Directory Management in FS: Design and Implementation

A Dissertation

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

MASTER OF ENGINEERING
(COMPUTER TECHNOLOGY & APPLICATIONS)

By

CHANDAN SINGH
College Roll No. 11/CTA/03
Delhi University Roll No. 3011

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 “**Directory Management in FS: Design and Implementation**” submitted by **Chandan Singh** 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	**Professor Asok De**
Head of Department	Head of Department
Department of Computer Engineering	Department of Information Technology
Delhi College of Engineering	Delhi College of Engineering
Delhi	Delhi

ACKNOWLEDGEMENT

I would like to thank all those people who made this project a success.

First of all, I owe this moment of satisfaction with a deep sense of gratitude to **Prof. Asok De**, Head of Department, Department of Information Technology, my project guide, who sowed the seed for this project and motivated me to go for it. His interest, skillful continuous guidance and constant supervision at every stage of the project, made it possible to complete without which nothing would have been possible.

I would like to express my gratitude towards Prof. D. Roy Choudhury, Head of Department, Department of Computer Engineering, Dr. Goldie Gabrani, Assistant Professor, Department of Computer Engineering, Mrs. Rajni Jindal, Lecturer, Department of Information Technology, Dr. S. K. Saxena, Lecturer, Department of Computer Engineering, Mr. Rajiv Kumar, Department of Computer Engineering for their constant support, encouragement and guidance.

I would like to extend my thanks and gratitude to everyone associated with Department of Computer Engineering for helping in any way in the completion of the project.

I would like to thank my friends and classmates for their unconditional support and motivation during this project.

Chandan Singh
Master of Engineering
Computer Technology & Applications
College Roll No. 11/CTA/03
Delhi University Roll No. 3011

ABSTRACT

The prime objective of this project is to provide, design and implementation of the directory management policy in a file system. This design is for Minix operating system, which is build on microkernel approach. The Minix file system is older than most of the modern file system, however, presumed to be the most reliable and compatible with UNIX from the user's point of view, though completely different inside.

In this dissertation, design of a new directory structure for the Minix file system has been proposed. Furthermore, the proposed directory structure has been implemented on the Minix file system. Since, Minix file system does not use directory buffer cache so, this dissertation offers implementation of a directory buffer cache for speedy access of the directory entries. The implementation of directory buffer cache helps the operating system to lower the disk-reads for directory entries by a significant amount.

In addition, it provides a method to construct a fast symbolic link for a file, if the target files and source files are located on two different devices. The modified file system has different structure compared to old file system, so, to write a new file system, a utility has been designed and implemented.

CONTENTS

CHAPTER 1 INTRODUCTION

1.1 Basic Concepts

File system is an important part of an operating system. It is the most visible aspect of an operating system. But before we delve into the problem definition and description, we should review some basic terms and their definitions.

1.1.1 Operating System

Operating system is suitably defined as:

" *It is a program that acts as an intermediary between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient an efficient manner*."[1]

It is a most fundamental system program, which controls all the computer's resources and provides the base upon which the application programs can be written. Operating systems are primarily resource managers; the main resource they manage is computer hardware in the form of processors, storage, input/output devices, communication devices and data.[2]

Thus, we can say that operating system is a software that ensures the correct operation of the computer system besides managing the resources. Operating system, on the basis of design, can broadly classified into following three categories:

- Monolithic system,
- Layered system, and
- Client-server model

Monolithic system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. In terms of information hiding, there is essentially none i.e. every procedure is visible to every other procedure. The result is a huge, monolithic program that is run in a single address space. It has a main program that invokes the requested service procedures, which in turn carry out the system calls. In addition, it has a set of utility procedure that help the service procedures. The main problem with this system is, there is no clear boundary between different modules e.g. kernel, memory module, file module etc. Somebody cannot change the one functionality of the system without changing others since they are interconnected. It requires full compilation and reinstallation for adaptation of the system. Figure 1-1 shows the example

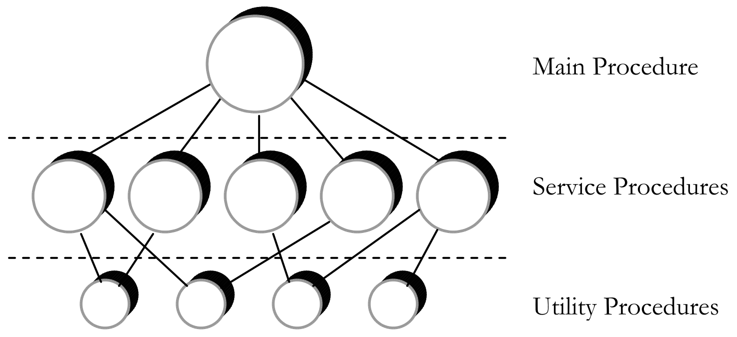


Figure 1-1. Structure of a monolithic system

of monolithic system.

Layered system is organized as a hierarchy of layers. Each one constructed upon the one below it. Each layer has specific task to perform. When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to make the equivalent of a system call. Other procedure or layer can be implemented without affecting the lower level functionality. This scheme is slightly similar to the scheme employed in UNIX operating system. This scheme was used inside the THE operating system. Figure 1-2 displays the

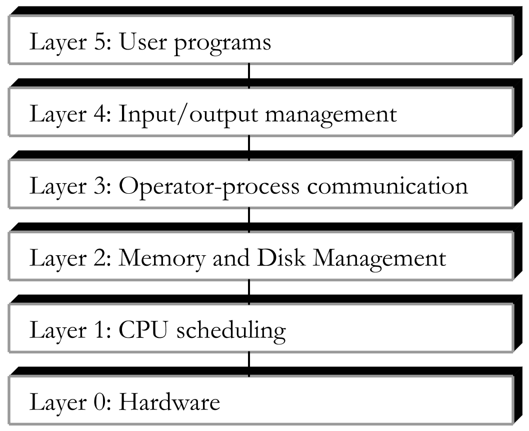

Layer 5: User programs
Layer 4: Input/output management
Layer 3: Operator-process communication
Layer 2: Memory and Disk Management
Layer 1: CPU scheduling
Layer 0: Hardware

Figure 1-2. Structure of a Layered system (THE operating system)[1]

THE operating system structure.

Client-server model has most of the operating system functions in user processes. That is all the code is move up into higher layers and remove as much as possible from the operating system, leaving a minimal kernel, called microkernel. All the kernel does is handle the communication between client procedure and server procedure. Thus, operating system splits into parts e.g. file service, terminal service, memory service etc. each of which only handles one facet of the system. As a consequence, if a bug in the one server is triggered, will not bring the whole machine down. This scheme is used inside MINIX operating system.

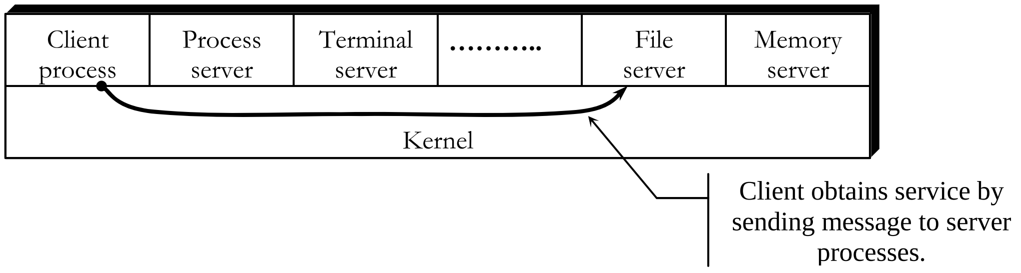


Figure 1-3. Structure of a client-server model

1.1.2 File System

Computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. For some applications, this size is adequate, but for others, it is far too small. Another problem with keeping information within a process' address space is that when the process terminates, the information is lost.

The usual solution to all these problems is to store information on disks and other nonvolatile external media, so the contents are persistent through power failures and system reboots. Computers can store information on several different storage media. The operating system provides a uniform logical view of information storage, so that the computer system will be convenient to use. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the *file*. Files are managed and mapped, by the operating system, onto physical devices. How they are structured, named, accessed, used, protected, and implemented are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the *file system*.

File system provides the mechanism for storage of and access to both data and programs of the operating system on disks or external media. The file system primarily consists of two distinct parts:

- Files and
- Directory structure

A file is defined, as "*A file is simply an ordered sequence of elements, where an element could be a machine word, a character, or a bit, depending upon the implementation. A user may create, modify or delete files only through the use of the file system*"[3] or a file is a named collection of related information. From the designer's point of view, a file is a sequence of bits, bytes, lines, or records, whose meaning is defined by the file's creator and user. A file is *named*, for the convenience of users, and is referred to by its symbolic name. A name is usually a string of characters. The exact rules for file naming vary somewhat from system to system.

A directory structure is used to organize and provide information about all the files in the system using directory entries. Directory may be termed, as "*directory is a special file which is maintained by the file system, and which contains a list of entries. To a user, an entry appears to be a file and is accessed in terms of its symbolic entry name, which is the user's file name*"[3]. An entry name need be unique only within the directory in which it occurs. In reality, each entry is a pointer of one of two kinds. The entry may point directly to a file, (which may itself be a directory) which is stored in secondary storage, or else it may point to another entry in the same or another directory. An entry, which points directly to a file, is called a *branch*, while an entry, which points to another directory entry is called a *link*.

Each branch contains a description of the way in which it may be used and of the way in which it is being used. This description includes information such as the actual physical address of the file, the time this file was created or last modified, the time the file was last referred to, and access control information for the branch. The description also includes the current state of the file. Some of this information is unavailable to the user.

A *link* always eventually points to a branch, although possibly via a chain of links to the branch and thence to a file. Thus, the link and the branch both effectively point to the file. The only information associated with a *link* is the pointer to the entry to which it links. This pointer is specified in terms of a symbolic name, which uniquely identifies the linked entry within the hierarchy. A link derives its access control information from the branch to which it effectively points. In general, a user will usually not need to know whether a given entry is a branch or a link, however, user easily may find out. Links are of following types:

- Hard link,
- Symbolic link, and
- Fast symbolic link

We will look into detail about the above given terms in next chapter.

1.2 Problem Description

To implement a new directory structure and directory management policy, the first issue involved is to select the suitable operating system. Moreover, the related aspect for the consideration is, operating system should be open source i.e. the source code of the operating system is available so that user may reject, accept, modify or extend the code. As well as, it must have a precise and modular implementation for further modification; otherwise, it will be just like a minefield. Besides that, it must be free from any copyright and licensing restriction, which prevents the redistribution and use of the modified code.

There are quite a few open source operating systems are available. However, considering the above-mentioned issues, the obvious choice is to select, the most admired and well-known open source Linux as target operating system. Linux resembles any other traditional, nonmicrokernel UNIX implementation. Developed by Linus Torvalds in 1991 from Minix, a simplified version of Unix intended for educational use, Linux has grown from a mere 10,000-line kernel to a full-featured operating system with more than 1.5 million lines of code. The number of Linux users has expanded at a similar rate, blossoming from fewer than 100 in 1991 to more than 7 million in 1998 [4]. Early in its development, Linux's source code was made available for free on the Internet. As a result, its history has been one of collaboration by many users from all around the world. From an initial kernel, Linux has grown to include evermore UNIX functionality [1].

Initially, Linux was originally programmed with a Minix-compatible file system, to ease exchanging data with the Minix development system. In later designs, the Minix file system was superseded by new modified file systems. Currently, Linux uses extended-file-system, *extfs*, which has been matured from a small file system to a very large file system. At present, it has a colossal structure. Since, Linux is nonmicrokernel based operating system, it needs recompilation of whole operating system for any change to come in effect. In addition, it has many other modules that may be affected by the changes, if applied across the file system module. And so, confronting with errors looming up inside other modules will not be of any help. These errors will deviate us from our main objective of this dissertation.

As mentioned above, Minix file system was the first file system for Linux operating system. Minix operating system is similar to UNIX in structure from user's point of view. It had been developed by Andrew S. Tanenbaum in the mid-1980s [5]. This system avoids licensing restrictions, so it can be used to study and research purpose. The Minix operating system is consist of about 50,000 lines of C code including few hundreds of code written in assembly language for minimal operation. The development of Minix is an ongoing proposition. Another key feature of the Minix operating system is its client-server or microkernel structure. Microkernel-based system, in which most of the operating systems runs as separate processes, mostly outside the kernel. They communicate by message passing. The kernel's job is to handle the message passing, interrupt handling, low-level process management, and possibly the input/output. Among the people who actually design operating systems, the 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”[6]. In this structure, it is relatively easy to replace a module without having to recompile or reinstall the entire system, as mentioned in earlier sections.

Minix file system has about 10,000 lines of C code. However, the Minix file system is quite limited in features and restricted in capabilities e.g. it was severely restricted by fourteen character file name limits. It has been revised a few times to enhance its features since it was first introduced. It has a very concise code, which is based on POSIX standard. It would be ideal to select this file system as target file system.

Minix does not use directory entries of variable name-length and so suffer from internal fragmentation inside directory block. It causes wastage of storage space. Fourteen character name-length constraint is, sometime, not desirable to user. This dissertation proposes the design and implementation for variable name length directory entries. Minix does not use fast symbolic link, moreover, it does not provide linkage between two files located on two different devices. To implement the fast symbolic link, system must provide symbolic link. This dissertation provides the design and implementation of fast symbolic link and symbolic link for files located on different devices.

Another difficulty in Minix is, it reads the directory entry from the disk, each time, if it is required by the file system. That is, a lot of disk read lead to lower throughput and so main impetus behind slower performance. This dissertation also provides a design for directory buffer cache to put it into the practice.

These modifications will finally lead to functional and performance enhancement in Minix file system, which is now deemed to use on low-end systems, with different directory management policy.

1.3 Dissertation Organisation

This dissertation has been dissected into following chapters:

- Chapter 2: In this chapter, basic file system concepts have been explained.
- Chapter 3: This chapter covers the Minix file system architecture.

- Chapter 4: In this chapter, design and implementation of variable name length directory entry, fast symbolic link and symbolic link has been discussed.
- Chapter 5: This chapter discuss about design and implementation of directory object construction and its use with directory entry buffer cache.
- Chapter 6: This chapter presents conclusions and future work.
- References
- Source code

CHAPTER 2 FILE SYSTEM

2.1 Files

From the users’ standpoint, the most important aspect of a file system is how it appears to them, that is, what constitutes a file, what is the length of file name, how files are named, and protected, what operations are allowed on files, and so on. The details of whether linked lists or bit maps are used to keep track of free storage and how many sectors there are in a logical block are of less interest, although they are of great importance to the designers of the file system. Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. Of course, the most part of this process is hidden to the user. In this section, we will look at different characteristics of files.

2.1.1 File Attributes

A file has certain attributes, which vary from one operating system to another, but typically consist of these:

- Symbolic name of the file,
- Type of the file,
- Location of the file on the disk or device,
- Current size of the file,
- Access-control information, and
- Time and date of creation, last access and last modification.

The information about all files is kept in the directory structure that also resides on disk or device. In general, directories are special type of files. The size of directory itself may be megabytes. We will discuss the directory organization, later in this chapter.

2.1.2 File Operations

Different systems provide different operations to allow storage and retrieval. Following are the minimal set of file operations:

- Creating a file: Free space in the file system must be found for the file and its entry must be made inside the directory.
- Opening a file: The purpose of this operation is to fetch attributes and list of disk addresses into main memory for rapid access on later calls.
- Closing a file: All the data blocks of the file are written to the disk and directory entry is updated, accordingly.
- Writing a file: The system must search the directory to find the location of the file. The system must keep a write pointer to the location in the file where the next write is to take place.
- Reading a file: The directory is searched for the associated directory entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. The bytes come from the current-file-position pointer.
- Repositioning within a file: The directory is searched for the appropriate entry, and the current-file-position is set to a given value.
- Deleting a file: After locating the directory entry, all file space is released so that it can be reused by other files. Subsequently, erase the directory entry.
- Truncating a file: This is similar to deleting the file except the directory entry is not erased, and so all attributes of file remain unchanged except the contents.

Other common operations include appending new information to the end of file and renaming an existing file. These primitive operations may then be combined to perform other file operations. For instance, creating a copy of a file may be accomplished by creating a new file and reading from the old and writing to the new.

2.1.3 File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in figure 2-1. The file in figure 2-1(a) 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. Any meaning must be imposed by user level programs. Both UNIX and MS-DOS 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 anyway that is convenient.

Another structure is shown in figure 2-1(b). In this model, 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 that the read operation returns one record and the write operations overwrites or appends one record.

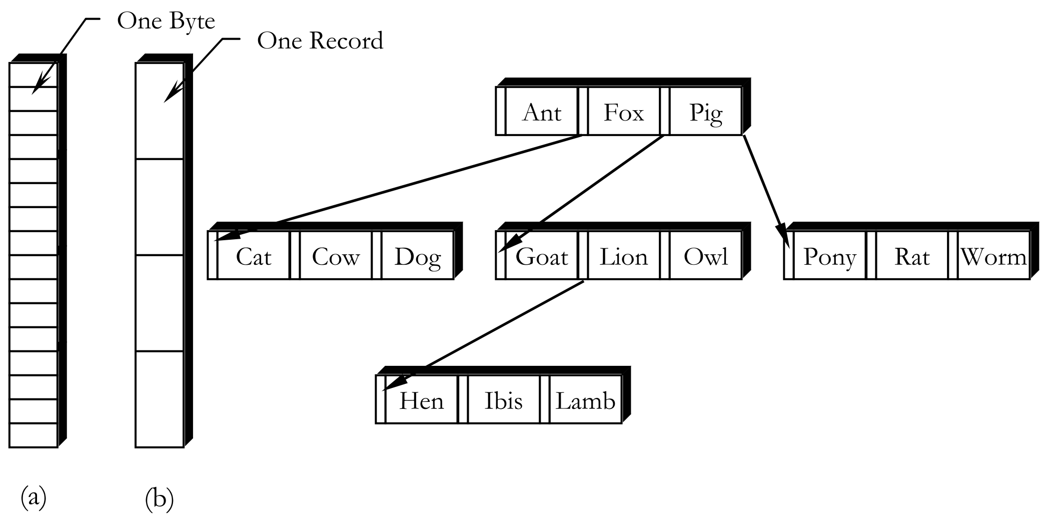


Figure 2-1. Three kinds of files. (a) Byte sequence (b) Record sequence (c) Tree

The third kind of file structure is shown in figure 2-1(c). In this organization, 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.

2.1.4 File Types

One major consideration in designing a file system is whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. Many operating systems support several types of files. However, all operating system must recognize one file type, their own executable file. All the files are just a sequence of bytes, but the operating system will only execute a file if it has the proper format. For example, UNIX and MS-DOS 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 input /output 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.

2.1.5 File Implementation

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. Various methods are used in different operating systems. Some of the space allocation schemes are:

- Contiguous allocation
- Linked list allocation
- Linked list allocation using an index
- Indexed allocation or I-nodes

Contiguous allocation is the simplest allocation scheme, which 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. In systems where files must be written in a single blow, it can be used to great advantage, however. The second disadvantage is the fragmentation of the disk that results from this allocation policy. Space is wasted that might otherwise have been used.

Linked list allocation scheme keeps each file as a linked list of data blocks. The first word of each block is used as a pointer to the next one. The rest of the block is for data. 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.

Linked list allocation using an index eliminates the disadvantage of the linked list allocation by taking the pointer word from each disk block and putting it in a table or index in memory. Figure 2-2 shows what the table looks like. 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 figure 2-2, we can start with block 4 and follow the chain all 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. MS-DOS 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.

0	
1	Unused Block
2	10
3	11
4	7 (file A start here)
5	Unused Block
6	3 (file B start here)
7	2
8	Unused Block
9	Unused Block
10	12
11	14
12	0
13	Unused Block
14	0
15	Unused Block

Figure 2-2. Linked list allocation using a table in main

Indexed allocation or I-nodes uses an index block. Each file has a little table called an i-node (index node), which lists the attributes and disk addresses of the file's blocks. 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 addresses 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.

2.2 Directories

The file system of computers can be extensive. Some systems store thousands of files on hundreds of gigabytes of disk. To manage all these data directories are used. From designers standpoint, directories are a special kind of file that stores the information about other files. To keep track of files, file systems normally have directories, which, in many systems, are themselves files. Information stored in directory is data about files, which belongs to that directory. When a process creates a file or directory, it gives the file or directory a name. When the process terminates, the file or directory continues to exist and can be accessed by other processes using its name. This symbolic name is a major attribute of files and directories. The exact rules for file or directory naming vary somewhat from system to system e.g. Some file system supports names as long as 255 characters, Some file system distinguishes between upper case letters and lower case letters, whereas others do not. UNIX falls in the first category; MS-DOS falls in the second.

2.2.1 Directory Operations

Some of the operations that are performed on directories are:

- Search for a file: To find the attributes of a file if symbolic name is provided or to find all files, which has symbolic name of a particular pattern.
- Create a file: New files are created and their entries are made inside the directory.
- Delete a directory: To delete an empty directory.

- List a directory: To return the contents of the directories.
- Rename a file: Symbolic name provides a method to distinguish files. Renaming a file requires positioning within the directory to change the file attribute.
- Linking a file: To make accessible, same file from different directories. That is same file is pointed by two or more directory entries.
- Unlinking a file: To remove the directory entry from the file. If file is present in one directory, it is removed from the file system. If it is present in multiple directories, only the path name specified is removed.
- Traverse the file system: To access every directory and every file within a directory.

2.2.2 Directory Structure

A directory typically contains a number of entries, one per file. There are two methods two store the file name and its attributes. In first method, file name and attributes are stored together as a single entry. This structure is shown in figure 2-3(a). However, in second method, directory entry holds the file name and a pointer to a data structure where the attributes and disk addresses are found. This structure is shown in figure 2-3(b). Both of these structures are commonly used.

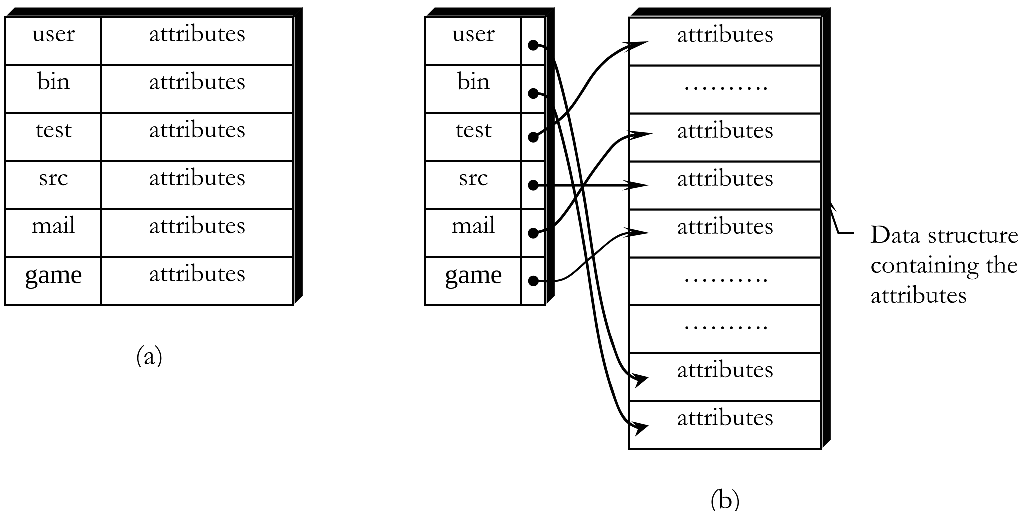


Figure 2-3. (a) Attributes in directory entry. (b) Attributes elsewhere

When a file is opened, the operating system searches its directory until it finds the name of the file to be opened. It then extracts the attributes and disk address e.g. either directly from the directory entry or from the data structure pointed to, and puts them in a table in main memory. All subsequent references to the file use the information in main memory.

