTRIES; j++)
                                                if (dir[j].d_ino)
                                printf("\tInode %2d: %s\n", dir[j].d_ino, dir[j].d_name);
                                }

                }
  }

  printf("%d inodes used.  %d zones used.\n", next_inode - 1, next_zone);
}


int read_and_set(n)
block_t n;
{
/* The first time a block is read, it returns all 0s, unless there has
 * been a write.  This routine checks to see if a block has been accessed.
 */

  int w, s, mask, r;

  if (sizeof(char *) == 2 && n >= MAX_INIT) pexit("can't initialize past 128M");
  w = n / 8;
  s = n % 8;
  mask = 1 << s;
  r = (umap[w] & mask ? 1 : 0);
  umap[w] |= mask;
  return(r);
}

void usage()
{
  fprintf(stderr,
                "Usage: %s [-1dlot] [-b blocks] [-i inodes] special [proto]\n",
                progname);
  exit(1);
}

void special(string)
char *string;
{
  fd = creat(string, 0777);
  close(fd);
  fd = open(string, O_RDWR);
  if (fd < 0) pexit("Can't open special file");
}



void get_block(n, buf)
block_t n;
char buf[BLOCK_SIZE];
{               /* Read a block. */

  int k;

  /* First access returns a zero block */
  if (read_and_set(n) == 0) {
                copy(zero, buf, BLOCK_SIZE);
                return;
  }
  lseek(fd, (off_t) n * BLOCK_SIZE, SEEK_SET);
  k = read(fd, buf, BLOCK_SIZE);
  if (k != BLOCK_SIZE) {
                pexit("get_block couldn't read");
  }
}

void put_block(n, buf)
```

```
block_t n;
char buf[BLOCK_SIZE];
{
/* Write a block. */

  (void) read_and_set(n);

 /* XXX - check other lseeks too. */
  if (lseek(fd, (off_t) n * BLOCK_SIZE, SEEK_SET) == (off_t) -1) {
                pexit("put_block couldn't seek");
  }
  if (write(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
                pexit("put_block couldn't write");
  }
}


/* Dummy routines to keep source file clean from #ifdefs */

void flush()
{
  return;
}

void cache_init()
{
  return;
}
```

fsck3.c

```
#define INODES_PER_BLOCK        V3_INODES_PER_BLOCK
#define INODE_SIZE ((int)       V3_INODE_SIZE)
#define WORDS_PER_BLOCK         (BLOCK_SIZE / (int) sizeof(bitchunk_t))
#define MAX_ZONES               (V3_NR_DZONES+V3_INDIRECTS+(long)V3_INDIRECTS*V3_INDIRECTS)
#define NR_DZONE_NUM            V3_NR_DZONES
#define NR_INDIRECTS            V3_INDIRECTS
#define NR_ZONE_NUMS            V3_NR_TZONES
#define ZONE_NUM_SIZE           V3_ZONE_NUM_SIZE
#define bit_nr bit_t
#define block_nr block_t
#define d_inode d3_inode
#define d_inum d_ino
#define dir_struct struct direct
#define i_mode d3_mode
#define i_nlinks d3_nlinks
#define i_size d3_size
#define i_zone d3_zone
#define zone_nr zone_t

#define NAME_MAX 14
#define LINK_MAX 127

#include <sys/types.h>
#include <sys/dir.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <minix/config.h>
#include <minix/const.h>
#include <minix/type.h>
#include "../../fs/const.h"
#include "../../fs/inode.h"
#include "../../fs/type.h"
```

```
#undef printf                           /* defined as printk in "../fs/const.h" */

#include <stdio.h>

#define BITSHIFT        4       /* = log2(#bits(int)) */
#define MAXPRINT        8       /* max. number of error lines in chkmap */
#define MAXDIRSIZE      5000    /* max. size of a reasonable directory */
#define CINDIR          128     /* number of indirect zno's read at a time */
#define CDIRECT          16     /* number of dir entries read at a time */

/* Macros for handling bitmaps.  Now bit_t is long, these are bulky and the type demotions produce a lot of lint.  The
explicit demotion in POWEROFBIT is for efficiency and assumes 2's complement ints.  Lint should be clever enough
not to warn about it since BITMASK is small, but isn't.  (It would be easier to get right if bit_t was was unsigned (long)
since then there would be no danger from wierd sign representations.  Lint doesn't know we only use non-negative bit
numbers.) There will usually be an implicit demotion when WORDOFBIT is used as an array index.  This should be
safe since memory for bitmaps will run out first. */