There are various types of directory structure. Most common schemes for defining the logical structure of a directory are as follows.

2.2.2.1 Single-Level Directory

The simplest directory structure is the single level directory structure. All files are contained in the same directory as shown in figure 2-4. In figure 2-4, arrow is pointing from directory (parent) to files contained in that directory (child). This scheme has certain limitations, however, when the number of files increases or when there is more than one user. Since all files are in the same directory, they must have unique names. If there are many users, and they choose the same file names, conflicts and confusion will quickly make the system unworkable. Even with a single user, as the number of files increases, it becomes difficult to remember the names of all files to create only files with unique names. This system model was used by the first microcomputer operating systems but it is rarely seen any more.

2.2.2.2 Two-Level Directory

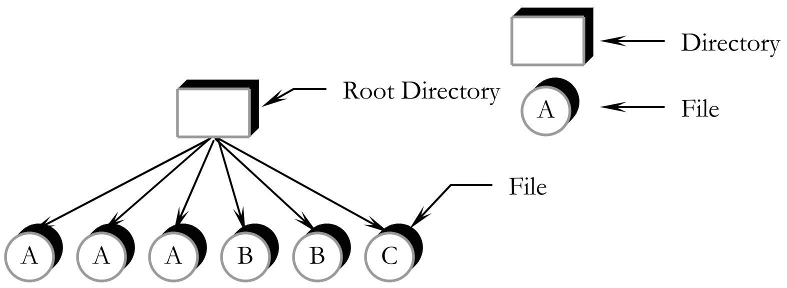


Figure 2-4. Single-level directory

An improvement on the idea of having a single directory for all files in the entire system is to have a separate directory for each user. In the two-level directory structure, each user has his own user-file-directory. Each user-file-directory has a similar directory structure, but lists only the files of a single user. When a user refers to a particular file, only his own user-file-directory is searched. Thus, different users may have files with the same name, as long as all the file names within each user-file-directory are unique.

To create a file for user, the operating system searches only that user's user-file-directory to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local user-file-directory. Thus, it cannot delete another user's file that has the same name.

This design eliminates name conflicts among users but is not satisfactory for users with a large number of files. This structure effectively isolates one user form another. This

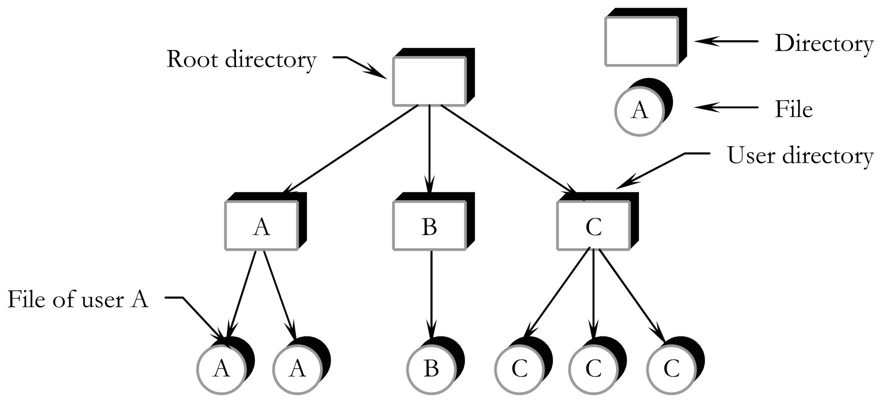


Figure 2-5. Two-level directory structure

isolation is an advantage when the users are completely independent, but is a disadvantage when the users want to cooperate on same task and to access one another's files. Some way is needed to group files of both users together in flexible ways.

2.2.2.3 Tree-Structured Directory

The natural generalization is to extend the directory structure to a tree of arbitrary height. With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This generalization allows users to create their own directories and to organize their files accordingly. It is the most common directory structure. The tree has a root directory. Every file in the system has a unique path name. A path name is the path from the root, through all the subdirectories, to a specified path. A directory or subdirectory contains a set of files or subdirectories. A directory is simply another file, but it is treated in special way. All directories have the same internal format. One flag in attribute data structure is used to distinguish entries as file or directory entries.

In, normal use, each user has a current directory. The current directory should contain most of the files that are of current interest t the user. When reference is made to a file, the current directory is searched. If a file is needed that is not is the current directory, then the user must either specify a path name or change the current directory to be the directory holding that file. To change the current directory to a different directory, a system call is provided which takes a directory name as a parameter and uses it to redefine the current directory.

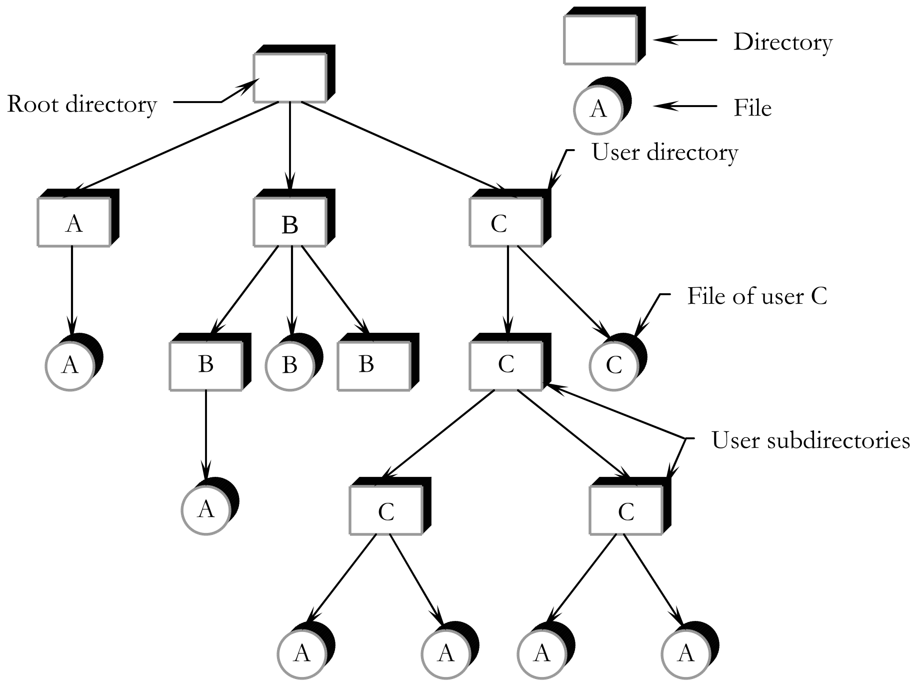


Figure 2-6. Tree structured directory

Allowing the user to define his own subdirectories permits him to impose a structure on his own files. This structure might result in separate directories for files associated with different topics or different information. With a tree-structured directory system, users can access in addition to their files, the files of other users. A user can access files of another user by specifying their path names. MS-DOS operating system uses this tree-structured directory approach.

2.2.2.4 Acyclic-Graph Directory

A tree structured prohibits the sharing of files or directories. An acyclic-graph allows directories to have shared subdirectories and files. The same file may be in two different directories. An acyclic-graph is a natural generalization of tree structured directory scheme.

An acyclic-graph directory structure is more flexible than is a simple tree structure, but is more complex. Several problems must be considered carefully. Notice that a file may now have multiple path names. Consequently, distinct file names may refer to the same file. If we are trying to traverse the entire file system, this problem becomes significant, since we do not want to traverse shared structure more than once. Another problem involves deletion. Decision has to be made for the deallocation and reuse of the space allocated to a shared file. One possibility is to remove a file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file-pointers contain the actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into middle of other file.

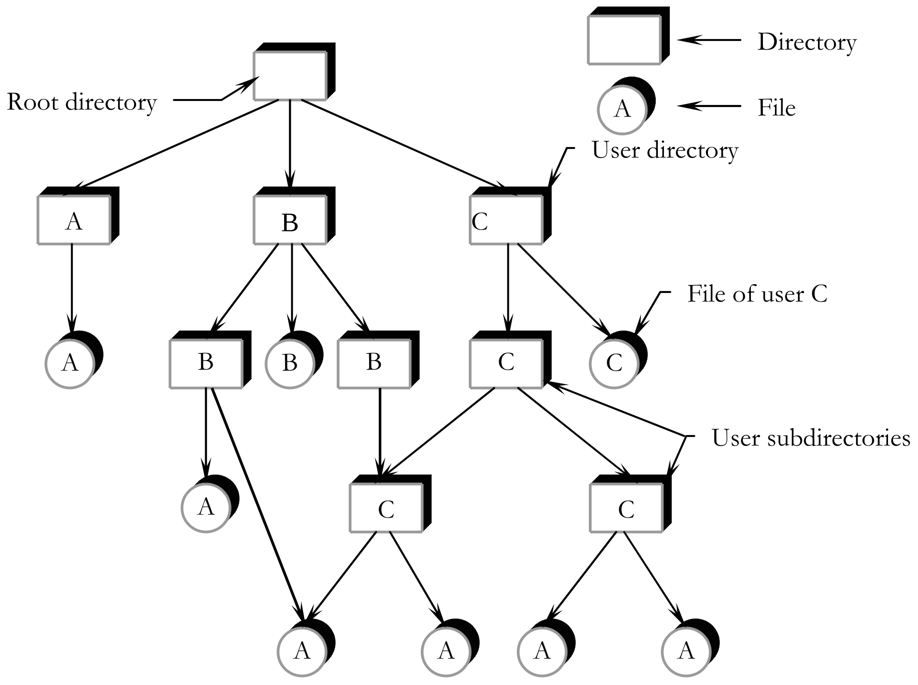


Figure 2-7. Acyclic-graph directory structure

In a system where sharing is implemented by symbolic links (see definition in next section), this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed. If the entry itself is removed, the space for the file is deallocated, leaving the links dangling. Other links are left until the attempt is made to use them. If attempt fails, the access is treated just like any other illegal name. Another problem is, what to do when a file is deleted and another file of same name is created, before a symbolic link to the original file is used? This problem is handled using different method on different systems. Easiest method is, access the new file.

Another approach is to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that last reference to the file has been deleted. A count of the number of references is stored inside file attributes. A new link increments the reference count; deleting a link decrements the count. When the count is zero, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for nonsymbolic links.

2.2.2.5 General-Graph Directory

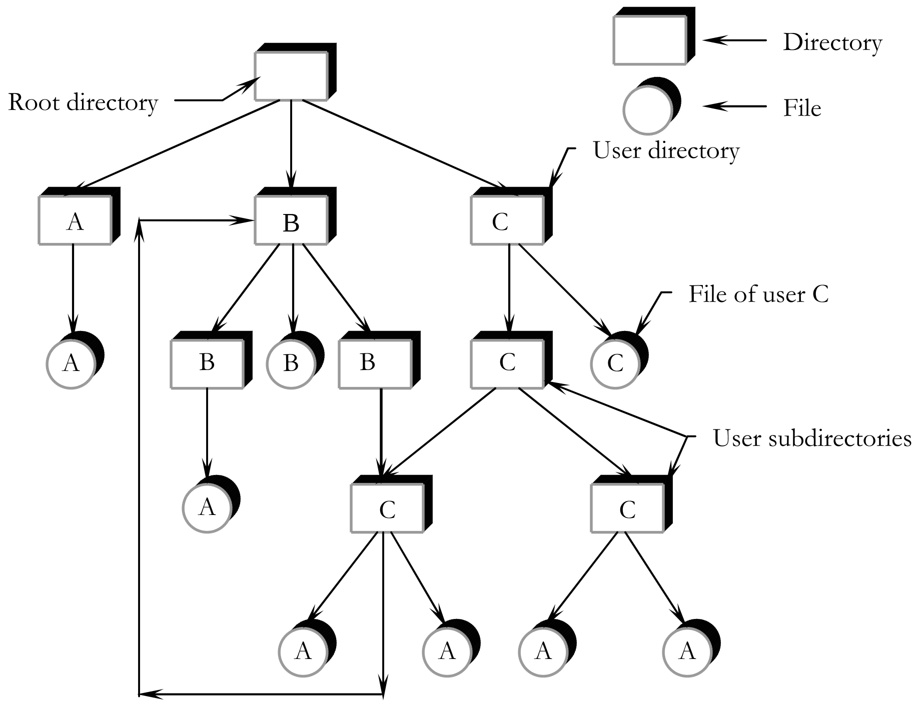


Figure 2-8. General-graph directory structure

One serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to arbitrarily limit the number of directories that will be accessed during a search.

In this structure a major problem involved with file deletion, if we are using reference count similar to acyclic-graph, it is possible that the reference count is nonzero due to self-referencing pointer of file. To solve this problem garbage collection scheme is used. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then a second pass collects everything that is not marked onto a list of free space. However, this method is very time consuming and is thus seldom attempted. Thus, an acyclic graph structure is much easier to work with.

2.2.3 Path Names

When a file system makes use of a tree, acyclic-graph or general-graph structured directory system. Some way is needed for specifying file names. Two different methods are commonly used:

- Absolute path names
- Relative path names

An *absolute path name* begins at the root and follows a path down to the specified file, giving the directory names on the path. As an example, the path */usr/dce/mailbox* means that the root directory contains a subdirectory *usr*, which in turn contains a subdirectory *dce*, which contains the file *mailbox*. Absolute path names always start at the root directory and are unique. In UNIX, the components of the path are separated by /. In MS-DOS, the separator is \. In MULTICS, it is >. No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

A *relative path name* defines a path from the current directory to the specified file. This is used in conjunction with the concept of the current directory. A user can designate one directory as the current working directory; in that case, all path names not beginning at the root directory are taken relative to the working directory.

2.2.4 Directory Implementation

It is time to turn from the user's point of view of the file system to the implementor's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like and similar interface issues. Implementors are interested in how files and directories are stored, how disk space is managed and how to make everything work efficiently and reliably. In this section, we will see the different directory entry format of few operating systems.

Before a file can be read, it must be opened. 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. 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.

2.2.4.1 Directories in MS-DOS

Directory entry of MS-DOS is 32 bytes long and contains the file name, attributes, and the number of the first disk block. The first block number is used as an index into a table of the type of figure 2-2. By following the chain, all the blocks can be found. In MS-DOS, directories may contain other directories, leading to a hierarchical file system. Figure 2-9 shows directory entry structure of MS-DOS operating system.

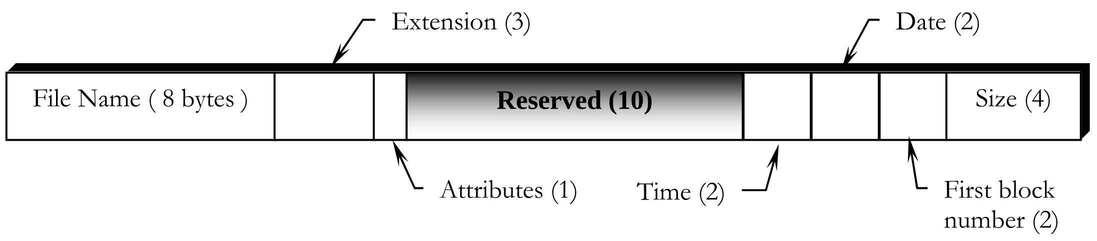


Figure 2-9. The MS-DOS directory entry structure

2.2.4.2 Directories in UNIX

The directory structure traditionally used in UNIX is extremely simple. Each entry contains just a file name and its i-node number. All the information about the type, size, times, ownership, and disk blocks is contained in the i-node (index 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.

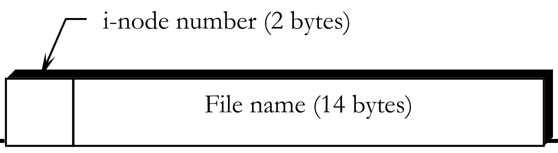


Figure 2-10. The UNIX directory entry structure

2.3 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. 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. For this reason, nearly all file systems chop files up into fixed size blocks that need not be adjacent.

The major problem with fixed size block is management of free blocks. Two methods are widely used. 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 bytes block and a 32 bit disk block number, each black on the free list holds the numbers of 255 free blocks. However, one slot is needed for the pointer to the next block. This scheme has been shown in figure 2-11(a).

The other free space management technique is the bit map. Its structure is shown in figure 2-11(b). A disk with *n* blocks requires a bit map with *n* bits. Free blocks are represented by 1's in the map, allocated blocks by 0's or vice versa. A 200M bytes disk requires 200K bits for the map, which requires only 2.5 blocks. The bit map requires less space, since it uses 1 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 the bit map.

If there is enough main memory to hold the bit map, that method is generally preferable, If, however, only one 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 one 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.

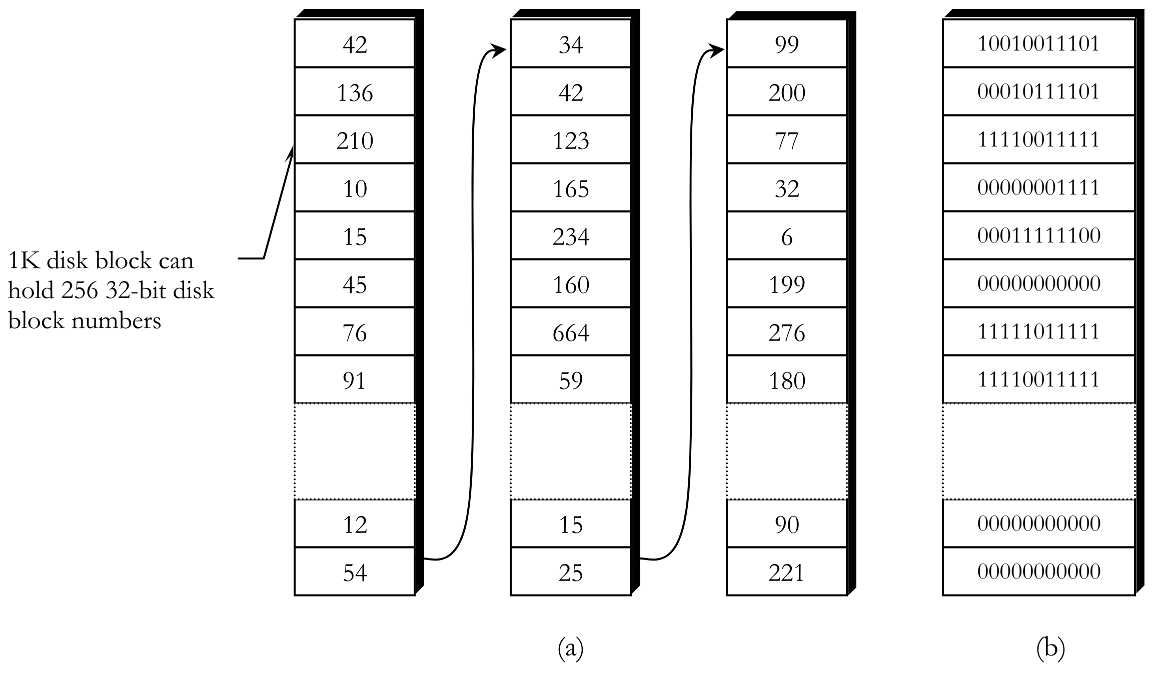


Figure 2-11. (a) Storing the free list on a linked list (b) A bit map

2.4 File System Performance

Access to disk is much slower than access to memory. Reading a memory word typically takes tens of nanoseconds. Reading a block from a hard disk may take fifty microseconds, a factor of four slower per 32 bit word, but to this must be added 10 to 20 milliseconds to seek to the track and then wait for the desired sector to arrive under the read head. If only a single word is needed, the memory access is of the order of 100,000 times as fast as disk access. As a result of this difference in access time, many file systems have been designed to reduce the number of disk accesses needed.

The most common technique used to reduce disk accesses is the block cache or buffer cache. (Cache is pronounced, “cash” and is derived from the French *cacher,* meaning *to* *hide*.) In this context, a cache is a collection of blocks that logically resides on the disk but are being kept in memory for performance reasons.

Various algorithms can be used to manage the cache, but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache, and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache. When a block has to be loaded into a full cache, some black has to be removed and rewritten to the disk if it has been modified since being brought in. This situation is very much like paging, and algorithms such as first-in-first-out and least-recently-used, are applicable to it.

There is a problem with the crashes and file system consistency. Let buffer cache is using least-recently-used replacement policy. If a critical block, such as an i-node block, is read into the cache and modified, but not rewritten to the disk, a crash will leave the file system in an inconsistent state. If the i-node block is put at the end of the least-recently-used chain, it may take quite a while before it reaches the front and is rewritten to the disk. Furthermore, some blocks, such as double indirect blocks, are rarely referenced two times within a short interval. These considerations lead to a modified least-recently-used scheme, considering two factors:

- Is the block likely to be needed again soon?
- Is the block essential to the consistency of the file system?

For both questions, blocks can be divided into categories. Blocks that will probably not be needed again soon go on the front, rather than the rear of the least-recently-used list, so their buffers will be reused quickly. Blocks that might be needed again soon, such as a partly full block that is being written, go on the end of the list, so they will stay around for a long time. The second question is independent of the first one. If the block is essential to the file system consistency (and it has been modified, it should be written to disk immediately, regardless of which end of the which least-recently-used list it is put on. By writing critical blocks quickly, we greatly reduce the probability that a crash will wreck the file system. Even with this measure to keep the file system integrity intact, it is undesirable to keep data blocks in the cache too long before writing them out. At regular time interval, it forces all the modified blocks out onto the disk immediately. UNIX uses this scheme and call SYNC system call at regular interval to write all the blocks of least-recently-used list on the disk. The MS-DOS way is to write every modified block to disk as soon as it has been written. Caches in which all modified blocks are written back to the disk immediately are called write-through caches. They require much more disk input/output than non-write-through caches. A consequence of this difference in caching strategy is that just removing a floppy disk from a UNIX system without doing a SYNC will usually result in lost data, and frequently in a corrupted file system as well. With MS-DOS, no problem arises.

Caching is not the only way to increase the performance of a file system. Another important technique is to reduce the amount of disk arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder. When an output file is written, the file system has to allocate the blocks one at a time, as they are needed. If the free blocks are recorded in a bit map, and the whole bit map is in main memory, it is easy enough to choose a free block as close as possible to the previous block. With a free list, part of which is on disk, it is much harder to allocate blocks close together. However, even with a free list, some block clustering can be done. The trick is to keep track of disk storage not in blocks, but in groups of consecutive blocks.

CHAPTER 3 EXISTING MINIX FILE SYSTEM

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

The Minix file system is just a big C program. To read and write files, user processes send messages to the file system telling what they want done. The file system does the work and sends back a reply. Its design has some important implications. The file system can be modified, experimented with, and tested almost completely independently of the rest of Minix. Moreover, it is very easy to move the whole file system to any computer that has a C compiler, compile it there, and use it as a stand-alone UNIX-like remote file server. The only changes that need to be made are in the area of how messages are sent and received, which differs from system to system. In this chapter, we will look at areas of file system design such as messages, the file system layout, i-nodes, the buffer cache, directories and paths.

3.1 Messages

The file system accepts 39 types of messages requesting work. All but two are for Minix system calls. The two exceptions are messages generated by other parts of Minix. Of the system calls, 31 are accepted from user processes. Six system call messages are for system calls that 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. The messages are shown in figure 3-1.

File system 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. This procedure has been shown in following code-fragment, which is a part of *main.c*.

Messages from users	*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 the new root directory***
CLOSE	***File descriptor of file to close***
CREAT	***Name of the file to be created***
DUP	***File descriptor***
FCNTL	***File descriptor, function code, arg***
FSTAT	***Name of file, buffer***
IOCTL	***File descriptor, function code, arg***
LINK	***Name of file link to, name of link***
LSEEK	***File descriptor, offset, whence***
MKDIR	***File name, mode***
MKNOD	***File name of directory or special, mode, address***
MOUNT	***Special file, where to mount, read only flag***
OPEN	***Name of file to open, read/write flag***
PIPE	***Pointer to two 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***
SYNC	***(none)***
TIME	***Pointer to place where current time goes***
TIMES	***Pointer to buffer for prcess and child times***
UMASK	***Complement of mode mask***
UMOUNT	***Name of special file to unmount***
UNLINK	***Name of the file to unlink***
UTIME	***File name, file times***
WRITE	***File descriptor, buffer, how many bytes***
Messages from MM	*Input Parameters*
EXEC	***Pid***
EXIT	***Pid***
FORK	***Parent and child pid***
SETGID	***Pid, real and effective gid***
SETSID	***Pid***
SETUID	***Pid real and effective gid***
Other Messages	*Input Parameters*
REVIVE	***Process to revive***
UNPAUSE	***Process to check***

Figure 3-1. Messages of Minix

```
while (TRUE) {
        …………. /* initialize fs_call and who */
        if (fs_call < 0 || fs_call >= NCALLS) /* NCALLS is maximum number of calls*/
                error = EBADCALL;
        else
                error = (*call_vector[fs_call])(); /* Call the internal function that does the
work. */

        /* Copy the results back to the user and send reply. */
        ………….
        reply(who, error);
        ………….
}
```

3.2 File System Outline

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 (portion of a) hard disk. In all cases, the layout of the file system has the same structure. Figure 3-2 shows this layout. Each File systems, 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 has 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.

The super-block contains information describing the layout of the file system. It is illustrated in figure 3-3. 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 1K bytes or 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, each i-node occupy 64 bytes, a 1K block holds up to 16 i-nodes. With 128 usable i-nodes, 8 disk blocks are needed to contain them all.

Disk storage can be allocated in units, called *zones*, of 1, 2, 4, 8 or in general 2^n blocks. The zone bit map keeps track of free storage in zones, not blocks. For all standard floppy disks used by Minix file system, 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.

The number of blocks per zone is not stored in the super-block, as it is never needed. All 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. For example, with 8 blocks per zone, $\log_2 8 = 3$, so to find the zone containing block 128 we shift 128 right 3 bits to get zone 16.

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. The first data zone designates zone 1 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. If we examine the bit maps on a newly formatted disk, we will find that 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.

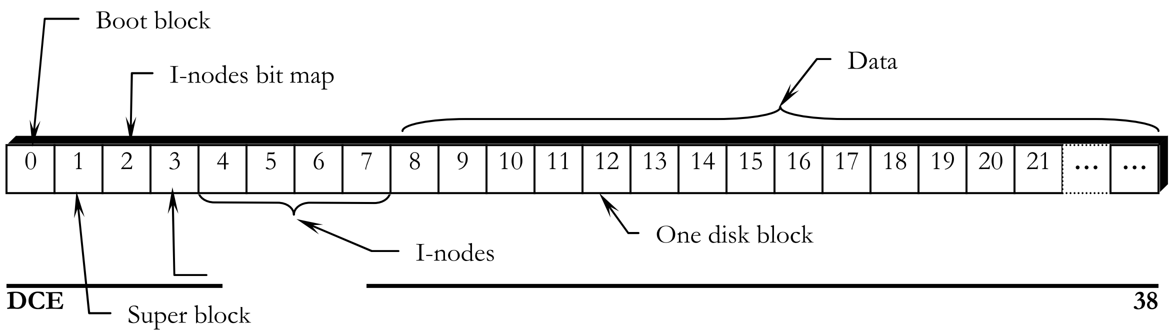


Figure 3-2. Disk-layout of the Minix file system

The information in the super-block is redundant because sometimes it is needed in one form and sometimes in another. Structure of super block is given in figure 3-3. With 1K devoted to the super block, it makes sense to compute this information and store it in the super-block 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. The rest of the super-block is wasted anyhow, so using up another word of it costs nothing.

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

Figure 3-3 The Minix super block structure present on disk and in memory

Pointer to i-node for root of mounted file system
Pointer to i-node mounted upon
i-nodes/block
Device number
Read-only flag
Big-endian FS flag
File System version
Direct zones/i-node
Indirect zones/indirect block
Firs free bit in i-node bit map
First free bit in zone bit map

Figure 3-4 The Minix super block structure present in memory but not on disk

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 in the super-block of 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. The magic number in the super-block is used to identify the file system and its version as a valid Minix file system. Attempts to mount a file system not in Minix format, such as an MS-DOS diskette, will be rejected by the MOUNT system call, which checks the super-block for a valid magic number and other things. Before a disk can be used as a Minix file system, it must have the structure of figure 3-2.

3.3 Bit Maps

Minix keeps tracks of which i-nodes and zones are free by using two bit maps i.e. i-node bit map and zone bit map. When a file is removed, it is very 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 zero. Zones are released from the zone bit map in the same way.

Logically, 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. In fact, 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, when the pointer must be updated to point to the new next free i-node, which will often turn out to be the next one, or a close one. Similarly, when an i-node is freed, a check is made to see if the free i-node comes before the currently pointed-to one, and the pointer is updated if necessary. If every i-node slot on the disk is full, the search routine returns a zero, that is why zeroth i-node is not used; so it can be used to indicate the search failed. When a new file system is created, it zeroes i-node 0 and sets the lowest bit in the bit map to 1, so the file system will never attempt to allocate it. It, also, applies to the zone bit map. Logically, 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.

As Minix developed, and larger disks became much more common, it was obvious that changes were desirable. Many files are smaller than 1K, so increasing the block size would mean wasting disk bandwidth, reading and writing mostly empty blocks and wasting precious main memory storing them in the buffer cache. The zone size could have been increased, but a larger zone size means more wasted disk space, and it was still desirable to retain efficient operation on small disks. Another reasonable alternative would have been to have different zone sizes on large and small devices.

Zones also introduce an unexpected problem, best illustrated by a simple example, again with 4K zones and 1K blocks. Suppose that a file is of length 1K, meaning that 1 zone has been allocated for it. The blocks between 1K and 4K contain garbage (residue from the previous owner), but no harm is done because the file size is clearly marked in the i-node as 1K. In fact, the blocks containing garbage will not be read into the block cache, since reads are done by blocks, not by zones. Reads beyond the end of a file always return a count of 0 and no data. Now someone seeks to 32768 and writes 1byte. The file size is now changed to 32769. Subsequent seeks to 1K followed by attempts to read the data will now be able to read the previous contents of the block, a major security breach.

The solution is to check for this situation when a write is done beyond the end of a file, explicitly zero all the not yet allocated blocks in the zone that was previously allocated. Although this situation rarely occurs, the code has to deal with it, making the system slightly more complex.

3.4 I-nodes

The layout of the Minix file system i-node is given in figure 3-5. 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 UNIX 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 file system. The Minix i-node access, modification time and i-node change times are standard. 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, as shown in figure 3-6, 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.

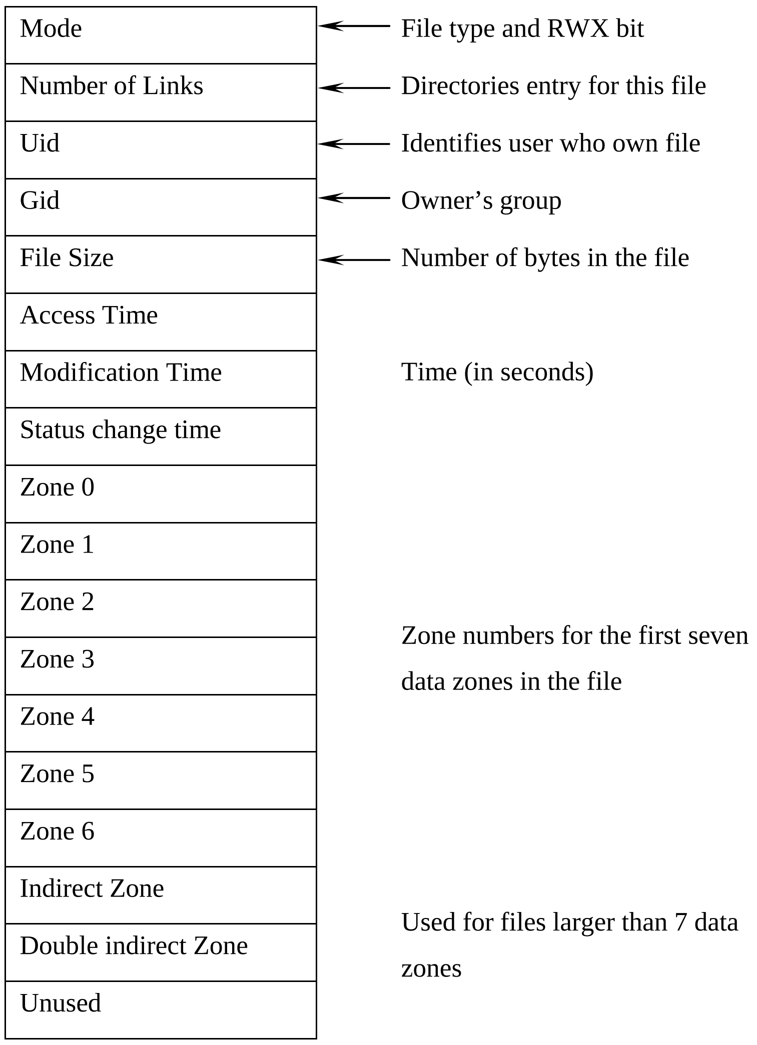


Figure 3-5. The Minix i-node structure

```
EXTERN struct inode {
………………..
………………..
```

```
  /* 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 */
};
```

Figure 3-6. The i-node information that not reside on the disk

The main function of a file’s i-node is to tell where the data blocks of the given file are. The first seven zone numbers are given right in the i-node itself. With zones and blocks both 1K bytes, files up to 7K bytes do not need indirect blocks. Beyond 7K bytes, indirect zones are needed, using the scheme of figure 3-7, except that only the single and double indirect blocks are used. With 1K byte blocks and zones and 32 bit 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 64M bytes. The maximum size of a Minix file system is 1G bytes, so modification to use the triple indirect block or larger zone sizes could be useful if it were desirable to access very large files on a Minix system.

The i-node also holds the mode information, which tells what kind of a file it is (e.g. regular, directory, block special, or character special files) gives the protection information and user-id and group-id 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. This field is different from the counter (present only in the i-node table in memory, not on the disk) that tells how many times the file is currently open, typically by different processes.

3.5 Directories and Paths

Another important subsystem within the file system is the management of directories and path names. Many system calls have a file name as a parameter. What is really needed is the i-node for that file, so it is up to the file system to look up the file in the directory tree and locate its i-node.

To look up the path */usr/dce/mailbox*, first, the file system locates the root directory. In Minix, its i-node is located at a fixed place on the disk. 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.

From this i-node, the system Locates the directory for */usr* and looks up the next component, *dce*, in it. When it has found the entry for *dce*, it has the i-node for the directory */usr/dce*. From this i-node, it can find the directory itself and look up *mailbox*. The i-node for this file is then read into memory and kept there until the file is closed. The lookup process is illustrated in figure 3-7.

Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of starting from the root directory. Every directory has entries for **.** and **..** which are put there when the directory is created. The entry **.** has the i-node number for the current directory, and the entry for **..** has the i-node number for the parent directory. Thus, a procedure looking up **..**/*dce/prog.c* simply looks up **..** in the working directory, finds the i-node number for the parent directory, and searches that directory for *dce*. No special mechanism is needed to handle these names. As far as the directory system is concerned, they are just ordinary ASCII strings, just the same as any other names.

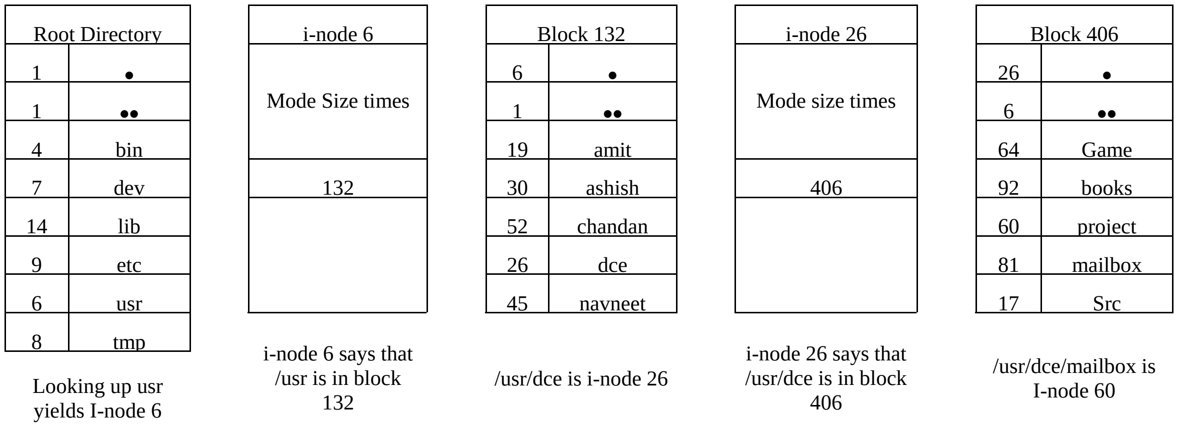


Figure 3-7. Path look-up inside Minix file system

The only complication involved is what to do when a mounted file system is encountered. The usual configuration for Minix system 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 users' directories, on a separate device mounted on */usr*. Here we look at how mounting is done. When the user types the command *mount /dev/fd0 /usr* on the terminal, the file system contained on floppy disk is mounted on top of */usr* in the root file system.

The file systems before and after mounting are shown in figure 3-8. The key to the whole mount business 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

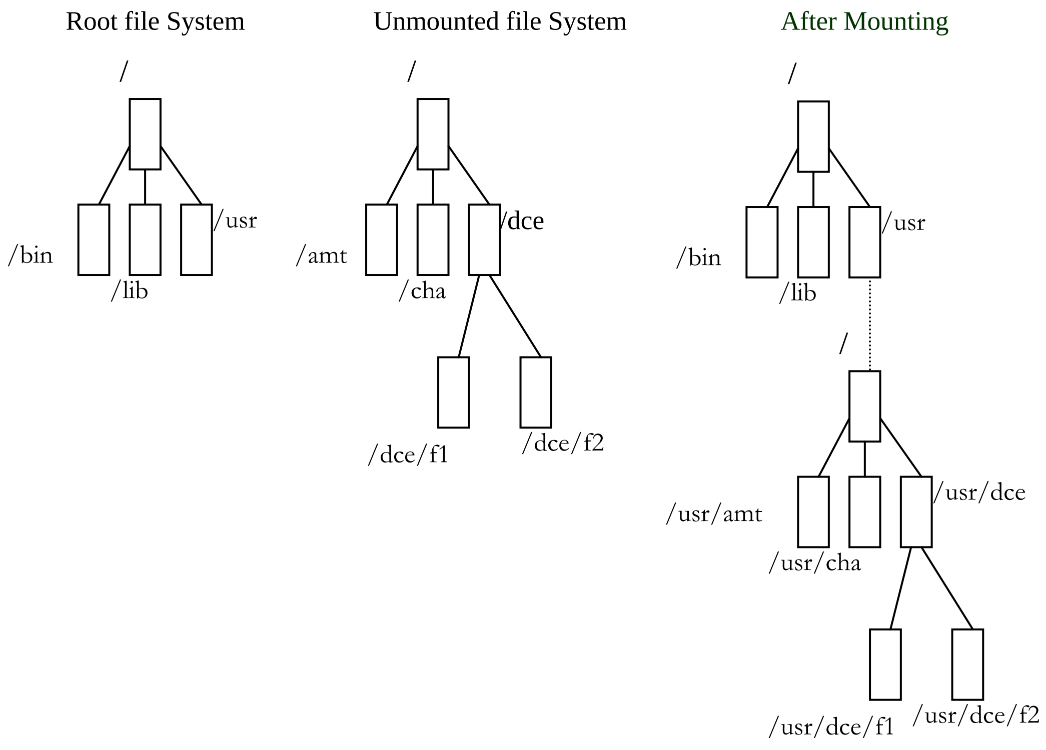


Figure 3-8. Mounting in Minix file system

system in the i-node table. In figure 3-4, 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. This glue is what makes mounted file systems work.

When a path such as */usr/dce* is being looked up, the file system will see a flag in the i-node for */usr* and realize that it must continue searching at the root i-node of the file system mounted on */usr*. Now, the system searches all the super-blocks in memory until it finds the one whose i-node mounted on field points to */usr*. This must be the super-block for the file system mounted on */usr*. Once it has the super-block, it is easy to follow the other pointer to find the root i-node for the mounted file system. Now the file system can continue searching. In this example, it looks for *dce* in the root directory of floppy disk.

3.6 Links

Shared files and subdirectories are implemented using link in MINIX.A link is a pointer to another file or subdirectory. There are three types of links, as mentioned in chapter 1.

3.6.1 Hard Link

The hard links are just another pointers to same i-node. A hard link is just an inode number in some directory entry. Its disadvantage is, it does not work properly between files situated on two different devices.

3.6.2 Symbolic Link

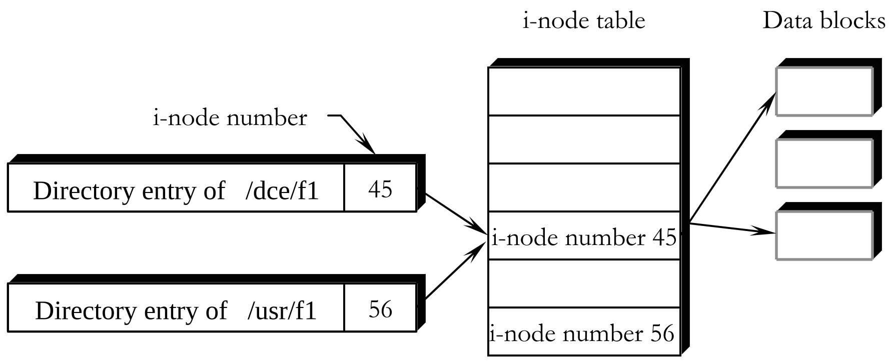


Figure 3-7. Working of hard link

It is just a path name, which is accessible from an inode. When the kernel reaches a symbolic link, it will follow it in run time using its normal way of reaching directories and files. It has disadvantage that it consumes at least one i-node and a data block as well as it has additional time delay for data block reference.

3.6.3 Fast Symbolic Link

It is similar to symbolic link in function but it does not use any data block on the file system. The target name is stored in the inode itself rather than a data block. Thus, for path reference it does not goes to the data block. At run time, it retrieves path from the inode for traversal.

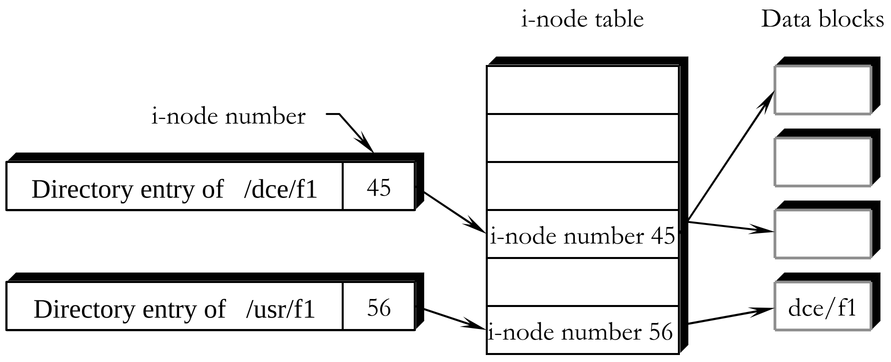


Figure 3-8. Working of symbolic link

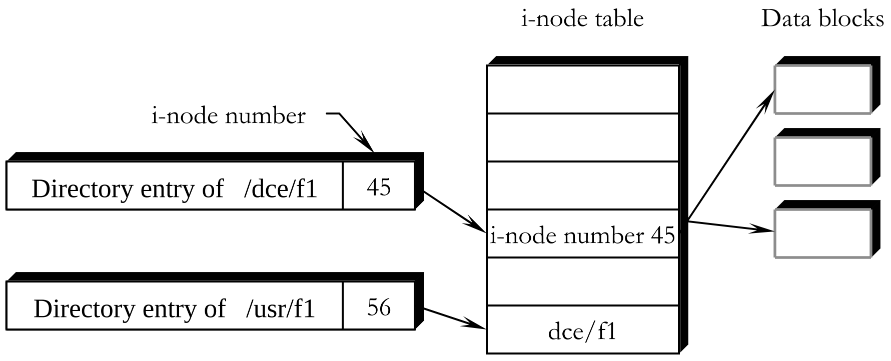


Figure 3-9. Working of fast symbolic link

CHAPTER 4 LONG FILE NAME AND FAST SYMBOLIC LINK

In this chapter, we will discuss the design and implementation of two functional enhancements. First of these two is, the support of variable size long file name. And the second is, to support fast symbolic link.

4.1 Long File Name

To support variable size file name, it is essential to modify the existing directory structure. Since, directory structure stores the file name. It is the target of modification.

4.1.1 Existing Directory Structure and issues

The existing directory structure is given below.

```
#define  DIRSIZ  14

struct direct
{
        ino_t    d_ino;
        char     d_name[DIRSIZ];
};
```

This directory is used by the existing Minix file system, as Minix use this directory structure to store file name of maximum length of 14 characters with corresponding inode number of the file. This structure corresponds to the figure 2-10 of UNIX operating system.