#define BITMASK         ((1 << BITSHIFT) - 1)
#define WORDOFBIT(b)    ((b) >> BITSHIFT)
#define POWEROFBIT(b)   (1 << ((int) (b) & BITMASK))
#define setbit(w, b)    (w[WORDOFBIT(b)] |= POWEROFBIT(b))
#define clrbit(w, b)    (w[WORDOFBIT(b)] &= ~POWEROFBIT(b))
#define bitset(w, b)    (w[WORDOFBIT(b)] & POWEROFBIT(b))

#define ZONE_CT         360     /* default zones  (when making file system) */
#define INODE_CT        95      /* default inodes (when making file system) */

#include "../../fs/super.h"
struct super_block sb;

#define STICKY_BIT01000         /* not defined anywhere else */

/* Ztob gives the block address of a zone btoa gives the byte address of a block */
#define ztob(z)         ((block_nr) (z) << sb.s_log_zone_size)
#define btoa(b)         ((long) (b) * BLOCK_SIZE)
#define SCALE           ((int) ztob(1))         /* # blocks in a zone */
#define FIRST           ((zone_nr) sb.s_firstdatazone)      /* as the name says */

/* # blocks of each type */
#define N_SUPER         1
#define N_IMAP          (sb.s_imap_blocks)
#define N_ZMAP          (sb.s_zmap_blocks)
#define N_ILIST         ((sb.s_ninodes+INODES_PER_BLOCK-1) / INODES_PER_BLOCK)
#define N_DATA          (sb.s_zones - FIRST)

/* Block address of each type */
#define BLK_SUPER       (SUPER_BLOCK)
#define BLK_IMAP        (BLK_SUPER + N_SUPER)
#define BLK_ZMAP        (BLK_IMAP  + N_IMAP)
#define BLK_ILIST       (BLK_ZMAP  + N_ZMAP)
#define BLK_FIRST       ztob(FIRST)
#define ZONE_SIZE       ((int) ztob(BLOCK_SIZE))
#define NLEVEL          (NR_ZONE_NUMS - NR_DZONE_NUM + 1)

/* Byte address of a zone/of an inode */
#define zaddr(z)        btoa(ztob(z))
#define inoaddr(i)      ((long) (i - 1) * INODE_SIZE + (long) btoa(BLK_ILIST))
#define INDCHUNK        ((int) (CINDIR * ZONE_NUM_SIZE))
#define DIRCHUNK        ((int) (CDIRECT * DIR_ENTRY_SIZE))

char *prog, *device;            /* program name (fsck), device name */
int firstcnterr;                /* is this the first inode ref cnt error? */
bitchunk_t *imap, *spec_imap;   /* inode bit maps */
bitchunk_t *zmap, *spec_zmap;   /* zone bit maps */
bitchunk_t *dirmap;             /* directory (inode) bit map */
char rwbuf[BLOCK_SIZE];         /* one block buffer cache */
block_nr thisblk;               /* block in buffer cache */
char nullbuf[BLOCK_SIZE];       /* null buffer */
nlink_t *count;                 /* inode count */
int changed;                    /* has the diskette been written to? */
```

```
struct stack {
        dir_struct *st_dir;
        struct stack *st_next;
        char st_presence;
} *ftop;

int dev;                        /* file descriptor of the device */

#define DOT                     1
#define DOTDOT                  2

/* Counters for each type of inode/zone. */
int nfreeinode, nregular, ndirectory, nblkspec, ncharspec, nbadinode;
int npipe, nsyml, ztype[NLEVEL];
long nfreezone;

int repair, automatic, listing, listsuper;      /* flags */
int firstlist;                                  /* has the listing header been printed? */
unsigned part_offset;                           /* sector offset for this partition */
char answer[] = "Answer questions with y or n.  Then hit RETURN";