However, to change the existing file-name convention involves many difficulties. For example, assume that we are going to change size of macro DIRSIZ from 14 to 62. Only, replacement of macro DIRSIZ 14 with DIRSIZ 62 will not begin to store file name of 62 character long. There is a macro NAME_MAX defined in *limits.h*, that has a value of 14. Its value is same as DIRSIZ. NAME_MAX has been used in whole file system for storage and retrieval of file name. In addition, if we replace NAME_MAX value with 62 will not do the whole purpose.

This change will have severe effect on the file system, as directory entries are stored on the data blocks. And, when storage and retrieval of filename and their corresponding is required the data read from the data block is size of structure *direct* i.e. *sizeof (struct direct)*. Assume, the size of structure was, previously, 16 byte but after the modification in value of DIRSIZ, size of structure changes to 64 byte. It means that previously data read from the directory block was 16 bytes but now it is 64 bytes, and so it is trying to read the other 3 directory-entries of old directory structure. Thus, it will provide erroneous result. This difficulty teach us that we must start from scratch, that is, write a new file system with modified structure, otherwise, it will not work. However, in operating system design another major problem is to support backward compatibility i.e. system must support at least few last version's (if not, all) compatibility. Support of old versions is not something one reads about in theoretical texts, but it is always a concern for the implementer of a new version of any software. One must decide how much effort devote to making life easier for users of old versions.

Writing file system involves, writing of boot block, super-block, i-node bitmap blocks, zone bitmap blocks, i-node blocks, data blocks and two directory entries for the root directory. As we know that any directory contains at least two directory entry, one is for dot (**.**) directory and another is for dotdot (**..**) directory. Dot directory offers inode number of current directory and dotdot directory gives inode number of parent directory. It is necessary for path traversal in a hierarchical directory system. When file system is first written, there is just one directory, that is, root directory. Since, the inode number of root directory of any file system is always fix, so it doesn't contain the directory entry for the root directory. Root directory, too, contains two directory entries for dot and dotdot. Therefore, when we write a new file system, it must have at least two directories entry for dot and dotdot directory of root directory and each must have 62 characters space to store the file name, which is the modified file name length. It also writes magic number in super-block to identify the file system as valid Minix file system. The magic number, also, manages to supply the information about Minix file system version.

The above given sections was just to show the effect of name size change of a file system. However, we have to implement variable size file name directory structure. The reason behind this is to save the disk space wastage. For example, when we store the directory entry for dot (**.**), it has string length of one character, that is, all it need is disk space of 1 byte for character "**.**" and two bytes for i-node number. So at most, it needs 3 bytes of space on disk. As we know, the directory entry size is of 16 bytes (which is, in turn, is equal to *sizeof(direct)*). Thus here, the loss is of 16 – 3 = 13 bytes. Not all file names are of one character string length. It does vary. But still, there is loss of some space if file name size is less than 14 character. Moreover, we want to increase the file name size, that is, of 62 characters. As generally observed, the file name size are between 10 to 30 characters, however, in some cases its value is much more. In this case, too, there will be loss of space on disk, if file name size is less than 62 characters. This wastage of space can be stopped using variable name size file name length, at the cost of some extra information stored in the directory structure.

4.1.2 Modified directory structure

The modified directory structure will consist of some extra fields, in addition to the existing fields. The structure of new directory entry is given below. It is defined inside the *dir.h* file.

```
#define  V3_DIRSIZ      54

struct direct4
{
        ino_t           d_ino;          /* inode number */
        off_t           d_rclen;        /* directory entry length */
        unsigned short  d_nmlen;        /* file name length */
        char            d_name[DIRSIZ]; /* file name */
};
```

That is, the maximum length of the file name is 54 characters (this value is alterable), but it size on directory entry will vary. The *d_ino* will store the i-node number of the file, *d_rclen* will be used to find the next valid directory entry in the directory block, *d_nmlen* will have file name length information and,

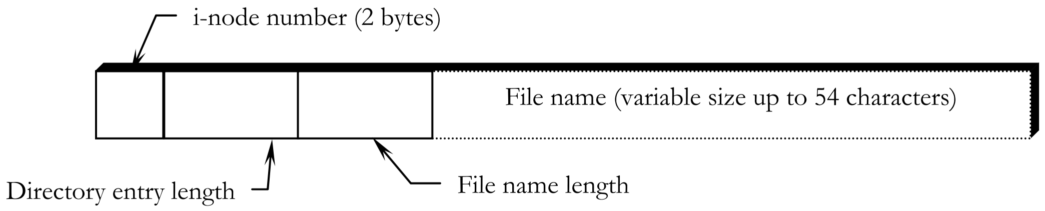


Figure 4-1. The modified directory entry

finally, *d_name* will store the file name.

For reason of efficiency, the length of file name string is multiple of four and, therefore, null characters will be added for padding at the end of the file name. We can see the layout of the entries in figure 4-1.

4.1.3 Implementation

Before we investigate the implementation of the variable size name length directory entry, we must look different macros and variables that have been used in the program.

```
#define  BLOCK_SIZE          1024    /* directory block size */
#define  SUPER_V3            0x345F  /* magic number for modified file system */
#define  SUPER_V3_REV        0x5F34   /* reverse magic number for modified file system */
#define  V3                  3       /*version number of modified file systems*/

#define  PATH_MAX            1023    /* number of characters in a path name */
#define  V3_NAME_MAX         54      /* number of characters in a file name of V3*/
```

The BLOCK_SIZE is equal to 1024; which is the size of data block on a Minix file system.

The SUPER_V3 and REV_SUPER_V3 is the magic number for new file system. As mentioned in above section that magic number provides information about file system, as well as, about its version. Magic number can be any number but one precaution must be taken. Its bit pattern must not be a palindrome. As this magic number, also, provide the information about big-endian or little-endian storage scheme on disk, that is, to swap byte or not when we read it form the disk. Note SUPER_V3 and REV_SUPER_V3 are byte swapped number. These both numbers are different from the existing file system magic number. Thus, by observing magic number we can tell about the file system version, that is, it is new file system or existing one.

The V3 Macro is nothing but to indicate the version number of modified file system.

These all macros are defined in *const.h* of file system, except last two, which are defined inside *limits.h*. Macro PATH_MAX has been redefined from 255 to 1023. Since, we are going to increase the file name size to 54.

When entries are made in directory block, some parameters are required quite frequently. These are:

```
#define  OFFNAME             ((int)(&(((struct direct4 *) (0))->d_name)))
                                                            /* Offset of name */
#define   OFFSET(a)          (BLOCK_SIZE - ( a % BLOCK_SIZE))
                                                            /* Back Block Offset */
#define  GPFY(x)                     (((int) (( MIN( x, V3_DIRSIZ) + 3)/4))*4)
                                                  /*Groupify the data byte in 4byte
multiple*/
#define   NEXTPTR_DIR(a,b)   (struct direct4 *) (((char *)(a)) + (b))
                                                  /*pointer to the next entry*/
```

Macro OFFNAME returns the offset of *d_name* inside the *direct4* structure. We cannot calculate it by simply adding the elements of the structure *direct4* i.e *sizeof(d_ino) + sizeof(d_rclen) + sizeof(d_nmlen)* because Its always wrong to assume that the size of a structure is the sum of the sizes its members. Because of alignment requirements for different objects, there may be unnamed holes in a structure. The *sizeof* operator returns the proper value [7].

Macro OFFSET(a) returns the number of bytes left after the specified position “a” in the directory block if its position “a” is given. It will be required to check the availability of space left in directory block. Note, BLOCK_SIZE is equal to the size of a directory block.

Macro GPFY(x) takes a value “x” and returns a value that is multiple of four. Since we are going to store the file name in multiple of four bytes, it provides the number of bytes required. For example, if we take the value of “x” as 1, 2, 3 or 4; it will return 4, for 5,6,7 or 8; it will return 8 and, so on. It also checks the file name size. If the file name size is more than V3_DIRSIZ, it uses “x” as V3_DIRSIZ. MIN is a macro that returns the minimum of two numbers.

Macro NEXTPTR_DIR(a,b) returns the address of next directory entry if the address of current entry is “a” and the next entry is “b” byte farther from the current entry. This value is required when we traverse the directory entries in directory block either for look-up process or to make an entry of a directory or a file, or for deletion of a directory entry.

These all macros are defined in *const.h* of file system.

The Minix file system uses buffer cache to speedup the processing of operating system. This buffer contains every type of blocks that come for processing inside main memory. The modified buffer structure is given here.

```
struct buf {
  /* Data portion of the buffer. */
        union {
        char            b__data[BLOCK_SIZE];                /* ordinary user data */
        /*****************************************************/
        struct direct   b__v2_dir[V2_NR_DIR_ENTRIES];       /* directory block V2*/
        char            b__vardirsz[BLOCK_SIZE];            /* directory block V3 */
        /*****************************************************/
        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 */
  } b;
  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 */
  block_t             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];

#define  b_v2_dir         b.b__v2_dir
#define  b_vardirsz       b.b__vardirsz
```

All the elements of the above given *buf* structure is similar to the existing Minix operating system's *buf* structure except, character array *b_vardirsz* of size BLOCK_SIZE. It will not change the size of structure *buf* as it is a part of union and there is already an element *b_data* with the same size. This declaration is to make declaration distinguishable. The *b_dir* of the existing system has been changed to *b_v2_dir* to differentiate between block containing old and new directory entries. It is always uncomfortable to handle long variable name, so it is easy to use macro for these, too.

4.1.3.1 Writing New File System

Before we write the procedure for look-up, insertion and deletion of directory entry inside a directory block, we must write new file system with new directory structure on it. The *mkfs.c* contains the code for the writing of a file system. It had been modified to incorporate the changes made inside directory structure. After compilation it generates the utility *mkfs* that will be used in following format at command prompt to write a new file system of version 3 on device */dev/fd0.*

mkfs -3 /dev/fd0

The steps involved in writing file system are:

1. The *mkfs.c* first checks the options and arguments supplied to the program. If the option -3 is present, it writes new file system with modified directory structure that is version 3; otherwise, it writes existing file system of version 2.

2. When it sees -3, as an option, this program assign 3 to a local variable *fs_version*; otherwise 2.

3. Now it tries to calculate the size of the device in kilobytes using function *sizeup()*, if this value is not passed, explicitly, as an argument, that is, *mkfs -3 /dev/fd0 1440*. Where 1440 is size of device in kilobytes.

4. On the basis of the size obtained in step 3, it calculates the number of block that can be written on this device. Furthermore, the number of block is used to calculate the number of i-nodes required and number of zone present on the disk.

5. Number of i-nodes and zones helps in calculating the number of i-node bitmap and number of zone bit map required to handle all these i-nodes and zones. Number of i-nodes is also used to determine the number of i-node blocks required to store the i-nodes.

6. Important part of the file system writing is to write super-block. The *s_imap_blocks* and *s_zmap_blocks* store the number of i-node bitmap and number of zone bit map. It also stores the offset of first data zone in *s_firstdatazone*, which is easy to calculate from aforementioned information. The *s_log_zone_size* contains the result of logarithm of base 2 of blocks per zone. It also stores the maximum size of a file that can reside on this device, in *s_max_size*. Finally, depending on the file system chosen, it writes the magic number in the super-block at *s_magic*.

7. After that, *rootdir()* function writes root as well as dot(**.**) and dotdot(**..**) directories inside the root directory. This procedure is mentioned inside *enter_dir()* function. The function updates the file size inside root directory inode. The main modification in this utility is inside this function. This writes dot(**.**) and dotdot(**..**) directory name with their inode number that is in itself root

directory inode number for both entries. It also stores the directory entry length for both entries as well as name length of both entries, which in turn is 1 and 2.

4.1.3.2 Insertion, Look-up and Deletion of Directory Entry

The insertion, look-up and deletion of directory entries are performed inside *search_dir.c*. This file has a function *search_dir()* which takes four variables as an argument, these are:

- Pointer to inode for directory to search, that is, parent directory inode number; *ldir_ptr*
- Component to search for, that is, file name; *string*
- Pointer to inode number; *numb*
- Flag indicating, whether the process is insertion, look-up, deletion or just to check the emptiness of a directory; ENTER, LOOK_UP, DELETE or IS_EMPTY

Since, we are providing backward compatibility for the existing file system, we will have to provide the method for coexistence of both file system, that is, version 2 and version 3. When insertion, deletion or look-up is required for a given directory of inode pointer *ldir_ptr*, the first check is performed to the corresponding super-block's *s_version*. If, it returns the value V2 then the existing procedure is used; otherwise, procedure written for V3 (modified file system) is used.

If device has MInix file system version 3 i.e. V3, we perform another check to decide, if it is insertion, deletion or look-up process. We will look one by one all process. Lets, investigate the insertion or ENTER operation for new directory entry, in modified file system with modified directory structure.

1. If the i-node pointer *ldir_ptr* contains the zone numbers then acquire first directory block. If there is not a first block then acquire a data block (which will become a directory block) for directory entries.
2. Make a pointer to the first byte of the directory.

3. Check, if inode number of this entry is zero and space left in the block is greater than the coming directory entry.

   3.1. If yes, test if, after the insertion of current directory entry, there will be much space left for the other directory entry.

      3.1.1. If yes, make directory entry with zero inode and appropriate directory entry length, to signify that it may contain a directory entry at later stage.

      3.1.2. Insert the current directory entry with *d_ino* as inode number *numb, d_name* as file name *string* with *d_rclen* as appropriate directory entry length and *d_nmlen* as length of string *string*. Update i-node's time information, accordingly.

      3.1.3. Make the i-node and directory block flag "dirty" for the updation of corresponding disk information. Free the current directory block. Exit from the program with OK return signal.

   3.2. If no, test if, the directory entry length *d_rclen* of the directory entry is zero.

      3.2.1. If yes, it means end of the directory entries encountered. Update *d_rclen* of previous entry to contain length of the previous directory entry. If it is not a new block. Now, check if the same block has enough space for the current directory entry.

         3.2.1.1. If yes, make the directory entry in the same block with *d_ino* as inode number *numb, d_name* as file name *string*, with appropriate file name length *d_nmlen* as length of string *string* with directory entry length *d_rclen* zero, to signify end of the directory entries. Rest of the procedure is same as step 3.1.3.

         3.2.1.2. Otherwise, test if space left is not much enough to write any other directory entry.

3.2.1.2.1. If yes, update the *d_rclen* of previous entry to acquire the space left in the block.

3.2.1.2.2. If no, make a directory entry with *d_ino* inode number zero and *d_rclen* as size of the left space in directory block with *d_nmlen* as zero.

3.2.1.2.3. Make the directory block flag “dirty” for the updation of corresponding disk information. Free the current directory block.

3.2.1.2.4. Acquire a new data block, which will eventually become a directory block. Set the directory entry pointer to the first byte of the block. Write directory entry with *d_ino* inode number as *numb*, *d_name* storing string *string, d_rclen* as zero, signifying end of the directory entry, and *d_nmlen* as length of the string *string.*

3.2.1.2.5. Make the i-node and directory block flag “dirty” for the updation of corresponding disk information. Free the current directory block. Return from the program with OK signal.

3.2.2. If *d_rclen* is not zero, it means last directory entry might be in next block. So acquire the next directory block of the directory entries and set pointer to the first byte of this block. Then, go back to step 3.

The whole procedure can be summarized for ENTER operation as, we look for the free directory entry whose size is greater than the new directory entry or look for the end of directory entries. If a free directory entry of enough size is located, it is acquired and used by the new directory entry. If the acquired directory entry is too big to hold the new directory entry, it is divided into two parts. One is used by the new directory entry and other is become available as a free directory entry for later stage. If the end of directory entries is encountered then the new directory entry is written after the last directory entry. Of course, it may require acquisition of a new directory block.

The deletion and look-up process for file system version 3 are slightly similar. Both searches for the directory entry of a given file name *string*. When it is located in a directory block, LOOK-UP process returns the corresponding i-node number, while DELETE removes the directory entry. The IS_EMPTY is also not very different, it just checks whether directory is empty or not, that is, it contains other file or directory besides dor(**.**) and dotdot(**..**). Here is the procedure for these all three operations.

1. It is similar to first step of ENTER operation. If the i-node pointer *ldir_ptr* contains the zone numbers then acquire first directory block. If there is not a first block then return with error code ENOENT, which signify that "no entry exist for such file or directory". ENOENT is defined inside *errno.h*.

2. Make a pointer to the first byte of the directory.

3. Test if, the *d_rclen*, that is, directory entry length is not equal to zero.

   3.1. If yes, do following steps.

      3.1.1. Test if, the *d_ino* i-node number is not equal to zero and it is IS_EMPTY operation. If yes, compare file name *d_name* of current entry with dot(.) and dotdot(..). If result is false, then free the directory block and exit from the operation returning ENOTEMPTY signifying that "directory is not empty".

      3.1.2. Check if, it is not IS_EMPTY operation, current directory entry's *d_ino* inode number is not zero, and its *d_name* is same as we are looking for.

         3.1.2.1. If it satisfies, the all given conditions then check if it is a DELETE operation.

3.1.2.1.1. If yes, delete the content of *d_ino* inode number and set *d_name* to null string. Set the directory block as a dirty block.

3.1.2.1.2. Otherwise, It's a LOOK_UP operation so return the inode number *d_ino* of the directory entry.

3.1.2.1.3. Free the directory block and exit with return of OK.

3.1.3. If it does not satisfy all the condition of step 3.1.2 then check if, the next entry is in same block or in the next block.

3.1.3.1. If, it is in the next block then free the current directory block and read the next directory block. Set pointer to the first directory entry of the new directory block.

3.1.3.2. If, it is in the same block then move pointer to the beginning of next directory entry.

3.2. If the *d_rclen* is zero; it means, this is the last directory entry. Do the following steps.

3.2.1. Perform all steps from 3.1.1 and 3.1.2.1.3.

3.2.2. If it does not satisfy all the condition of step 3.1.2 then check if, it is a IS_EMPTY operation, if yes, free the directory block and return OK, otherwise, free the directory block and return ENOENT, that is, no entry of such file or directory found.

4. Go back to step 3.

Deletion and look-up process are not very different from the ENTER operation. These operations, too, start from the first directory block of a directory. Matching of file name with *d_name* is performed for each directory entry. If it matches, the LOOK_UP operation return I-node number and DELETE deletes the directory entry. It searches the directory block until end of directory entries is not encountered. If it does not match to any entry, it returns ENOENT. The IS_EMPTY is very simple operation. It just sees if any entry name is not equal to dot(**.**) or dotdot(**..**). If it founds a directory entry with different name then it returns ENOTEMPTY otherwise, returns OK.

Note that to support new file system, slight modification in function *read_super()* of *super.c* and *fsck.c* has been provided to support the file system and its check routine.

4.2 Fast Symbolic Link

Minix does not support fast symbolic link and symbolic link for the files located on two different devices in existing version. In this section, we will look at the implementation the fast symbolic link. This is a functional enhancement in the existing Minix file system.

4.2.1 Modification of I-node Structure

The concept behind creating a fast symbolic link is to use the unused space of an inode. When we create a link as a hard link it has a major problem that it cannot be used for the files located on two different devices as mentioned in chapter three. To solve this problem, symbolic link are used. However, the symbolic link uses an i-node and a data block to write the path inside the data block. Thus, the symbolic link is a special file, which contains a path of a file or directory. This path is traversed at run time to use the target file or directory. In the existing file system the maximum path length possible is 255 characters but in this modified file system it is 1023 characters. So, a symbolic link will need, at most, a data block to write the path name. The problem with the symbolic link is, it need double reference of disk, first for i-node and second is for data block. Another problem is, if the path name is very short, it still needs a data block and an i-node. This problem has been removed with the use of fast symbolic link. Symbolic link uses i-node to store the path name of the target file or directory. As we know, i-node has 4 X 10 = 40 bytes of space for the storage of direct and indirect zone numbers. Fast symbolic link utilizes this space for the storage of path names, if the number of characters in path name is less than 40. Thus, without using a data block it provides the property of a symbolic link. Since it uses i-node structure for storage of target path name, lets take a view of i-node structure and changes made inside i-node structure.

```
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)*/
  /*************************************************/
  union
  {
        zone_t  i__zone[V2_NR_TZONES];   /*zone numbers for direct, ind, and dbl ind */
        char    i__data[V2_NR_CHARS];            /* storage space for fast symbolic link */
  }i;
  /*************************************************/
    ………………. /* there are few more entries */
    ……………….
} inode[NR_INODES];

#define  i_zone    i.i__zone
#define  i_data    i.i__data
```

In existing Minix, the i-node structure contains element *i_zone[V2_NR_TZONES]* instead of *union i*. This union provide access of same space with different name, as V2_NR_CHARS is defined in *fs/const.h* as:

```
#define  V2_NR_TZONES              10
                                           /* total number zone numbers in a V2 inode */
#define  V2_ZONE_NUM_SIZE          usizeof (zone_t)
                                           /* number of bytes in V2 zone  */
#define  V2_NR_CHARS               (V2_ZONE_NUM_SIZE * V2_NR_TZONES)
                                           /*number of character space in inode*/
```

The changes made inside the *include/const.h* is

```
#define   I_FAST_SLINK 0050000                   /* fast symbolic link */
#define   I_SYMBOLIC_LINK 0001000        /* symbolic link */
```

Macro I_FAST_LINK and I_SYMBOLIC_LINK has been used to identify the i-node mode type, that is, if it is a fast symbolic link or a simple symbolic link. There are some changes in macro definitions of *include/sys/stat.h*. However, these are not of a great importance.

In the *link.c*, the implementation of fast symbolic link and symbolic link has been provided. In the existing file system, only hard link has been implemented in *link.c* but there have been many modification inside the function *do_link()* to implement fast symbolic link and symbolic link for the files located on two different devices. This routine automatically takes decision that whether to use fast symbolic link or simple symbolic link. The command for symbolic link creation is

ln   target_path   link_path

where *target_path* is the path name of the target file or directory and *link_path* is the path name of the link.

4.2.2 Implementation

To make a symbolic link implementation steps are:

1. Extract the target path name and link path name from command.
2. Check, if target file or directory name exists. If it does not exist, then return with error message and exit.
3. Check, if the target is directory and super user is not using the command, if so, return with an error message and exit.
4. Check, if link file name already exist in the directory. If yes, return with appropriate error message and exit.
5. If everything is fine, check, if the files are located on two different devices.
    - 5.1. If yes, check if target path name is absolute or not. If it is not an absolute path, exit with an error message "*Absolute path required for the source*".
        - 5.1.1. Now check, if target path name length is less than the zone space size of an inode, that is, V2_NR_CHARS.

5.1.1.1. If it is less than the V2_NR_CHARS, set a flag and mode bits indicating fast symbolic link.

5.1.1.2. If it is greater than the V2_NR_CHARS, set a flag and mode bits indicating symbolic link.

5.1.2. Acquire a new i-node, set its mode bits, and size equal to the path name length of the target file or directory.

5.1.3. Check, If the flag indicates that we are implementing fast symbolic link.

5.1.3.1. If yes, write the target path name inside inode.

5.1.3.2. Otherwise, acquire a data block and copy target path name inside the data block. Set the dirty bit of data block and free it.

5.1.4. The i-node dirty bit is set and its corresponding entry is updated on disk.

5.2. Otherwise, if the target and link files are on same device, make a hard link with existing implementation.

This code writes a fast symbolic link on a device. However, we need a method to traverse and use the target file using this link.

The function *slink_traverse()* does the traversal for the fast symbolic link and symbolic link. It has been implemented inside *path.c* file. The *eat_path()* function of the existing Minix file system inside *path.c* uses this *slink_traverse()* function to traverse the link if it founds any link. Otherwise, it uses old approach of path traversal. To incorporate the symbolic link traversal few changes have been implemented inside *path.c*. Similarly, some changes have been implemented inside *open.c* for the creation of an i-node. If the path provided as argument to this function contains a link in the path then its traversal is performed using *slink_traverse()* function.

CHAPTER 5 DENTRY OBJECT AND CACHE

In this chapter, we will discuss the design and implementation of directory entry cache or dentry cache. These two names will be interchangeably used inside this dissertation. To use this dentry cache, the formation of a directory entry object from directory entry is performed, which is retrieved after disk read.

In the standard Minix file system, for every look-up of directory entry disk read is performed, that is, even if a directory entry is in use very frequently, it has to perform disk read. This situation may occur when a user performs some frequent operations on a file or if he is using a directory, quite frequently. As mentioned in chapter 2, the disk access is slower by order of 100,000 than main memory access. Therefore, it will be useful to cache the directory entry information in main memory. Thus, improvement in terms of speed can be achieved for access of directory entry information, at the cost of some space for the directory entry cache or dentry cache inside main memory. Let us start design of dentry cache with directory entry object, which will be stored inside, this cache.

5.1 Dentry Object

Dentry object or directory entry object is the information or data that is created from the directory entry located on disk. Directory entry object has following structure.

```
EXTERN struct direntry
{
        dev_t             d_dev;                    /*whose dir block is this*/
        int               d_inumb;                  /*inode number associated with dir */
        int               d_pnum;                   /*inode number of parent directory */
        struct direntry   *d_next;                  /*pointer to the next dentry*/
        struct direntry   *d_prev;                  /*pointer to the previous node*/
        struct direntry   *d_hash;                          /*pointer to the hash chain*/
        char              d_name[V3_NAME_MAX];              /*filename or directory name*/
}direntry[NR_DIRENT];

EXTERN struct direntry    *dir_front;               /*points to the least recently used free block*/
EXTERN struct direntry    *dir_rear;                /*points to the most recently used free blocks */

#define NIL_DIRENT         (struct direntry *)0      /*null pointer for directory entry block*/
#define D_HASH_MASK  (NR_DIR_HASH - 1)              /* mask will be use to find the hash value */

EXTERN struct direntry    *dir_hash[NR_DIR_HASH]; /* dentry hash table */
```

The structure name is *direntry*. The *d_dev* is used to hold the device number. Without this information, it may be difficult to identify the files, if they have the same symbolic name and same parent directory i-node number but they are situated on different device. In design, care has been taken for the case that two same file name, same parent directory i-node number with same device number does not exist.

The *d_inum* is the field to hold the inode number of directory or file, which is retrieved from directory entry through disk. When a request for i-node number is required for a given file name, this is the element which is returned. In short, it is the field in directory entry cache, where i-node number of directory entry is stored.

The *d_pnum* is the field to store the parent directory inode number of a given directory entry. Since, it will be very difficult to determine the i-node number of a file or directory on the basis of their symbolic name. There is very high possibility of similar name file or directory in different directory. Thus, at a given time, these same filename files might exist in directory entry cache, simultaneously. In this case, it will be very difficult to determine the correct i-node number of the file or directory. Special care is taken for insertion of a duplicate symbolic name file or directory inside a given directory. Thus, in a given directory, two files or directories does not exist with same symbolic name. Therefore, we will never encounter files or directories with same symbolic name and same parent directory i-node number except they are on different device.

The *d_next, d_prev,* and *d_hash* are the pointer to the other dentry objects or directory entry objects. First two pointers provide a means to implement a doubly linked list of the dentry objects. The last pointer *d_hash* is used to point the dentry object with same hash value. This pointer will help in search of a dentry object, if the hash value obtained after the application of hash function on their parent directory i-node number is same.

The *d_name* is used to store symbolic name of the file or directory inside a dentry object. This is the field, which is matched by other file name to return i-node number, if they match (of course, other constraint must be satisfied).

The *dir_front* and *dir_rear* are pointer to the front and rear end of the doubly linked list directory entry cache. This doubly linked list, maintain the free dentry object block. Dentry object block is a part of dentry cache that stores one directory entry object.

The above given structure has been written inside *dentry.h.*

The D_HASH_MASK has been used to find the hash value. Bitwise *AND* operation of this macro with parent directory i-node number results into a hash value, which is further used to make entry inside a hash table. The macro NR_DIR_HASH is defined inside *const.h.* Other macros that are defined in *const.h* but used in the design and implementation of directory entry cache are:

```
#define  NR_DIRENT       128                         /* # of blocks in the dir cache */
#define  NR_DIR_HASH    NR_DIRENT/2                  /* size of dir hash table */

#define  FOUND_D          4        /* entry found in the dentry table*/
#define  NFOUND_D         5        /* entry not found in the dentry table*/
#define  ENTER_D          6        /* tells search_dentry to enter dir entry*/
#define  DELETE_D         7        /* tells search_dentry to delete a dentry */
```

NR_DIRENT is macro, which defines the maximum number of dentry objects that can reside in directory entry cache at a time, that is, it is the maximum number of directory entry objects. This value is slightly bounded by the number of inode table entries. The value must be equal to NR_INODES unless there are too many links inside the file system. If the file system has to many links then it means it may use many dentry object in dentry cache for the same inode number with different file name.

Macro NR_DIR_HASH has been used to define the number of buckets in the hash table of directory entry cache. Its size depends on the NR_DIRENT macro. If, assume, the directory entry cache is full, then for successful search of a directory entry object will take

$1 + lf/2$

probes, on average. Where,

$lf = N/M$;

N is the number of directory entry object in the table and M is the number of hash keys or hash bucket list, that is, NR_DIR_HASH. The *lf* is called load factor of hash table. Maximum value of N can be equal to NR_DIRENT so the maximum value of *lf* can be NR_DIRENT/NR_DIR_HASH, which is 2 in this case. Thus for successful search, probes required on this hash table is 1+2/2 = 2.

Unsuccessful search on the hash table requires, where collision is resolved by chaining,

exp(-lf) + lf

probes on average. This expression results into 2.135 or almost 3 probes. These values are fair for our implementation [8][9].

Macro FOUND_D is returned if the item to be searched is found in the directory entry cache on search. Similarly, NFOUND_D is returned if the item is not found in the directory entry cache. The ENTER_D flag tells to the function *search_dentry()* to enter the directory entry object inside the directory entry cache. The DELETE_D flag tells to the function *search_dentry()* to delete the directory entry object from the cache if the directory entry object is invalid.

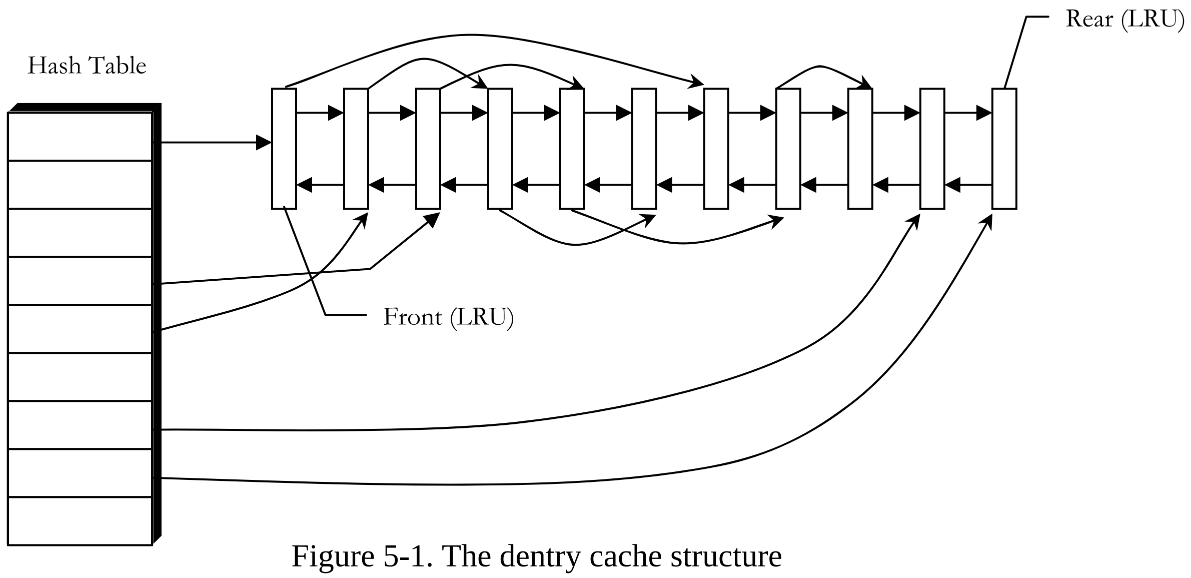


Figure 5-1. The dentry cache structure

5.2 The Dentry Cache

The directory entry cache has been implemented as an array of buffers; each consisting of a header containing pointers and some variables to store directory entry information, that is, each of this stores a directory entry object or dentry object. All the buffers that are not in use are chained together in a doubly linked list, from most recently used (MRU) to least recently used (LRU) as illustrated in figure 5-1.

In addition, to be able to quickly determine if a given dentry object block is in the cache or not, a hash table is used. All the buffers containing a block that has hash code *k* are linked together on a single linked list pointed to by entry *k* in the hash table. The hash function just extracts the low order *n* bits from the block number using D_HASH_MASK macro, so blocks from different devices appear on the same hash chain. Every buffer is on one of these chains.

5.2.1 Dentry Cache Initialization

When the file system is initialized after Minix is booted, all dentry object buffers are unused. At this time, *load_dentry()* function is called for the initialization of the dentry object buffers or dentry cache. This function makes a doubly liked list of all NR_DIRENT dentry object buffers. Initially, all dentry object buffers are in a single chain pointed to by the 0th hash table entry. All the other hash table entries, that is, 1 to (NR_DIR_HASH -1) contain a null pointer, but once the system starts, buffers will be removed from the 0th chain and other chains will be built. This *load_dentry()* function is called from *fs_init()* function. Both of these functions are defined inside *main.c* file.

5.2.2 Dentry Cache at Work

As elaborated in last chapter, the actual insertion, deletion and look-up for a directory entry is performed inside *search_dir.c* file or in *search_dir()* function. So this is the place where we must use dentry cache. In *search_dir()* function, initially *search_dentry()* function is called with input argument, parent directory inode pointer, inode number, file or directory name with flag for insertion, deletion or look-up. If, this function returns FOUND_D, it means we can return from the *search_dir()* function without actually looking the directory entry of the disk, that is, we have obtained the required information from the dentry cache. Otherwise, actual disk read or write is performed for the directory entry on disk.

If the deletion of a directory entry on disk is performed, it is necessary to delete the corresponding entry in dentry cache; otherwise, on next look-up it may return invalid inode number. So *search_dir()* function is called with DELETE_D flag. We will discuss in detail about this function in coming paragraph.

Similarly, when first time look-up of i-node number or search process is performed for a given directory or file name, the *search_dentry()* function makes an entry inside the dentry cache for the currently acquired directory entry information from disk. At this time *search_dentry()* function is called with ENTER_D flag. Next look-up or search of inode number for the same file or directory are handled by *search_dentry()* function with LOOK_UP flag. It returns the inode number of the file or directory with a flag FOUND_D to indicate that directory entry is available inside dentry cache and it is returning valid i-node number. If, it returns something other than FOUND_D, it means that the required file or directory inode number is not present in dentry cache. Therefore, it needs disk read for the directory entry.

If the *search_dentry()* function is called with ENTER or DELETE flag, it returns the not OK to indicate that no operation will be performed on dentry cache.

For all these operations, *search_dentry()* does not perform the actual job. The actual job is done by *get_dentry()* function defined inside the same file *search_dentry.c*. The *get_dentry()* takes the similar argument as *search_dentry()* function. This *get_dentry()* function calls a *dir_rm_lru()* function at some stages. The main function of this *dir_rm_lru()* function is to remove the directory entry object buffer from the 0th hash chain or more precisely from the LRU chain. The argument passed to this function is the pointer to the dentry object buffer, which is going to be removed from the LRU chain.

The job performed by the *get_dentry()* function can be roughly termed in following steps.

1. The *get_dentry()* function takes as argument, an inode pointer to parent directory, file or directory name, inode number of the file or directory passed as second argument and a flag indicating if it is a look-up (LOOK_UP), or insertion (ENTER_D) or deletion (DELETE_D) operation. This function returns 0 if the entry is not found inside the directory entry cache; otherwise, it returns 1.

2. First, this function checks if the parent directory i-node's device number is not zero. If it is zero, it throws a message indicating invalid inode number and exits this function. Otherwise, it goes to the next step.

3. With the help of D_HASH_MASK, hash key of the parent directory i-node number is obtained. Corresponding to this hash key, hash table bucket's pointer of the corresponding hash-chain is obtained.

4. On this hash-chain search is performed on the basis of device number, parent directory inode number and the file or directory name of the dentry object buffer entry and argument passed, until NIL_DIRENT, that is, end of chain is not encountered or matched directory object in cache is found.

5. If the flag is LOOK_UP or DELETE_D, that is, operation to be performed is look-up (search) or deletion on dentry object and on search, the dentry cache has not any entry corresponding to the argument passed entry, results into exit of function with return value zero, indicating no entry found.

6. If the flag is ENTER_D, that is, operation to be performed is insertion into dentry cache and on search, the dentry cache has not any entry corresponding to the argument passed entry, front-most dentry object of the LRU chain is acquired.

7. For all the flag LOOK_UP, DELETE_D and ENTER_D, if on search, dentry object is found corresponding to the argument passed, It is acquired.

8. The acquired dentry object (in step 6 or 7) is removed from the LRU chain using function *dir_rm_lru()*.

9. Find the hash key of the acquired dentry object using its parent directory inode number that is provided by *d_pnum* and D_HASH_MASK. Now on the basis of this hash key, hash tables bucket of the key is found. Here we get the front pointer of the hash chain. The acquired dentry object is removed from the hash chain.

10. If the deletion operation (DELETE_D) is to be performed on the dentry cache, device number (*d_dev)*, parent directory inode number *(d_pnum)*, and inode number *(d_inum)* is set to zero of the acquired dentry object. Moreover, this acquired dentry object buffer is put at the front of the LRU chain, so that it can be acquired if any further insertion of directory entry takes place inside dentry cache. After these operations exit from this function is performed with return value 1.

11. If the operation is insertion, the dentry object's *d_dev* is set to parent directory's device number, the *d_pnum* is set to parent directory's inode number, the *d_inum* is set to inode number of the file or directory and the *d_name* is set to the file or directory name.

12. If the operation is LOOK_UP, inode number of the file or directory is returned from the dentry object buffer. But, disk read function is not performed.

13. Now put this dentry object buffer on the front of hash bucket chain of corresponding hash key, so that next search will take less time.

14. Put this block on the rear end of LRU chain, so that it will stay here for a longer time. Now exit the function with return value zero.

5.2.3 Use of LIRS algorithm

Least recently used (LRU) algorithm is as good as replacement policy but it is not very good for the weak locality workloads [10]. There is a modification in LRU policy has been provided as LIRS algorithm. This algorithm has been implemented to replace the dentry object in the dentry cache. The LIRS algorithm maintains two queues, one for low-high inter-recency objects (S queue) and other for high inter-recency object (Q queue). Recency has been defined as the number of distinct dentry object that has been called between two consecutive call of a dentry object. That is, each debtry object has its recency information. This recency information helps determining the removal of a dentry object. When a dentry object is required, it is searched inside the S queue which contains information about low inter recency and high inter recency object information. But Q queue store information about high inter recency object information. So, the recency has been used to find the eviction of a dentry object in the dentry cache.

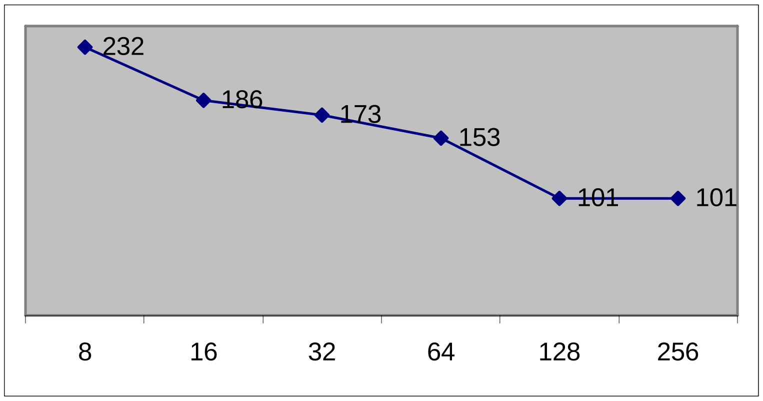
5.2.4 Comparison

The use of dentry cache will provide the less number of disk-reads at the cost of main memory space. Since the disk reads takes more time than main memory access, which is in order of 10,000. Thus, it may cause improvement in the terms of average time of directory entry read operation. In this section, it has been shown the comparison of disk reads for few specific operations before use of dentry cache and after the use of dentry cache.

When system boots, it performs many read for different directory entries. Following is the detail of disk reads after root login is performed on the system.

NR_DIRENT (number of object space in dentry cache)	Number of disk reads before use of dentry cache	Number of disk reads after use of dentry cache	Percentage decrement in disk reads
8	611	232	(611-232)/611=62.02%
16	611	186	69.55%
32	611	173	71.68%
64	611	153	74.95%
128	611	101	83.46%
256	611	101	83.46%

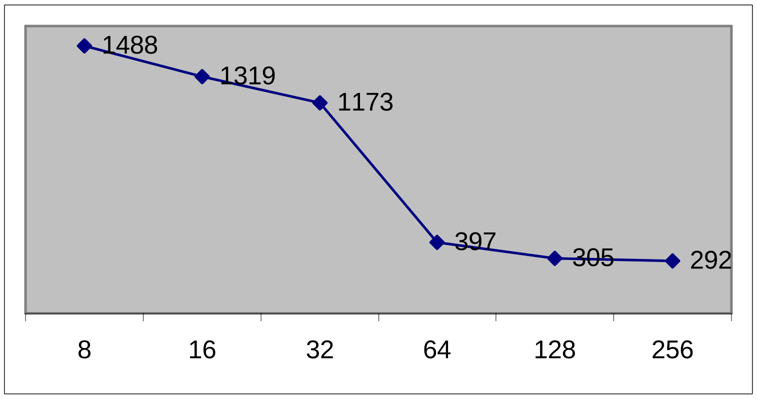
Last two entries has the same disk read, because that is the number of distinct directory entries read from the disk, and all of them are residing in main memory. The following chart displays the decrement in disk read after the increment in the value of NR_DIRENT for above given operation.



Compilation of the kernel is a very complex process that involves read of many directory entries. So following is the observation after a file system compile is performed on the Minix system.

NR_DIRENT (number of object space in dentry cache)	Number of disk reads before use of dentry cache	Number of disk reads after use of dentry cache	Percentage decrement in disk reads
8	5267	1488	71.74%
16	5267	1319	74.95%
32	5267	1173	77.72%
64	5267	397	92.46%
128	5267	305	94.20%
256	5267	292	94.45%

We can see from the both table that use of dentry cache has reduced the disk read by a significant amount. Thus, speed up might be quite visible in multi-user environment. The following graph displays the decrement in disk read after increase in NR_DIRENT for above operation.



CHAPTER 6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this dissertation, a design has been proposed for the directory management in the Minix file system. During the course of this project, existing directory management policy of the Minix file system has been modified to support functional and performance related enhancement. During the course of this project, file system of Minix operating system has been studied in detail and the new design has been proposed to support variable length long file name, fast symbolic link as functional improvements and use of directory entry object with directory entry cache for performance enhancement.

Use of long file name lifts the restriction of fourteen-character file name from the existing system. Moreover, it provides variable size directory entry structure for the storage on the disk, which is of fixed size in standard Minix file system (as explained earlier). Thus, improves the file system capability.

Fast symbolic link design and implementation has been provided. In the standard Minix file system, hard link was the only method to carry out linking between two files with a restriction of files must be on same device. However, in our implementation this limit has been hoisted. It, now, establishes fast symbolic link or simple symbolic link for the two files located on two different devices, depending on their path name length. The use of every fast symbolic link saves disk space of one data block (It has the size of 1024 bytes in standard Minix file system), which is a major improvement in terms of free disk space.

Another improvement provided in terms of performance is, use of directory entry cache. As explained earlier, in standard Minix file system, every time i-node number of a file or directory is required, it performs many disk read and so delays the throughput. However, in this implementation, a directory entry cache has been used, which resides in the main memory, and contains the file name or directory name with its i-node number, its parent i-node and device number. Directory entry cache uses LRU technique for the replacement of a directory entry with new directory entry. With use of this cache, only first reference to a directory entry requires disk read. Further reference to the same file name does not require disk read until it goes out of the directory cache.

The design, as proposed in this dissertation, is also successfully implemented, installed and tested with Minix operating system and it is in a ready to use stage. This complete implementation consists of several changes in the standard Minix file system, presenting all of which on this dissertation will deviate the focus from the primary changes which are the building blocks of this proposed design. Instead the focus of this dissertation is to present these primary changes in an elaborate manner.

6.2 Future Work

Directory entry cache uses LRU technique for the replacement of a directory entry information, that is, a directory object. As a future work, different replacement algorithm such as LRFU[25], 2Q[26], ULC[22], etc. can be implemented and their performance can be analyzed on the directory cache. For further work, best of these algorithms can be implemented on buffer cache of Minix file system.

Currently, Minix operating system supports only its own file system. Its design can be extended to support other file systems. Since the directory entries of other file systems differ in their structure for example, file name size might be different, or it has no i-nodes and data block numbers are itself in the directory entry. Therefore, the directory handling for those file system must be in accordance with their structure of directory entry.

Another, improvement possible is, currently Minix file system directory hierarchy is of acyclic-graph type. General-graph directory structure can be implemented, however, it is not very useful except that it provides fast method for traversal inside whole directory structure.