_PROTOTYPE(int main, (int argc, char **argv));
_PROTOTYPE(void initvars, (void));
_PROTOTYPE(void fatal, (char *s));
_PROTOTYPE(int eoln, (int c));
_PROTOTYPE(int yes, (char *question));
_PROTOTYPE(int atoo, (char *s));
_PROTOTYPE(int input, (char *buf, int size));
_PROTOTYPE(char *alloc, (unsigned nelem, unsigned elsize));
_PROTOTYPE(void printname, (char *s));
_PROTOTYPE(void printrec, (struct stack *sp));
_PROTOTYPE(void printpath, (int mode, int nlcr));
_PROTOTYPE(void devopen, (void));
_PROTOTYPE(void devclose, (void));
_PROTOTYPE(void devio, (block_nr bno, int dir));
_PROTOTYPE(void devread, (long offset, char *buf, int size));
_PROTOTYPE(void devwrite, (long offset, char *buf, int size));
_PROTOTYPE(void pr, (char *fmt, int cnt, char *s, char *p));
_PROTOTYPE(void lpr, (char *fmt, long cnt, char *s, char *p));
_PROTOTYPE(bit_nr getnumber, (char *s));
_PROTOTYPE(char **getlist, (char ***argv, char *type));
_PROTOTYPE(void lsuper, (void));
_PROTOTYPE(void getsuper, (void));
_PROTOTYPE(void chksuper, (void));
_PROTOTYPE(void lsi, (char **clist));
_PROTOTYPE(bitchunk_t *allocbitmap, (int nblk));
_PROTOTYPE(void loadbitmap, (bitchunk_t *bitmap, block_nr bno, int nblk));
_PROTOTYPE(void dumpbitmap, (bitchunk_t *bitmap, block_nr bno, int nblk));
_PROTOTYPE(void fillbitmap, (bitchunk_t *bitmap, bit_nr lwb, bit_nr upb, char **list));
_PROTOTYPE(void freebitmap, (bitchunk_t *p));
_PROTOTYPE(void getbitmaps, (void));
_PROTOTYPE(void putbitmaps, (void));
_PROTOTYPE(void chkword, (unsigned w1, unsigned w2, bit_nr bit, char *type, int *n, int *report));
_PROTOTYPE(void chkmap, (bitchunk_t *cmap, bitchunk_t *dmap, bit_nr bit, block_nr blkno, int nblk, char *type));
_PROTOTYPE(void chkilist, (void));
_PROTOTYPE(void getcount, (void));
_PROTOTYPE(void counterror, (Ino_t ino));
_PROTOTYPE(void chkcount, (void));
_PROTOTYPE(void freecount, (void));
_PROTOTYPE(void printperm, (Mode_t mode, int shift, int special, int overlay));
_PROTOTYPE(void list, (Ino_t ino, d_inode *ip));
_PROTOTYPE(int Remove, (dir_struct *dp));
_PROTOTYPE(void make_printable_name, (char *dst, char *src, int n));
_PROTOTYPE(int chkdots, (Ino_t ino, off_t pos, dir_struct *dp, Ino_t exp));
_PROTOTYPE(int chkname, (Ino_t ino, dir_struct *dp));
_PROTOTYPE(int chkentry, (Ino_t ino, off_t pos, dir_struct *dp));
_PROTOTYPE(int chkdirzone, (Ino_t ino, d_inode *ip, off_t pos, zone_nr zno));
_PROTOTYPE(void errzone, (char *mess, zone_nr zno, int level, off_t pos));
_PROTOTYPE(int markzone, (zone_nr zno, int level, off_t pos));
_PROTOTYPE(int chkindzone, (Ino_t ino, d_inode *ip, off_t *pos, zone_nr zno, int level));
```

```
_PROTOTYPE(off_t jump, (int level));
_PROTOTYPE(int zonechk, (Ino_t ino, d_inode *ip, off_t *pos, zone_nr zno, int level));
_PROTOTYPE(int chkzones, (Ino_t ino, d_inode *ip, off_t *pos, zone_nr *zlist, int len, int level));
_PROTOTYPE(int chkfile, (Ino_t ino, d_inode *ip));
_PROTOTYPE(int chkdirectory, (Ino_t ino, d_inode *ip));
_PROTOTYPE(int chklink, (Ino_t ino, d_inode *ip));
_PROTOTYPE(int chkspecial, (Ino_t ino, d_inode *ip));
_PROTOTYPE(int chkmode, (Ino_t ino, d_inode *ip));
_PROTOTYPE(int chkinode, (Ino_t ino, d_inode *ip));
_PROTOTYPE(int descendtree, (dir_struct *dp));
_PROTOTYPE(void chktree, (void));
_PROTOTYPE(void printtotal, (void));
_PROTOTYPE(void chkdev, (char *f, char **clist, char **ilist, char **zlist));