REFERENCES

1. **A. Silberschatz and P.B. Galvin:** ***Operating System Concepts,*** John Wiley and Sons, Inc., 2000

2. **Harvey M. Deitel:** ***An Introduction to Operating System,*** Addison-Wesley Publishing Company, 1990

3. **R. C. Daley and P. G. Neumann:** ***A General-Purpose File System for Secondary Storage,*** Fall Joint Computer Conference, 1965

4. **James Sanders:** ***Linux, Open Source, and Software's Future,*** IEEE software, September/October, 1998

5. **A. S. Tanenbaum and A. S. Woodhull:** ***Operating Systems Design and Implementation,*** Pearson Education, 2004

6. ***Linus vs. Tanenbaum,*** http://people.fluidsignal.com/~luferbu/misc/Linus_vs_Tanenbaum.html

7. **Brian W. Kernighan and Dennis M. Ritchie:** ***The C programming language,*** Prentice Hall of India, 1994

8. **Yedidyah Langasm, Moshe J. Augenstein, Aaron M. Tanenbaum:** ***Data Strcutures using C and C++,*** Prentice Hall of India, 1999

9. **Donald E. Knuth,** ***The Art of Computer Programming: Sorting and Searching,*** Pearson Education, 2004

10. **M. J. Bach:** ***The Design of the UNIX Operating System,*** Prentice Hall of India, 1987

11. **Daniel P. Bovet and Marco Cesati:** ***Understanding the Linux Kernel,*** O'Reilly Publication, 2004

12. **Richard Stones and Neil Matthew:** ***Beginning Linux Programming,*** WROX Publishers, 2002

13. **W.R. Stevens:** ***Advanced Programming in the UNIX Environment,*** Addison-Wesley, 1992

14. **U. Vahalia:** ***UNIX Internals-The New Frontiers***, Prentice Hall, 1996

15. **W. Stallings:** ***Operating Systems,*** Prentice Hall of India, 1995

16. **Andrew S. Tanenbaum and Marteen van Steen:** ***Distributed Systems: Principal and Paradigms,*** Pearson Education, 2004

17. **Song Jiang and Xiaodong Zhang:** ***Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance,*** IEEE Transactions on Computers, 2005

18. **K. Thompson:** ***UNIX Implementation,*** Bell System Technical Journal, 1978

19. **John S. Heidemann and Gerald J. Popek:** ***File System Development with Stackable Layers,*** ACM Transactions on Computer Systems, 1993

20. **Mohamed Mohy El Din Mahmoud and Amr El-Kadi:** ***A DOS/Linux Extensible File System,*** IEEE Transactions on Computers, 1997

21. **Butler W. Lampson and Robert F. Sproull:** ***An Open Operating System for a Single-User Machine,*** ACM Operating Systems Review, 1979

22. **Song Jiang and Xiaodong Zhang:** ***ULC: A File Block Placement and Replacement Protocol to Effectively Exploit Hierarchical Locality in Multi-level Buffer Caches,*** Proceedings of the 24th International Conference on Distributed Computing Systems, 2004

23. **Margo Seltzer, Keith Bostic, Marshall Kirk McKusick and Carl Staelin:** ***An Implementation of a Log- Structured File System for UNIX,*** Winter USENIX, 1993

24. **Jinhyuk Yoon, Sang Lyul Min and Yookun Cho:** ***Buffer Cache Management : Predicting the Future from the Past,*** Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks, 2002

25. **D.Lee, J.Choi, J.Kim, S.H.Noh, S.L.Min, Y.Cho and C.S.Kim:** ***LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies,*** IEEE Transactions on Computers, 2001

26. **T. Johnson and D.Shasha,** ***2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm,*** Proceedings of the 20th International Conference on VLDB, 1994

27. ***Marshall Kirk McKusick, William N. Joy, Samuel J. Leflller and Robert S. Fabry:*** **A Fast File System for UNIX,** *ACM Transactions on Computer Systems, 1984*

28. **Minix Homepage:**
http://www.cs.vu.nl/~ast/minix.html

29. **Minix Usenet Newsgroup:**
comp.os.minix

30. **Other Minix Related information:**
http://www.minix1.hampshire.edu

```
/*****************************************************************************************/
                                                            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__v2_dir[V2_NR_DIR_ENTRIES];                 /* directory block */
char b__vardirsz[BLOCK_SIZE];
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 */
} 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 */
block_t 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_v2_dir        b.b__v2_dir
#define b_v3_dir        b.b__v3_dir
#define b_vardirsz      b.b__vardirsz
#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

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

EXTERN struct buf *front;         /* points to ledce 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 */

/*****************************************************************************************/
                                                           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 NR_DIRENT        128    /* # of blocks in the dir cache */
#define NR_DIR_HASH               64            /* size of dir hash table */

/* The type of sizeof may be (unsigned) long.  Use the following macro for
 * taking the sizes of small objects so that there are no surprises like
 * (small) long constants being passed to routines expecting an int.
 */
#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 SUPER_V3      0x345F
#define SUPER_V3_REV  0x5F34


#define V1                  1           /* version number of V1 file systems */
#define V2                  2           /* version number of V2 file systems */
#define V3                  3          /*version number of v3 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 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 FOUND_D                   4             /* entry found in the dentry table*/
#define NFOUND_D                  5              /* entry not found in the dentry table*/
#define ENTER_D                   6              /* tells search_dentry to enter dir entry*/
#define DELETE_D    7

#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 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 V2_NR_CHARS     (V2_ZONE_NUM_SIZE * V2_NR_TZONES)
                                                                   /*number of character space in inode*/
#define V2_DIR_ENTRY_SIZE  usizeof(struct direct)  /* # bytes/dir entry for v2 */
#define V3_DIR_ENTRY_SIZE  usizeof(struct direct3)              /* # bytes/dir entry for v3 */
#define V2_NR_DIR_ENTRIES  (BLOCK_SIZE / V2_DIR_ENTRY_SIZE)                /* # dir entries/blk v2 */
#define V3_NR_DIR_ENTRIES  (BLOCK_SIZE / V3_DIR_ENTRY_SIZE)                /* # dir entries/blk v3 */
#define printf printk
struct direct4
{
        ino_t d_ino;
        off_t d_rclen;
        unsigned short d_nmlen;
        char d_name[V3_NAME_MAX];
};
#define OFFNAME    (((struct direct4 *) (0))->d_name)
                                                                          /* Offset of Name */
#define OFFSET(a)    (BLOCK_SIZE - ( a % BLOCK_SIZE))
                                                                          /* Back Block Offset */`
#define GRPFY(x)     (((int) (( MIN( x, V3_NAME_MAX) + 3)/4))*4)
                                                          /*Groupify the data byte in 4byte multiple*/




/*******************************************************************************************/
                                                      dentry.h
/*******************************************************************************************/
                                           /*This file is used for defining dentry table*/

EXTERN struct direntry
{
        dev_t               d_dev;              /*whose dir block is this*/
        int                 d_inumb;            /*inode number associated with dir */
        int                 d_pnum;             /*inode number of parent directory */
        struct direntry *d_next;                /*pointer to the next dentry*/
        struct direntry *d_prev;                /*pointer to the previous node*/
        struct direntry *d_hash;                /*pointer to the hash chain*/
        char         d_name[NAME_MAX];          /*filename or directory name*/
}direntry[NR_DIRENT];

#define NIL_DIRENT (struct direntry *)0                 /*null pointer for directory entry block*/

EXTERN struct direntry *dir_hash[NR_DIR_HASH];   /* dentry hash table */

EXTERN struct direntry *dir_front;                          /*points to the ledce recently used free block*/
EXTERN struct direntry *dir_rear;                           /*points to the most recently used free blocks */

#define D_HASH_MASK          (NR_DIR_HASH - 1)              /* mask will be use to find the hash value */


/*******************************************************************************************/
                                                        fs.h
/*******************************************************************************************/
/* This is the mdceer 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"

/************************************************************************************************/
                                                  glo.h
/************************************************************************************************/
/* EXTERN should be extern except for the table file */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

/* File System global variables */
EXTERN struct fproc *fp;                   /* pointer to caller's fproc struct */
EXTERN int super_user;                     /* 1 if caller is super_user, else 0 */
EXTERN int dont_reply;                     /* normally 0; set to 1 to inhibit reply */
EXTERN int susp_count;                     /* number of procs suspended on pipe */
EXTERN int nr_locks;                       /* number of locks currently in place */
EXTERN int reviving;                       /* number of pipe processes to be revived */
EXTERN off_t rdahedpos;                    /* position to read ahead */
EXTERN struct inode *rdahed_inode;         /* pointer to inode to read ahead */

EXTERN int fsys_ver; /*store version type*/
EXTERN char *errmsg;

/* The parameters of the call are kept here. */
EXTERN message m;               /* the input message itself */
EXTERN message m1;                         /* the output message used for reply */
EXTERN int who;                            /* caller's proc number */
EXTERN int fs_call;             /* system call number */
EXTERN char user_path[PATH_MAX];/* storage for user path name */

/* The following variables are used for returning results to the caller. */
EXTERN int err_code;                       /* temporary storage for error number */
EXTERN int rdwt_err;                       /* status of ldce disk i/o request */

/* Data which need initialization. */
extern _PROTOTYPE (int (*call_vector[]), (void) );      /* sys call table */
extern int max_major;                                            /* maximum major device (+ 1) */
extern char dot1[2];                                    /* dot1 (&dot1[0]) and dot2 (&dot2[0]) have a special */
extern char dot2[3];                                    /* meaning to search_dir: no access permission check. */

/************************************************************************************************/
                                                  lirs.h
/************************************************************************************************/

#define NR_LIR 100
#define NR_HIR 28

#define IN_USE                     1
#define NOT_IN_USE                 2
#define LIR                                  3
```

```
#define HIR_P                        4
#define HIR_NP                       5


int lirs_in_use;
int hir_in_use;
int lir_in_use;

struct lirs_q
{
        int block;
        int dev;
        short flag;
        struct lirs_q *next, *prev;
}lirs[NR_BUFS];

struct lirs_q *lirs_front,*lirs_rear;

struct hir_q
{
        int block;
        int dev;
        short flag;
        struct hir_q *next, *prev;
}hir[NR_HIR];

struct hir_q *hir_front,*hir_rear;


/*********************************************************************************************/
                                        proto.h
/*********************************************************************************************/


/* Function prototypes. */

/* Structs used in prototypes must be declared as such first. */
struct buf;
struct filp;
struct inode;
struct super_block;

/* device.c */
_PROTOTYPE( void call_task, (int task_nr, message *mess_ptr)            );
_PROTOTYPE( void dev_opcl, (int task_nr, message *mess_ptr)             );
_PROTOTYPE( int dev_io, (int rw_flag, int nonblock, Dev_t dev,
                        off_t pos, int bytes, int proc, char *buff)     );

/* inode.c */
_PROTOTYPE( struct inode *alloc_inode, (Dev_t dev, Mode_t bits)         );
_PROTOTYPE( void dup_inode, (struct inode *ip)                          );
_PROTOTYPE( void free_inode, (Dev_t dev, Ino_t numb)                    );
_PROTOTYPE( struct inode *get_inode, (Dev_t dev, int numb)              );
_PROTOTYPE( void put_inode, (struct inode *rip)                         );
_PROTOTYPE( void update_times, (struct inode *rip)                      );
_PROTOTYPE( void rw_inode, (struct inode *rip, int rw_flag)             );
_PROTOTYPE( void wipe_inode, (struct inode *rip)                        );

/* link.c */
_PROTOTYPE( int do_link, (void)                                                 );
_PROTOTYPE( int do_unlink, (void)                                               );
_PROTOTYPE( int do_rename, (void)                                               );
_PROTOTYPE( void truncate, (struct inode *rip)                                  );


/* main.c */
_PROTOTYPE( void main, (void)                                                   );
_PROTOTYPE( void reply, (int whom, int result)                                  );
```

```
/* mount.c */
_PROTOTYPE( int do_mount, (void)                                              );
_PROTOTYPE( int do_umount, (void)                                             );

/* open.c */
_PROTOTYPE( int do_close, (void)                                              );
_PROTOTYPE( int do_creat, (void)                                              );
_PROTOTYPE( int do_lseek, (void)                                              );
_PROTOTYPE( int do_mknod, (void)                                              );
_PROTOTYPE( int do_mkdir, (void)                                              );
_PROTOTYPE( int do_open, (void)                                               );

/* path.c */
_PROTOTYPE( struct inode *advance,(struct inode *dirp, char string[NAME_MAX]));
_PROTOTYPE( int search_dir, (struct inode *ldir_ptr,
                        char string [NAME_MAX], ino_t *numb, int flag));
_PROTOTYPE( struct inode *eat_path, (char *path)                    );
_PROTOTYPE( struct inode *ldce_dir, (char *path, char string [NAME_MAX]));


/* protect.c */
_PROTOTYPE( int forbidden, (struct inode *rip, Mode_t access_desired)       );
_PROTOTYPE( int read_only, (struct inode *ip)                                 );


/* read.c */
_PROTOTYPE( int do_read, (void)                                               );
_PROTOTYPE( struct buf *rahead, (struct inode *rip, block_t baseblock,
                        off_t position, unsigned bytes_ahead)                 );
_PROTOTYPE( void read_ahead, (void)                                 );
_PROTOTYPE( block_t read_map, (struct inode *rip, off_t position)             );
_PROTOTYPE( int read_write, (int rw_flag)                                     );
_PROTOTYPE( zone_t rd_indir, (struct buf *bp, int index)                      );


/* super.c */
_PROTOTYPE( bit_t alloc_bit, (struct super_block *sp, int map, bit_t origin));
_PROTOTYPE( void free_bit, (struct super_block *sp, int map,
                                                   bit_t bit_returned)        );
_PROTOTYPE( struct super_block *get_super, (Dev_t dev)              );
_PROTOTYPE( int mounted, (struct inode *rip)                                  );
_PROTOTYPE( int read_super, (struct super_block *sp)                          );


/* write.c */
_PROTOTYPE( void clear_zone, (struct inode *rip, off_t pos, int flag)       );
_PROTOTYPE( int do_write, (void)                                              );
_PROTOTYPE( struct buf *new_block, (struct inode *rip, off_t position)      );
_PROTOTYPE( void zero_block, (struct buf *bp)                       );

/***************************************************************************************************/
                                              super.h
/***************************************************************************************************/

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) */

  /* 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 */


/*******************************************************************************************/
                                              dentry.c
/*******************************************************************************************/
/* This file is used to handle the dentry cache. it implements the lru
policy for the cache and uses hash table for quick access of
the dentry objects*/

#include "fs.h"
#include "inode.h"
#include "dentry.h"
#include <string.h>

FORWARD _PROTOTYPE( void dir_rm_lru, (struct direntry *dip) );
FORWARD _PROTOTYPE( int get_dentry, (struct inode *ldir_ptr,
                                char string[NAME_MAX], ino_t *numb, int flag));

 /*===========================================================================*
  *                                   search_dentry                           *
  *===========================================================================*/
PUBLIC int search_dentry (ldir_ptr, string, numb, flag)

  register struct inode *ldir_ptr; /* parents inode entry */
  char string[NAME_MAX]; /* name of the directory */
  ino_t *numb; /* inode number */
  int flag; /* flag : LOOK_UP || DELETE || ENTER_D ; for ENTER just return !OK*/

 {
   /* This function searches for the dir entry of the corresponding
      dir or file name. It returns OK if entry is found.*/

   register struct direntry *dip;
   int b;

   if(( flag == ENTER)||(flag == DELETE) ) return(!OK);

   if( flag == DELETE_D)
   {
               get_dentry(ldir_ptr, string, numb, DELETE_D);
               return(1);
   }

   if(flag == LOOK_UP)
   {
               b = get_dentry(ldir_ptr, string, numb, flag);
               if(b == 0)
                           return(NFOUND_D); /* search is over, no entry found */
               else
               {
                           /*  printf(" F:%d",++counter1);*/
                           return(FOUND_D); /* got the entry*/
               }

   }

   if( flag == ENTER_D )
  {
            /*printf(" E:%d",++counter2);*/
       get_dentry(ldir_ptr, string, numb, ENTER_D);

  }

  return(!OK);

 }


/*=============================================================================*
 *                                   get_dentry                                *
 *=============================================================================*/
 PRIVATE int get_dentry(ldir_ptr, string, numb, flag)

 register struct inode *ldir_ptr; /* parents inode entry */
 char string[NAME_MAX]; /* name of the directory */
 ino_t *numb; /* inode number */
 int flag;
/****
if flag == LOOK_UP search for the 'string' and return indoe number in 'numb'
if flag == ENTER_D fills the dobject for future use
if flag == DELETE_D delete the dobject, make invalid
*****/

{
              int b;
      int lfl;
            register struct direntry *dip, *prev_ptr;

            lfl = 0;  /* clear flag for future use */

            /*search the hash chain for the given ldir_ptr->i_num.    */
            if(ldir_ptr->i_dev == NO_DEV)
            {
                        printf("Invalid entry");
                        return(0);
            }
            else
            {
                        b = (int) (ldir_ptr->i_num & D_HASH_MASK);
                        dip = dir_hash[b];
                        while( dip != NIL_DIRENT)
                        {
                                    if( (dip->d_dev == ldir_ptr->i_dev) && ( dip->d_pnum == ldir_ptr->i_num ) && ( strncmp( string,
dip->d_name, NAME_MAX) == 0))
                                    {
                                                lfl = 1;
                                                break; /* dobject found, break */
                                    }
                                    else
                                    {
                                                dip = dip->d_hash; /* move to next block on hash chain */
                                    }
                        }
            }

            if( (lfl == 0) && ((flag == LOOK_UP) || (flag == DELETE_D)) )
            {
                        return(0); /* dobject not found, return 0 */
            }
            if(lfl == 0)
            {
                        dip = dir_front;  /*remove the dobject fron front of the hash chain */
                        b = (int) (dip->d_pnum & D_HASH_MASK);
            }

            dir_rm_lru(dip);
            prev_ptr = dir_hash[b];
```

```
        if( prev_ptr == dip)
        {
                dir_hash[b] = dip->d_hash;
        }
        else
        {
        /* the block just taken is not on the front of its hash chain */
                while( prev_ptr->d_hash != NIL_DIRENT )
                {
                        if( prev_ptr->d_hash == dip)
                        {
                                prev_ptr->d_hash = dip->d_hash; /* found it */
                                break;
                        }
                        else
                        {
                                prev_ptr = prev_ptr->d_hash; /* keep looking */
                        }
                }
        }

        if( flag == DELETE_D)
        {
        /* put the block at the front of the cache chain */

                dip->d_dev = NO_DEV;
                dip->d_inumb = (ino_t)0;
                dip->d_pnum = (ino_t)0;
                dip->d_prev = NIL_DIRENT;
                dip->d_next = dir_front;

                if( dir_front == NIL_DIRENT)
                        dir_rear = dip;
                else
                        dir_front->d_prev = dip;

                dir_front = dip;
                return(1);
        }

        /* make entry to the freed dentry block from the hash chain, if ENTER_D*/
        if( (lfl == 0) && (flag == ENTER_D))
        {
                dip->d_dev = ldir_ptr->i_dev;
                dip->d_inumb = *numb;
                dip->d_pnum = ldir_ptr->i_num;
                strcpy(dip->d_name,string);
                b = (int) (ldir_ptr->i_num & D_HASH_MASK);
        }

        *numb = dip->d_inumb;  /* return inode value for the LOOK_UP */

        /* put block in front of hash chain */
        dip->d_hash = dir_hash[b];
        dir_hash[b] = dip;

        /* put blocks at rear end of the cache chain */
        dip->d_prev = dir_rear;
        dip->d_next = NIL_DIRENT;

        if( dir_rear == NIL_DIRENT)
                dir_front = dip;
        else
                dir_rear->d_next = dip;
        /* point rear to the ldce node */
        dir_rear = dip;

        return (1);
}
```

```
/*===========================================================================*
 *                              dir_rm_lru                                   *
 *===========================================================================*/
 PRIVATE void dir_rm_lru(dip)
 struct direntry *dip;
{
	/* removes a dentry from its LRU chain */
	struct direntry *next_ptr, *prev_ptr;

	next_ptr = dip->d_next;  /* Successor on LRU chain*/
	prev_ptr = dip->d_prev;  /* Predecessor on LRU chain*/

	if( prev_ptr != NIL_DIRENT )
		prev_ptr->d_next = next_ptr;
	else
		dir_front = next_ptr; /* block was at the front of chain */

	if( next_ptr != NIL_DIRENT )
		next_ptr->d_prev = prev_ptr;
	else
		dir_rear = prev_ptr; /* block was at rear of the chain */
}


/*******************************************************************************/
                                        inode.c
/*******************************************************************************/

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


/*===========================================================================*
 *                              get_inode                                    *
 *===========================================================================*/
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);
}


/*===========================================================================*
 *				put_inode				     *
 *===========================================================================*/
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);
  }
}

/*===========================================================================*
 *				alloc_inode				     *
 *===========================================================================*/
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 */

		wipe_inode(rip);
  }

  return(rip);
}

/*===========================================================================*
 *				wipe_inode				     *
 *===========================================================================*/
PUBLIC void wipe_inode(rip)
register struct inode *rip;	/* the inode to be erased */
{

  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 < V2_NR_TZONES; i++) rip->i_zone[i] = NO_ZONE;
}


/*===========================================================================*
 *				free_inode				     *
 *===========================================================================*/
PUBLIC void free_inode(dev, inumb)
dev_t dev;			/* on which device is the inode */
ino_t inumb;				/* number of inode to be freed */
{
/* Return an inode to the pool of unallocated inodes. */

  register struct super_block *sp;
  bit_t b;

  /* Locate the appropriate super_block. */
  sp = get_super(dev);
  if (inumb <= 0 || inumb > sp->s_ninodes) return;
  b = inumb;
  free_bit(sp, IMAP, b);
  if (b < sp->s_isearch) sp->s_isearch = b;
}

/*===========================================================================*
 *				rw_inode				     *
 *===========================================================================*/
PUBLIC void rw_inode(rip, rw_flag)
register struct inode *rip;	/* pointer to inode to be read/written */
int rw_flag;				/* READING or WRITING */
```

```
{
/* An entry in the inode table is to be copied to or from the disk. */

  register struct buf *bp;
  register struct super_block *sp;
  d1_inode *dip;
  d2_inode *dip2;
  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;

  /* 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
        new_icopy(rip, dip2, rw_flag, sp->s_native);

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

/*===========================================================================*
 *                              dup_inode                                    *
 *===========================================================================*/
PUBLIC void dup_inode(ip)
struct inode *ip;                  /* The inode to be duplicated. */
{
/* This routine is a simplified form of get_inode() for the case where
 * the inode pointer is already known.
 */

  ip->i_count++;
}

/*******************************************************************************************/
                                        link.c
/*******************************************************************************************/
/* This file handles the LINK and UNLINK system calls.  It also deals with
 * deallocating the storage used by a file when the ldce 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 "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.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])                       );


/*===========================================================================*
 *                                    do_link                                *
 *===========================================================================*/
PUBLIC int do_link()
{
/* Perform the link(name1, name2) system call. */
  char *temp;
  short flag; /* set if fdce symbolic link */
  mode_t bits;
  register struct inode *ip, *rip, *new_ip;
  register int r;
  char string[NAME_MAX];
  struct buf *bp;
  char path[PATH_MAX];

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

  /* save user_path for symbolic link */

  strncpy(path, user_path, PATH_MAX);

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

  /* 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 = ldce_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;
        }
  }

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

        {
                /* Start of (FDCE) SYMBOLIC LINK part */
                put_inode(rip);

                /* check if path is absolute or not */
                temp = &path[0];

                if(*temp != '/')
                {
                        put_inode(ip);
                        printf("\nAbsolute path required for the Source: %s\n", path);
                        return(ENOTABS);
                }

                if(name1_length <= V2_NR_CHARS)
                {
                        bits = I_FDCE_SLINK;
                        flag = 1;
                }
                else
                {
                        bits = I_SYMBOLIC_LINK;
                        flag = 0;
                }

                if ((new_ip = new_node(user_path, (mode_t)( bits | 0777),
                            (off_t)0)) == NIL_INODE)
                {
                        put_inode(new_ip);
                        put_inode(ip);
                        return(err_code);
                }

                truncate(new_ip);
                wipe_inode(new_ip);
                new_ip->i_size = name1_length;

                if (flag)
                {
                        /* stuff pathname into inode zone space */
                        memcpy(new_ip->i_data, path, name1_length);
                }
                else
                {
                    /* allocate disk block for name1 */
                        if ( (bp = new_block(new_ip, 0)) == NIL_BUF)
                        {
                                (void) unlink_file(ip, new_ip, string);
                                (new_ip->i_nlinks)--;
                                put_inode(new_ip);
                                put_inode(ip);
                                return(err_code);
                        }

                    /* stuff pathname into diskblock and set immediate writing */
                        memcpy(bp->b_data, path, name1_length);
                        bp->b_dirt = DIRTY;
                        put_block(bp, INODE_BLOCK);
                }

                new_ip->i_dirt = DIRTY;
                rw_inode(new_ip, WRITING);
                put_inode(new_ip);
                put_inode(ip);
                return(err_code);
}
```

```
/* Try to Hard 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);
}


/*===========================================================================*
 *                                do_unlink                                  *
 *===========================================================================*/
PUBLIC int do_unlink()
{

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

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

  /* The ldce 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);
}



/*===========================================================================*
 *                              truncate                                     *
 *===========================================================================*/
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;

  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);
  }

  /* Leave zone numbers for de(1) to recover file after an unlink(2).  */
}


/*===========================================================================*
 *                              remove_dir                                   *
 *===========================================================================*/
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 directory must not be anybody's root/working directory
   */

  int r;
  register struct fproc *rfp;

  /* 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);
}


/*===========================================================================*
 *                                unlink_file                                *
 *===========================================================================*/
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 fdceer 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;
  }

  put_inode(rip);
  return(r);
```

```
}


/*******************************************************************************************/
                                                    main.c
/*******************************************************************************************/
/* This file contains the main program of the File System.  It consists of
 * a loop that gets messages requesting work, carries out the work, and sends
 * replies.
 *
 * The entry points into this file are
 *   main:   main program of the File System
 *   reply:  send a reply to a process after the requested work is done
 */

struct super_block;                 /* proto.h needs to know this */

#include "fs.h"
#include <fcntl.h>
#include <string.h>
#include <sys/ioctl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/boot.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"
#include "dentry.h"

FORWARD _PROTOTYPE( void buf_pool, (void)                          );
FORWARD _PROTOTYPE( void fs_init, (void)                           );
FORWARD _PROTOTYPE( void get_boot_parameters, (void)               );
FORWARD _PROTOTYPE( void get_work, (void)                          );
FORWARD _PROTOTYPE( void load_ram, (void)                          );
FORWARD _PROTOTYPE( void load_super, (Dev_t super_dev)             );
FORWARD _PROTOTYPE( void load_dentry, (void)                       );


/*===========================================================================*
 *                                main                                       *
 *===========================================================================*/
PUBLIC void main()
{
/* This is the main program of the file system.  The main loop consists of
 * three major activities: getting new work, processing the work, and sending
 * the reply.  This loop never terminates as long as the file system runs.
 */
  int error;

  fs_init();

  /* This is the main loop that gets work, processes it, and sends replies. */
  while (TRUE) {
        get_work();                     /* sets who and fs_call */

        fp = &fproc[who];       /* pointer to proc table struct */
        super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);   /* su? */
        dont_reply = FALSE; /* in other words, do reply is default */

        /* Call the internal function that does the work. */
        if (fs_call < 0 || fs_call >= NCALLS)
                error = EBADCALL;
        else
                error = (*call_vector[fs_call])();

        /* Copy the results back to the user and send reply. */
		if (dont_reply) continue;
		reply(who, error);
		if (rdahed_inode != NIL_INODE) read_ahead(); /* do block read ahead */
  }
}


/*===========================================================================*
 *				get_work				     *
 *===========================================================================*/
PRIVATE void get_work()
{
  /* Normally wait for new input.  However, if 'reviving' is
   * nonzero, a suspended process must be awakened.
   */

  register struct fproc *rp;

  if (reviving != 0) {
		/* Revive a suspended process. */
		for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
			if (rp->fp_revived == REVIVING) {
				who = (int)(rp - fproc);
				fs_call = rp->fp_fd & BYTE;
				fd = (rp->fp_fd >>8) & BYTE;
				buffer = rp->fp_buffer;
				nbytes = rp->fp_nbytes;
				rp->fp_suspended = NOT_SUSPENDED; /*no longer hanging*/
				rp->fp_revived = NOT_REVIVING;
				reviving--;
				return;
			}
		panic("get_work couldn't revive anyone", NO_NUM);
  }

  /* Normal case.  No one to revive. */
  if (receive(ANY, &m) != OK) panic("fs receive error", NO_NUM);

  who = m.m_source;
  fs_call = m.m_type;
}


/*===========================================================================*
 *				reply					     *
 *===========================================================================*/
PUBLIC void reply(whom, result)
int whom;			/* process to reply to */
int result;			/* result of the call (usually OK or error #) */
{
/* Send a reply to a user process. It may fail (if the process has just
 * been killed by a signal), so don't check the return code.  If the send
 * fails, just ignore it.
 */

  reply_type = result;
  send(whom, &m1);
}


/*===========================================================================*
 *				fs_init					     *
 *===========================================================================*/
PRIVATE void fs_init()
{
/* Initialize global variables, tables, etc. */

  register struct inode *rip;
  int i;
  message mess;
  /* The following initializations are needed to let dev_opcl succeed .*/
  fp = (struct fproc *) NULL;
  who = FS_PROC_NR;

  buf_pool();                                   /* initialize buffer pool */
  get_boot_parameters();              /* get the parameters from the menu */
  load_ram();                                   /* init RAM disk, load if it is root */
  load_super(ROOT_DEV);                         /* load super block for root device */
  load_dentry();

  /* Initialize the 'fproc' fields for process 0 .. INIT. */
  for (i = 0; i <= LOW_USER; i+= 1) {
        if (i == FS_PROC_NR) continue;                /* do not initialize FS */
        fp = &fproc[i];
        rip = get_inode(ROOT_DEV, ROOT_INODE);
        fp->fp_rootdir = rip;
        dup_inode(rip);
        fp->fp_workdir = rip;
        fp->fp_realuid = (uid_t) SYS_UID;
        fp->fp_effuid = (uid_t) SYS_UID;
        fp->fp_realgid = (gid_t) SYS_GID;
        fp->fp_effgid = (gid_t) SYS_GID;
        fp->fp_umask = ~0;
  }

  /* Certain relations must hold for the file system to work at all. */
  if (SUPER_SIZE > BLOCK_SIZE) panic("SUPER_SIZE > BLOCK_SIZE", NO_NUM);
  if (BLOCK_SIZE % V2_INODE_SIZE != 0)          /* this checks V1_INODE_SIZE too */
        panic("BLOCK_SIZE % V2_INODE_SIZE != 0", NO_NUM);
  if (OPEN_MAX > 127) panic("OPEN_MAX > 127", NO_NUM);
  if (NR_BUFS < 6) panic("NR_BUFS < 6", NO_NUM);
  if (V1_INODE_SIZE != 32) panic("V1 inode size != 32", NO_NUM);
  if (V2_INODE_SIZE != 64) panic("V2 inode size != 64", NO_NUM);
  if (OPEN_MAX > 8 * sizeof(long)) panic("Too few bits in fp_cloexec", NO_NUM);

  /* Tell the memory task where my process table is for the sake of ps(1). */
  mess.m_type = DEV_IOCTL;
  mess.PROC_NR = FS_PROC_NR;
  mess.REQUEST = MIOCSPSINFO;
  mess.ADDRESS = (void *) fproc;
  (void) sendrec(MEM, &mess);
}


/*===========================================================================*
 *                              load_super                                   *
 *===========================================================================*/
PRIVATE void load_super(super_dev)
dev_t super_dev;                                /* place to get superblock from */
{
  int bad;
  register struct super_block *sp;
  register struct inode *rip;

  /* Initialize the super_block table. */
  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
        sp->s_dev = NO_DEV;

  /* Read in super_block for the root file system. */
  sp = &super_block[0];
  sp->s_dev = super_dev;

  /* Check super_block for consistency (is it the right diskette?). */
  bad = (read_super(sp) != OK);
  if (!bad) {
        rip = get_inode(super_dev, ROOT_INODE);       /* inode for root dir */
        if ( (rip->i_mode & I_TYPE) != I_DIRECTORY || rip->i_nlinks < 3) bad++;
  }
  if (bad)panic("Invalid root file system.  Possibly wrong diskette.",NO_NUM);
  sp->s_imount = rip;
  dup_inode(rip);
  sp->s_isup = rip;
  sp->s_rd_only = 0;
  return;
}


/*===========================================================================*
 *                              load_dentry                                  *
 *===========================================================================*/
PRIVATE void load_dentry()

{
  int i;
  register struct direntry *dp;

  dir_front = &direntry[0];
  dir_rear = &direntry[NR_DIRENT - 1];

  /* Initialize the directory table. */
  for (dp = &direntry[0]; dp < &direntry[NR_DIRENT]; dp++)
  {
        dp->d_dev = NO_DEV;
        dp->d_next = dp + 1;
        dp->d_prev = dp - 1;
        dp->d_hash = dp->d_next;
        dp->d_inumb = 0;
  }

  direntry[0].d_prev = NIL_DIRENT;
  direntry[NR_DIRENT - 1].d_next = NIL_DIRENT;
  direntry[NR_DIRENT - 1].d_hash = NIL_DIRENT;

  dir_hash[0] = dir_front;
  for( i=1; i < NR_DIR_HASH; i++)  dir_hash[i] = NIL_DIRENT;

  return;
}


/*****************************************************************************************/
                                        open.c
/*****************************************************************************************/

#include "fs.h"
#include <sys/stat.h>
#include <string.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"

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) );*/


/*===========================================================================*
```

```
 *                                              do_creat                                              *
 *===========================================================================*/
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);
}


/*===========================================================================*
 *                                              new_node                                              *
 *===========================================================================*/
PUBLIC 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 *rldce_dir_ptr, *rip;
  struct inode *old_workdir_ip;
  int slink_found;
  register int r;
  char string[NAME_MAX];
  int loops;

  old_workdir_ip = fp->fp_workdir; /* save the current working directory */
  loops = 0;

  do {
   slink_found = FALSE;

   /* See if the path can be opened down to the ldce directory. */
   if ((rldce_dir_ptr = ldce_dir(path, string)) == NIL_INODE) {
        fp->fp_workdir = old_workdir_ip; /* restore cwd */
        return(NIL_INODE);
   }

   /* The final directory is accessible. Get final component of the path. */
   rip = advance(rldce_dir_ptr, string);

   if (rip != NIL_INODE && (rip->i_mode & I_TYPE) == I_SYMBOLIC_LINK) {
        if (++loops > 8) {
                fp->fp_workdir = old_workdir_ip;
                put_inode(rldce_dir_ptr);
                err_code = ELOOP;
                return(NIL_INODE);
        }
        rip = slink_traverse(rip, path, string, rldce_dir_ptr);
        slink_found = TRUE;
        put_inode(rldce_dir_ptr);
        fp->fp_workdir = rip; /* cd to symlink target dir */
   }
  } while (slink_found);

  if ( rip == NIL_INODE && err_code == ENOENT) {
        /* Ldce path component does not exist.  Make new directory entry. */
        if ( (rip = alloc_inode(rldce_dir_ptr->i_dev, bits)) == NIL_INODE) {
                /* Can't creat new inode: out of inodes. */
                put_inode(rldce_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(rldce_dir_ptr, string, &rip->i_num,ENTER)) != OK) {
                        put_inode(rldce_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 ldce 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(rldce_dir_ptr);
 err_code = r;
 return(rip);
}


/*===========================================================================*
 *                              do_mknod                                     *
 *===========================================================================*/
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);
}


/*===========================================================================*
 *                              do_mkdir                                     *
 *===========================================================================*/
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];         /* ldce 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 = ldce_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);
  }

  /* 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' */
}


/*===========================================================================*
 *				do_close					     *
 *===========================================================================*/