int main(argc, argv)
int argc;
char **argv;
{
 register char **clist = 0, **ilist = 0, **zlist = 0;

 register devgiven = 0;
 register char *arg;
 time_t last_time, current=0;

 if ((1 << BITSHIFT) != 8 * sizeof(bitchunk_t)) {
        printf("Fsck was compiled with the wrong BITSHIFT!\n");
        exit(1);
 }

 sync();
 prog = *argv++;
 while ((arg = *argv++) != 0)
        if (arg[0] == '-' && arg[1] != 0 && arg[2] == 0) switch (arg[1]) {
                case 'a':  automatic ^= 1;          break;
                case 'c':
                        clist = getlist(&argv, "inode");
                        break;
                case 'i':
                        ilist = getlist(&argv, "inode");
                        break;
                case 'z':
                        zlist = getlist(&argv, "zone");
                        break;
                case 'r':  repair ^= 1;             break;
                case 'l':  listing ^= 1;            break;
                case 's':  listsuper ^= 1;          break;
                default:
                        printf("%s: unknown flag '%s'\n", prog, arg);
                }
        else {
                chkdev(arg, clist, ilist, zlist);
                clist = 0;
                ilist = 0;
                zlist = 0;
                devgiven = 1;
        }
 if (!devgiven) {
        printf("Usage: fsck [-acilrsz] file\n");
        exit(1);
 }
 return(0);
}

/* Check the super block for reasonable contents. */
void chksuper()
{
 register n;
 register off_t maxsize;
 time_t current=0;
```

```
  n = bitmapsize((bit_t) sb.s_ninodes + 1);
  if (sb.s_magic != SUPER_V3) fatal("bad magic number in super block");
  if (sb.s_imap_blocks < n) fatal("too few imap blocks");
  if (sb.s_imap_blocks != n) {
                pr("warning: expected %d imap_block%s", n, "", "s");
                printf(" instead of %d\n", sb.s_imap_blocks);
  }
  current = time(&current);

  if(sb.s_state == VALID_FS) {
     printf("MyFS file system All ready Cleaned");
     if(current - sb.s_last_check < sb.s_checkintervals)
         exit(0);
  }else
     printf("Checking MyFS file system");
  sb.s_last_check = current;
  sb.s_state = VALID_FS;

  n = bitmapsize((bit_t) sb.s_zones);
  if (sb.s_zmap_blocks < n) fatal("too few zmap blocks");
  if (sb.s_zmap_blocks != n) {
                pr("warning: expected %d zmap_block%s", n, "", "s");
                printf(" instead of %d\n", sb.s_zmap_blocks);
  }
  if (sb.s_firstdatazone >= sb.s_zones)
                fatal("first data zone too large");
  if (sb.s_log_zone_size >= 8 * sizeof(block_nr))
                fatal("log_zone_size too large");
  if (sb.s_log_zone_size > 8) printf("warning: large log_zone_size (%d)\n",
                        sb.s_log_zone_size);
  n = (BLK_ILIST + N_ILIST + SCALE - 1) >> sb.s_log_zone_size;
  if (sb.s_firstdatazone < n) fatal("first data zone too small");
  if (sb.s_firstdatazone != n) {
                printf("warning: expected first data zone to be %d ", n);
                printf("instead of %u\n", sb.s_firstdatazone);
  }
  maxsize = MAX_FILE_POS;
  if (((maxsize - 1) >> sb.s_log_zone_size) / BLOCK_SIZE >= MAX_ZONES)
                maxsize = ((long) MAX_ZONES * BLOCK_SIZE) << sb.s_log_zone_size;
  if (sb.s_max_size != maxsize) {
                printf("warning: expected max size to be %ld ", maxsize);
                printf("instead of %ld\n", sb.s_max_size);
  }
 }
```

REFERENCES

References:

[1] Andrew S. Tanenbaum, Albert S. Woodhull : Operating Systems Design and Implementation, Second Edition, Pearson Education, 2004.

[2] Daniel P. Bovet, Marco Cesati : Understanding the Linux Kernel, Second Edition, O'Reilly, 2004.

[3] Maurice J. Bach : The Design of the UNIX Operating System, PHI, 1986.

[4] Andrew S. Tanenbaum, Marteen V. Steen : "Distributed Systems Principles and Paradigm", Pearson Education.

[5] Peter Gutmann : "*Secure Deletion of Data from Magnetic and Solid-State Memory"* from Department of Computer Science, University of Auckland, published in the Sixth USENIX Security Symposium Proceedings, San Jose, California, July, 1996

[6] K. Thompson: "Unix Implementation", Bell System Technical Journal, Vol. 57, July-Aug. 1978.

[7] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. The Berkeley Fast Filesystem. In The Design and Implementation of the 4.4BSD Operating System, pages 269–84. Addison-Wesley, May 1996.

[8] *Partial Replication in the Vesta Software Repository* from SRC Research Report 172, Systems Research Center, COMPAQ Systems Research Center ,Palo Alto, California 94301

[9] Heidemann, John S. and Gerald J. Popek. "A Layered Approach to File System Development." UCLA Computer Science Department Technical Report (1991).

[9] The Linux Kernel: Blueprints for World Domination , Gary Lawrence Murphy, Principle Consultant TCI Ontario, Macmillan Computer Publishing

[10] *The Cedar file system*, ACM Volume 31 , March 1988, Authors David K. Gifford Laboratory for Computer Science, Cambridge, MA, Roger M. Needham Computer Laboratory, Cambridge, UK and Michael D. Schroeder Digital Equipment Corporation, Palo Alto, CA.

[11] IEEE: Information Technology- Portable Operating System Interface (POSIX), part 1: System Application Program Interface (API) [C Language], New York: Institute of Electrical & Electronics Engineers, Inc., 1990.

[12] *The Performance of µ-Kernel-Based Systems*, Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, Jean Wolter, 16th ACM Symposium on Operating Systems Principles 1997, Saint-Malo, France

[13] "Emergency Destruction of Information Storing Media", M.Slusarczuk et al, Institute for Defense Analyses, December 1987.

[14] "A Guide to Understanding Data Remanence in Automated Information Systems", National Computer Security Centre, September 1991.

[15] "Detection of Digital Information from Erased Magnetic Disks", Venugopal Veeravalli, Masters thesis, Carnegie-Mellon University, 1987.

[16] J. Liedtke. Toward real µ-kernels. Communications of the ACM, 39(9):70-77, September 1996.

[17] *A DOS/Linux Extensible File System*, Mohammed Mohy El Din Mahmoud and Amr El-Kadi Department of Computer Science, The american University in Cairo, IEEE 1997

[18] *A Fast File System for UNJX*, McKusic, Kirk. ACM Transactions on Computer Systems, no. 3 (August 1984)

[17] Vijay Mukhi: UNIX- The Open-Boundless C, BPB Publications.

[18] Richard Stones, Neil Matthew: Beginning Linux Programming, WROX Publishers.

[19] Sumitabh Das : Unix – Concepts and Applications, TATA McGRAW HILL.

[20] Abraham Silberschatz, Peter Baer Galvin : Operating System Concepts, Fifth Edition, Addison-Wesley, 1999.

[21] W.R. Stevens: “Advanced Programming in the UNIX Environment”, Addison-Wesley, 1992.

[22] W. Stallings: Operating Systems, Second Edition, Prentice Hall, 1995.

[23] U. Vahalia: UNIX Internals-The New Frontiers, Prentice Hall, 1996.

[24] D. Lewine: POSIX Programmer's Guide, O'Reilly & Associates, 1991.

Web links

[25] http://www.cs.vu.nl/minix/ : MINIX home Page.

[26] http://world.std.com/~bochs :home page of Bochs, an 80386 emulator.

[27] http://os.inf.tu-dresden.de/pubs/sosp97/ : The Performance of µ-Kernel-Based Systems

http://en.wikipedia.org/wiki/L4 : Information of microkernel_family.
file:///D:/wiki/History_of_operating_systems: For History of operating systems.
http://www.gnu.org/philosophy/linux-gnu-freedom.html : open source.

[28] http://www.linuxdoc.org/guides.html : Linux Kernel Internals.

[29] http://world.std.com/~bochs :home page of Bochs, an 80386 emulator.

[30]http://www.research.compaq.com/SRC/