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
                                 * ldce 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);
}



/*********************************************************************************************/
                                                        path.c
/*********************************************************************************************/
/* This file contains the procedures that look up path names in the directory
 * system and determine the inode number that goes with a given path name.
 *
 *  The entry points into this file are
 *   eat_path:          the 'main' routine of the path-to-inode conversion mechanism
 *   ldce_dir:          find the final directory on a given path
 *   advance:           parse one component of a path name
 *   search_dir: search a directory for a string and return its inode number
 */
```

```
#include "fs.h"
#include <string.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

PUBLIC char dot1[2] = ".";          /* used for search_dir to bypass the access */
PUBLIC char dot2[3] = "..";         /* permissions for . and ..                     */

FORWARD _PROTOTYPE( char *get_name, (char *old_name, char string [NAME_MAX]) );

/*===========================================================================*
 *                                eat_path                                   *
 *===========================================================================*/
PUBLIC struct inode *eat_path(path)
char *path;                         /* the path name to be parsed */
{
/* Parse the path 'path' and put its inode in the inode table. If not possible,
 * return NIL_INODE as function value and an error code in 'err_code'.
 */

  register struct inode *ldip, *rip, *old_workdir_ip;
  char string[NAME_MAX];            /* hold 1 path component name here */
  int slink_found;
  int loops;

  old_workdir_ip = fp->fp_workdir; /* save the current working directory */
  loops = 0;

  do {
   slink_found = FALSE;

   /* First open the path down to the final directory. */
   if ( (ldip = ldce_dir(path, string)) == NIL_INODE) {
		fp->fp_workdir = old_workdir_ip;
		return(NIL_INODE);  /* we couldn't open final directory */
   }

   /* The path consisting only of "/" is a special case, check for it. */
   if (string[0] == '\0') return(ldip);

   /* Get final component of the path. */
   rip = advance(ldip, string);

   if (rip != NIL_INODE && (rip->i_mode & I_TYPE) == I_SYMBOLIC_LINK) {
		if (++loops > 8) {
			put_inode(rip);
			put_inode(ldip);
			fp->fp_workdir = old_workdir_ip;
			err_code = ELOOP;
			return(NIL_INODE);
		}
		rip = slink_traverse(rip, path, string, ldip);
		slink_found = TRUE;
		fp->fp_workdir = rip; /* cd to link's starting dir */
		put_inode(rip);
   }
   put_inode(ldip);
  } while (slink_found);
  fp->fp_workdir = old_workdir_ip;
  return(rip);
}


/*===========================================================================*
 *                                ldce_dir                                   *
 *===========================================================================*/
PUBLIC struct inode *ldce_dir(path, string)
char *path;                         /* the path name to be parsed */
char string[NAME_MAX];              /* the final component is returned here */
{
/* Given a path, 'path', located in the fs address space, parse it as
 * far as the ldce directory, fetch the inode for the ldce directory into
 * the inode table, and return a pointer to the inode.  In
 * addition, return the final component of the path in 'string'.
 * If the ldce directory can't be opened, return NIL_INODE and
 * the reason for failure in 'err_code'.
 */

  register struct inode *rip;
  register char *new_name;
  register struct inode *new_ip;
  int loops;

  loops = 0; /* count symlink traversals */

  /* Is the path absolute or relative?  Initialize 'rip' accordingly. */
  rip = (*path == '/' ? fp->fp_rootdir : fp->fp_workdir);

  /* If dir has been removed or path is empty, return ENOENT. */
  if (rip->i_nlinks == 0 || *path == '\0') {
	err_code = ENOENT;
	return(NIL_INODE);
  }

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

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

	/* There is more path.  Keep parsing. */
	new_ip = advance(rip, string);
	if (new_ip == NIL_INODE) {
		put_inode(rip);
		return(NIL_INODE);
	}

	/* The call to advance() succeeded.  Fetch next component. */

	if ((new_ip->i_mode & I_TYPE) == I_SYMBOLIC_LINK) {
		++loops;
		if (loops > 8) {
			err_code = ELOOP;
			put_inode(rip);
			put_inode(new_ip);
			return(NIL_INODE);
		}
		new_ip = slink_traverse(new_ip, path, string, rip);
	} else
		path = new_name;

	put_inode(rip);             /* rip either obsolete or irrelevant */
```

```
            rip = new_ip;
  }
}


/*===========================================================================*
 *                                get_name                                   *
 *===========================================================================*/
PRIVATE char *get_name(old_name, string)
char *old_name;                         /* path name to parse */
char string[NAME_MAX];                  /* component extracted from 'old_name' */
{
/* Given a pointer to a path name in fs space, 'old_name', copy the next
 * component to 'string' and pad with zeros.  A pointer to that part of
 * the name as yet unparsed is returned.  Roughly speaking,
 * 'get_name' = 'old_name' - 'string'.
 *
 * This routine follows the standard convention that /usr/dce, /usr//dce,
 * //usr///dce and /usr/dce/ are all equivalent.
 */

  register int c;
  register char *np, *rnp;

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

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

  /* To make /usr/dce/ equivalent to /usr/dce, skip trailing slashes. */
  while (c == '/' && rnp < &old_name[PATH_MAX]) c = *++rnp;

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

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


/*===========================================================================*
 *                                advance                                    *
 *===========================================================================*/
PUBLIC struct inode *advance(dirp, string)
struct inode *dirp;             /* inode for directory to be searched */
char string[NAME_MAX];                  /* component name to look for */
{
/* Given a directory and a component of a path, look up the component in
 * the directory, find the inode, open it, and return a pointer to its inode
 * slot.  If it can't be done, return NIL_INODE.
 */

  register struct inode *rip;
  struct inode *rip2;
  register struct super_block *sp;
  int r, inumb;
  dev_t mnt_dev;
  ino_t numb;

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

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

  /* If 'string' is not present in the directory, signal error. */
  if ( (r = search_dir(dirp, string, &numb, LOOK_UP)) != OK) {
	err_code = r;
	return(NIL_INODE);
  }

  /* Don't go beyond the current root directory, unless the string is dot2. */
  if (dirp == fp->fp_rootdir && strcmp(string, "..") == 0 && string != dot2)
		return(get_inode(dirp->i_dev, (int) dirp->i_num));

  /* The component has been found in the directory.  Get inode. */
  if ( (rip = get_inode(dirp->i_dev, (int) numb)) == NIL_INODE)
	return(NIL_INODE);

  if (rip->i_num == ROOT_INODE)
	if (dirp->i_num == ROOT_INODE) {
	    if (string[1] == '.') {
		for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++){
			if (sp->s_dev == rip->i_dev) {
				/* Release the root inode.  Replace by the
				 * inode mounted on.
				 */
				put_inode(rip);
				mnt_dev = sp->s_imount->i_dev;
				inumb = (int) sp->s_imount->i_num;
				rip2 = get_inode(mnt_dev, inumb);
				rip = advance(rip2, string);
				put_inode(rip2);
				break;
			}
		}
	    }
	}
  if (rip == NIL_INODE) return(NIL_INODE);

  /* See if the inode is mounted on.  If so, switch to root directory of the
   * mounted file system.  The super_block provides the linkage between the
   * inode mounted on and the root directory of the mounted file system.
   */
  while (rip != NIL_INODE && rip->i_mount == I_MOUNT) {
	/* The inode is indeed mounted on. */
	for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
		if (sp->s_imount == rip) {
			/* Release the inode mounted on.  Replace by the
			 * inode of the root inode of the mounted device.
			 */
			put_inode(rip);
			rip = get_inode(sp->s_dev, ROOT_INODE);
			break;
		}
	}
  }
  return(rip);			/* return pointer to inode's component */
}


/*===========================================================================*
 *				search_dir				     *
 *===========================================================================*/
PUBLIC int search_dir(ldir_ptr, string, numb, flag)
register struct inode *ldir_ptr;	/* ptr to inode for dir to search */
char string[NAME_MAX];		/* component to search for */
ino_t *numb;			/* pointer to inode number */
int flag;			/* LOOK_UP, ENTER, DELETE or IS_EMPTY */
{
/* This function searches the directory whose inode is pointed to by 'ldip':
 * if (flag == ENTER)  enter 'string' in the directory with inode # '*numb';
 * if (flag == DELETE) delete 'string' from the directory;
* if (flag == LOOK_UP) search for 'string' and return inode # in 'numb';
* if (flag == IS_EMPTY) return OK if only . and .. in dir else ENOTEMPTY;
*
*    if 'string' is dot1 or dot2, no access permissions are checked.
*/

 register struct direct *dp;
 register struct buf *bp;
 int i, r, e_hit, t, match;
 mode_t bits;
 off_t pos;
 unsigned new_slots, old_slots;
 block_t b;
 struct super_block *sp;
 int extended = 0;

 int val; /* value returned by search_dentry */

/* If 'ldir_ptr' is not a pointer to a dir inode, error. */
 if ( (ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) return(ENOTDIR);

 r = OK;

 if (flag != IS_EMPTY) {
         bits = (flag == LOOK_UP ? X_BIT : W_BIT | X_BIT);

         if (string == dot1 || string == dot2) {
                 if (flag != LOOK_UP) r = read_only(ldir_ptr);
                                         /* only a writable device is required. */
    }
         else r = forbidden(ldir_ptr, bits); /* check access permissions */
 }
 if (r != OK) return(r);


 if(search_dentry(ldir_ptr, string, numb, flag) == FOUND_D)
 {
         return(OK);
 }

 /* Step through the directory one block at a time. */
 old_slots = (unsigned) (ldir_ptr->i_size/DIR_ENTRY_SIZE);
 new_slots = 0;
 e_hit = FALSE;
 match = 0;                              /* set when a string match occurs */

 for (pos = 0; pos < ldir_ptr->i_size; pos += BLOCK_SIZE) {
         b = read_map(ldir_ptr, pos);    /* get block number */

         /* Since directories don't have holes, 'b' cannot be NO_BLOCK. */
         bp = get_block(ldir_ptr->i_dev, b, NORMAL);        /* get a dir block */

         /* Search a directory block. */
         for (dp = &bp->b_dir[0]; dp < &bp->b_dir[NR_DIR_ENTRIES]; dp++) {
                 if (++new_slots > old_slots) { /* not found, but room left */
                         if (flag == ENTER) e_hit = TRUE;
                         break;
                 }

                 /* Match occurs if string found. */
                 if (flag != ENTER && dp->d_ino != 0) {
                         if (flag == IS_EMPTY) {
                                 /* If this test succeeds, dir is not empty. */
                                 if (strcmp(dp->d_name, "." ) != 0 &&
                                     strcmp(dp->d_name, "..") != 0) match = 1;
                         } else {
                                 if (strncmp(dp->d_name, string, NAME_MAX) == 0)
                                         match = 1;
                         }
                 }
```

```
                        if (match) {
                                /* LOOK_UP or DELETE found what it wanted. */
                                r = OK;
                                if (flag == IS_EMPTY) r = ENOTEMPTY;
                                else if (flag == DELETE) {
                                        /* Save d_ino for recovery. */

                        /*      delete entry from dentry table */
                                        search_dentry( ldir_ptr, string,
                                                        (ino_t *)0, DELETE_D);
                                        t = NAME_MAX - sizeof(ino_t);
                                        *((ino_t *) &dp->d_name[t]) = dp->d_ino;
                                        dp->d_ino = 0;          /* erase entry */
                                        bp->b_dirt = DIRTY;
                                        ldir_ptr->i_update |= CTIME | MTIME;
                                        ldir_ptr->i_dirt = DIRTY;
                                } else {
                                        sp = ldir_ptr->i_sp;    /* 'flag' is LOOK_UP */
                                        *numb = conv2(sp->s_native, (int) dp->d_ino);
                                        /* search complete, make dobject */
                                        search_dentry( ldir_ptr, string, numb, ENTER_D);
                                }
                                put_block(bp, DIRECTORY_BLOCK);
                                return(r);
                        }


                        /* Check for free slot for the benefit of ENTER. */
                        if (flag == ENTER && dp->d_ino == 0) {
                                e_hit = TRUE;           /* we found a free slot */
                                break;
                        }
                }

                /* The whole block has been searched or ENTER has a free slot. */
                if (e_hit) break;       /* e_hit set if ENTER can be performed now */
                put_block(bp, DIRECTORY_BLOCK);         /* otherwise, continue searching dir */
        }

        /* The whole directory has now been searched. */
        if (flag != ENTER) return(flag == IS_EMPTY ? OK : ENOENT);

        /* This call is for ENTER.  If no free slot has been found so far, try to
         * extend directory.
         */
        if (e_hit == FALSE) { /* directory is full and no room left in ldce block */
                new_slots++;                    /* increase directory size by 1 entry */
                if (new_slots == 0) return(EFBIG); /* dir size limited by slot count */
                if ( (bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
                        return(err_code);
                dp = &bp->b_dir[0];
                extended = 1;
        }

        /* 'bp' now points to a directory block with space. 'dp' points to slot. */
        (void) memset(dp->d_name, 0, (size_t) NAME_MAX); /* clear entry */
        for (i = 0; string[i] && i < NAME_MAX; i++) dp->d_name[i] = string[i];
        sp = ldir_ptr->i_sp;
        dp->d_ino = conv2(sp->s_native, (int) *numb);
        bp->b_dirt = DIRTY;
        put_block(bp, DIRECTORY_BLOCK);
        ldir_ptr->i_update |= CTIME | MTIME;    /* mark mtime for update later */
        ldir_ptr->i_dirt = DIRTY;
        if (new_slots > old_slots) {
                ldir_ptr->i_size = (off_t) new_slots * DIR_ENTRY_SIZE;
                /* Send the change to disk if the directory is extended. */
                if (extended) rw_inode(ldir_ptr, WRITING);
        }
        return(OK);
```

```
}

/*===========================================================================*
 *                              slink_traverse                               *
 *===========================================================================*/

/* copy path out of symlink's disk block -- return inode pointer to
   the directory that the path infers */

PUBLIC struct inode *slink_traverse(rip, path, string, ldip)
register struct inode *rip;
char *path;
char *string;
register struct inode *ldip;
{
  register char *p, *q, *old_path;
  char temp[PATH_MAX];
  struct buf *bp;
  block_t b;

  b = read_map(rip, 0);
  bp = get_block(rip->i_dev, b, NORMAL); /* get the pathname block */
  memcpy(temp, bp->b_data, rip->i_size);
  temp[rip->i_size] = '\0';
  put_block(bp, NORMAL);

  q = string;
  p = path;

/* Find where string ends in path.  String can only start after a / in path */

  while (*q) {
        while (*p == '/')               /* skip leading /'s */
                p++;
        while (*q == '/')               /* skip leading /'s */
                q++;

        if (*p != *q) {
                while (*p != '\0') {
                        ++p;
                        if (*p == '/')
                                break;
                }
                if (*p == '\0')
                        break;
                else
                        continue;
        }

        old_path = p;                   /* remember where you found it */

        while (*q) {
                p++, q++;  /* look at the next chars */
                if (*q == '\0')
                        break;

                if (*p != *q) { /* restart if all of string not in path */
                        p = old_path;
                        while (*p != '\0') {
                                ++p;
                                if (*p == '/')
                                        break;
                        }
                        if (*p == '\0')
                                break;
                        q = string;
                }
        }
 }
```

```
  q = temp;
  while (*q) ++q;                       /* find the end of temp */

  while (*p) *q++ = *p++;               /* add the path to temp */
  *q = '\0';                            /* guarantee that temp is null terminated */

  q = temp;
  p = path;
  while (*q) *p++ = *q++;               /* copy temp to path */
  *p = '\0';                            /* guarantee that path is null terminated */

  put_inode(rip);

  p = path;

  if (*p == '/') {
                rip = fp->fp_rootdir;
  } else {
                rip = ldip;
  }

  dup_inode(rip);
  return (rip);
}


/*********************************************************************************************/
                                        search_dir.c
/*********************************************************************************************/
/* This file contains the procedures that look up path names in the directory
 * system and determine the inode number that goes with a given path name.
 *
 *   search_dir: search a directory for a string and return its inode number
 */

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


/*===========================================================================*
 *                              search_dir                                   *
 *===========================================================================*/
PUBLIC int search_dir(ldir_ptr, string, numb, flag)
register struct inode *ldir_ptr;      /* ptr to inode for dir to search */
char string[V3_NAME_MAX];                     /* component to search for */
ino_t *numb;                                  /* pointer to inode number */
int flag;                             /* LOOK_UP, ENTER, DELETE or IS_EMPTY */
{
/* This function searches the directory whose inode is pointed to by 'ldip':
 * if (flag == ENTER)  enter 'string' in the directory with inode # '*numb';
 * if (flag == DELETE) delete 'string' from the directory;
 * if (flag == LOOK_UP) search for 'string' and return inode # in 'numb';
 * if (flag == IS_EMPTY) return OK if only . and .. in dir else ENOTEMPTY;
 *
 *    if 'string' is dot1 or dot2, no access permissions are checked.
 */
  register struct direct3 *ndp;
  register struct direct *dp;
  register struct buf *bp;
  int i, r, e_hit, t, match;
  mode_t bits;
  off_t pos;
  unsigned new_slots, old_slots;
  block_t b;
  struct super_block *sp;
  int extended = 0;
  unsigned short  m, n, a, d, len, totlen;
  off_t size;
  struct direct4 *dp4, *pdp4;

 /* If 'ldir_ptr' is not a pointer to a dir inode, error. */
  if ( (ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) return(ENOTDIR);

  r = OK;

  if (flag != IS_EMPTY) {
          bits = (flag == LOOK_UP ? X_BIT : W_BIT | X_BIT);

          if (string == dot1 || string == dot2) {
                  if (flag != LOOK_UP) r = read_only(ldir_ptr);
                                          /* only a writable device is required. */
     }
          else r = forbidden(ldir_ptr, bits); /* check access permissions */
  }
  if (r != OK) return(r);


  if(ENABLE_DIR_CACHE)
          if(search_dentry(ldir_ptr, string, numb, flag) == FOUND_D)
          {
                  return(OK);
          }

  new_slots = 0;
  e_hit = FALSE;
  match = 0;                              /* set when a string match occurs */

  if ( ldir_ptr-> i_sp-> s_version == V3)
  {
          if (flag == ENTER)
          {
                  size = 0;
                  len = strlen(string);
                  m = MIN( len, V3_NAME_MAX);
                  n = GPFY(len);
                  totlen = OFFNAME + n;

                  b = read_map(ldir_ptr, size);

                  if( b == NO_BLOCK)
                  {
                          if ( (bp = new_block( ldir_ptr,
                                                          ldir_ptr->i_size)) == NIL_BUF)
                          {
                                  return(err_code);
                          }
                  }
                  else
                  {
                          bp = get_block(ldir_ptr->i_dev, b, NORMAL);
                  }

                  dp4 = (struct direct4 *) &bp->b_vardirsz[0];

                  while(1)
                  {
                          if ((dp4->d_ino == 0) && (dp4->d_rclen >= totlen))
                          { /* space for the entry found on the block */
                                  if (dp4->d_rclen >= (n + (2 * OFFNAME) + 4))
                                  {    /* if space is very enough then make it
                                            * useful for other entry */
                                          pdp4 = NEXTPTR_DIR(dp4, totlen);
                                          pdp4->d_rclen = dp4->d_rclen - totlen;
                                          pdp4->d_nmlen = 0;
```

```
                pdp4->d_ino = 0;
                dp4->d_rclen = totlen;
        }

        (void) memset( dp4->d_name, 0, (size_t) n);
        for (i=0; string[i] && i<n; i++)
                dp4->d_name[i] = string[i];
        sp = ldir_ptr->i_sp;
        dp4->d_nmlen = m;
        dp4->d_ino = conv2( sp->s_native, (int) *numb);
        bp->b_dirt = DIRTY;
        put_block( bp, DIRECTORY_BLOCK);
        ldir_ptr->i_update |= CTIME | MTIME;
        ldir_ptr->i_dirt = DIRTY;
        return (OK);
}
else
{/* inode value is not 0 or not enough space in the block*/
        if( dp4->d_rclen == 0)
        {    /* end of the directory entries encountered */

                dp4->d_rclen = OFFNAME + GPFY( dp4->d_nmlen);

                if (dp4->d_nmlen == 0)
                { /* start of a new block*/
                        dp4->d_rclen = 0;
                }
                size += dp4->d_rclen;

                if ( totlen <= OFFSET(size) )
                { /* enough space in the same block */
                        dp4 = NEXTPTR_DIR(dp4, dp4->d_rclen);
                        (void) memset( dp4->d_name, 0, (size_t) n);
                        for (i=0; string[i] && i<n; i++)
                                dp4->d_name[i] = string[i];
                        sp = ldir_ptr->i_sp;
                        dp4->d_nmlen = m;
                        dp4->d_rclen = 0;
                        /* this is the ldce entry */
                        dp4->d_ino = conv2( sp->s_native, (int) *numb);
                        bp->b_dirt = DIRTY;
                        put_block( bp, DIRECTORY_BLOCK);
                        ldir_ptr->i_update |= CTIME | MTIME;
                        /* its need size updation too */
                        ldir_ptr->i_size = size + totlen;
                        ldir_ptr->i_dirt = DIRTY;
                        return (OK);
                }
                else
                { /* Not enough space in the current block */
                        if ( (OFFNAME + 4) > OFFSET(size))
                        { /*space left is, really ver low */
                                dp4->d_rclen += OFFSET(size);
                        }
                        else
                        { /*space is low but not very low. it can
                           *  store other entries */
                                dp4 = NEXTPTR_DIR(dp4, dp4->d_rclen);
                                dp4->d_nmlen = 1;
                                dp4->d_rclen = OFFSET(size);
                                dp4->d_ino = 0;
                        }

                        bp->b_dirt = DIRTY;
                        put_block( bp, DIRECTORY_BLOCK);
                        ldir_ptr->i_dirt = DIRTY;
                        ldir_ptr->i_update |= CTIME | MTIME;
                        /* this size increment will be same for both case */
                        ldir_ptr->i_size = size + OFFSET(size);
```

```
                                                if ( (bp = new_block( ldir_ptr,
                                                                ldir_ptr->i_size)) == NIL_BUF)
                                                {       return(err_code);}

                                                dp4 = (struct direct4 *) &bp->b_vardirsz[0];

                                                (void) memset(dp4->d_name, 0, (size_t) n);
                                                for (i=0; string[i] && i<n; i++)
                                                        dp4->d_name[i] = string[i];
                                                sp = ldir_ptr->i_sp;
                                                dp4->d_nmlen = m;
                                                dp4->d_rclen = 0;
                                                /* this is the first and ldce entry of the block */
                                                dp4->d_ino = conv2( sp->s_native, (int) *numb);
                                                bp->b_dirt = DIRTY;
                                                put_block( bp, DIRECTORY_BLOCK);
                                                ldir_ptr->i_update |= CTIME | MTIME;
                                                /* its need size updation too */
                                                ldir_ptr->i_size = size + totlen;
                                                ldir_ptr->i_dirt = DIRTY;
                                                /* write the inode */
                                                rw_inode( ldir_ptr, WRITING);
                                                return (OK);
                                        }/* end of else part*/
                                }/* end of if part "if( dp->d_rclen == 0)"*/
                                else
                                {/* go to next directory entry */
                                        a = size / BLOCK_SIZE;
                                        size += dp4->d_rclen;
                                        d = size / BLOCK_SIZE;
                                        if (d > a)
                                        { /* switch to next block */
                                                put_block (bp, DIRECTORY_BLOCK);
                                                b = read_map( ldir_ptr, size);
                                                bp = get_block( ldir_ptr->i_dev, b, NORMAL);
                                                dp4 = (struct direct4 *) &bp->b_vardirsz[0];
                                        }/* end of switch to next block */
                                        else
                                        {
                                                dp4 = NEXTPTR_DIR(dp4, dp4->d_rclen);
                                        }
                                }/*end of "go to next directory entry" */
                        }/*end of "indoe value is not 0 or not enough space" */
                }/* end of while loop */
        }
        else
        {
                size = 0;
                len = strlen(string);
                n = GPFY(len);

                b = read_map(ldir_ptr, size);

                if( b == NO_BLOCK ) return(ENOENT);

                bp = get_block(ldir_ptr->i_dev, b, NORMAL);
                dp4 = (struct direct4 *) &bp->b_vardirsz[0];

                while(1)
                {
                        if (dp4->d_rclen != 0)
                        {
                                m = MAX(dp4->d_nmlen, len);
                                if( (flag == IS_EMPTY) && (dp4->d_ino != 0))
                                {
                                        if( strcmp(dp4->d_name, ".") != 0 &&
                                           strcmp(dp4->d_name, "..") != 0 )
                                                put_block(bp, DIRECTORY_BLOCK);
                                                return( ENOTEMPTY );
                                }
                if( (flag != IS_EMPTY) && (dp4->d_ino != 0) &&
                        (strncmp (dp4->d_name, string, m) == 0))
                {
                        if (flag == DELETE)
                        {
                                dp4->d_ino = 0;
                                bp->b_dirt = DIRTY;
                                ldir_ptr->i_update |= CTIME | MTIME;
                                ldir_ptr->i_dirt = DIRTY;
                        }
                        else
                        {
                                *numb = conv2(ldir_ptr->i_sp->s_native,
                                                (int)dp4->d_ino);
                        }
                        put_block(bp, DIRECTORY_BLOCK);
                        return(OK);
                }/* end if part*/
                else
                {
                        a =(unsigned int) (size / BLOCK_SIZE);
                        size = size + dp4->d_rclen;
                        d =(unsigned int) (size / BLOCK_SIZE);

                        if (d > a)
                        {
                                put_block(bp, DIRECTORY_BLOCK);
                                b = read_map(ldir_ptr, size);
                                if( b == NO_BLOCK)
                                {
                                        printf("sorry position not found\n");
                                        return(ENOENT);
                                }
                                bp = get_block(ldir_ptr->i_dev, b, NORMAL);
                                dp4 = (struct direct4 *) &bp->b_vardirsz[0];
                        }/*end of if part*/
                        else
                        {
                                dp4 = NEXTPTR_DIR(dp4, dp4->d_rclen);
                        }/*end of else part*/
                }/*end of else part*/
        }/*end of if part*/
        else
        {    /* end of directory entry encountered */
                m = MAX(dp4->d_nmlen, len);
                /* check ldce entry */

                if((flag == IS_EMPTY) && (dp4->d_ino != 0))
                {
                        if( strcmp(dp4->d_name, ".") != 0 &&
                           strcmp(dp4->d_name, "..") != 0 )
                                return( ENOTEMPTY );
                }

                if ( (flag != IS_EMPTY) && (dp4->d_ino != 0)
                        && (strncmp(dp4->d_name, string, m) == 0))
                {
                        if (flag == DELETE)
                        {
                                dp4->d_ino = 0;
                                bp->b_dirt = DIRTY;
                                ldir_ptr->i_update |= CTIME | MTIME;
                                ldir_ptr->i_dirt = DIRTY;
                        }
                        else
                        {
                                *numb = conv2(ldir_ptr->i_sp->s_native,
                                                (int)dp4->d_ino);
                        }
```

```
                                        r = OK;
                                }/*end of if part*/
                                else
                                {
                                        r= (flag == IS_EMPTY ? OK : ENOENT );
                                                                /* not found*/
                                }/*end of else part*/

                                put_block(bp, DIRECTORY_BLOCK);
                                return(r);
                        }/*end of else part*/
                }/*end of while */
        }

}
else
{
        /* Step through the directory one block at a time. */
         old_slots = (unsigned) (ldir_ptr->i_size/V2_DIR_ENTRY_SIZE);

         for (pos = 0; pos < ldir_ptr->i_size; pos += BLOCK_SIZE) {
                b = read_map(ldir_ptr, pos);        /* get block number */

                /* Since directories don't have holes, 'b' cannot be NO_BLOCK. */
                bp = get_block(ldir_ptr->i_dev, b, NORMAL);             /* get a dir block */

                /* Search a directory block. */
                for (dp = &bp->b_v2_dir[0]; dp < &bp->b_v2_dir[V2_NR_DIR_ENTRIES]; dp++) {
                        if (++new_slots > old_slots) { /* not found, but room left */
                                if (flag == ENTER) e_hit = TRUE;
                                break;
                        }

                        /* Match occurs if string found. */
                        if (flag != ENTER && dp->d_ino != 0) {
                                if (flag == IS_EMPTY) {
                                        /* If this test succeeds, dir is not empty. */
                                        if (strcmp(dp->d_name, "." ) != 0 &&
                                            strcmp(dp->d_name, "..") != 0) match = 1;
                                } else {
                                        if (strncmp(dp->d_name, string,V2_NAME_MAX) == 0)
                                                match = 1;
                                }
                        }

                        if (match) {
                                /* LOOK_UP or DELETE found what it wanted. */
                                r = OK;
                                if (flag == IS_EMPTY) r = ENOTEMPTY;
                                else if (flag == DELETE) {
                                        /* Save d_ino for recovery. */
                        /*      delete entry from dentry table */
                                        if(ENABLE_DIR_CACHE)
                                        {
                                                search_dentry( ldir_ptr, string,
                                                        (ino_t *)0, DELETE_D);
                                        }
                                        t = V2_NAME_MAX - sizeof(ino_t);
                                        *((ino_t *) &dp->d_name[t]) = dp->d_ino;
                                        dp->d_ino = 0;          /* erase entry */
                                        bp->b_dirt = DIRTY;
                                        ldir_ptr->i_update |= CTIME | MTIME;
                                        ldir_ptr->i_dirt = DIRTY;
                                } else {
                                        sp = ldir_ptr->i_sp;    /* 'flag' is LOOK_UP */
                                        *numb = conv2(sp->s_native, (int) dp->d_ino);
                                        /* search complete, make dobject */
                                        if(ENABLE_DIR_CACHE)
                                        {
                                                search_dentry( ldir_ptr,
                                                                string, numb, ENTER_D);
                                                }
                                        }
                                        put_block(bp, DIRECTORY_BLOCK);
                                        return(r);
                                }


                                /* Check for free slot for the benefit of ENTER. */
                                if (flag == ENTER && dp->d_ino == 0) {
                                        e_hit = TRUE;           /* we found a free slot */
                                        break;
                                }
                        }

                        /* The whole block has been searched or ENTER has a free slot. */
                        if (e_hit) break;          /* e_hit set if ENTER can be performed now */
                        put_block(bp, DIRECTORY_BLOCK);        /* otherwise, continue searching dir */
                }

                /* The whole directory has now been searched. */
                if (flag != ENTER) return(flag == IS_EMPTY ? OK : ENOENT);

                /* This call is for ENTER.  If no free slot has been found so far, try to
                 * extend directory.
                 */
                if (e_hit == FALSE) { /* directory is full and no room left in ldce block */
                        new_slots++;                            /* increase directory size by 1 entry */
                        if (new_slots == 0) return(EFBIG); /* dir size limited by slot count */
                        if ( (bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
                                return(err_code);
                        dp = &bp->b_v2_dir[0];
                        extended = 1;
                }

  /* 'bp' now points to a directory block with space. 'dp' points to slot. */
                (void) memset(dp->d_name, 0, (size_t) V2_NAME_MAX); /* clear entry */
                for (i = 0; string[i] && i < V2_NAME_MAX; i++) dp->d_name[i] = string[i];
                sp = ldir_ptr->i_sp;
                dp->d_ino = conv2(sp->s_native, (int) *numb);
                bp->b_dirt = DIRTY;
                put_block(bp, DIRECTORY_BLOCK);
                ldir_ptr->i_update |= CTIME | MTIME;      /* mark mtime for update later */
                ldir_ptr->i_dirt = DIRTY;
                if (new_slots > old_slots) {
                        ldir_ptr->i_size = (off_t) new_slots * V2_DIR_ENTRY_SIZE;
                        /* Send the change to disk if the directory is extended. */
                        if (extended) rw_inode(ldir_ptr, WRITING);
                }
}/* end of version else part */
  return(OK);
}


/*********************************************************************************************/
                                                        super.c
/*********************************************************************************************/

#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)

/*===========================================================================*
 *                                      alloc_bit                               *
 *===========================================================================*/
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;     /* ldce block, wrap around */
        word = 0;
  } while (--bcount > 0);
  return(NO_BIT);                         /* no bit could be allocated */
}


/*===========================================================================*
 *                                    free_bit                               *
 *===========================================================================*/
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);
}


/*===========================================================================*
 *                                    get_super                              *
 *===========================================================================*/
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 */
}


/*===========================================================================*
```

```
 *                                      mounted                                         *
 *===========================================================================*/
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);
}


/*===========================================================================*
 *                                      read_super                              *
 *===========================================================================*/
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);
  } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
        version = V2;
        native  = (magic == SUPER_V2);
  }

    else if (magic == SUPER_V3 || magic == conv2(BYTE_SWAP, SUPER_V3)) {
        version = V3;
        native  = (magic == SUPER_V3);
  }
    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);
  /* In V1, the device size was kept in a short, s_nzones, which limited
   * devices to 32K zones.  For V2, it was decided to keep the size as a
   * long.  However, just changing s_nzones to a long would not work, since
   * then the position of s_magic in the super block would not be the same
   * in V1 and V2 file systems, and there would be no way to tell whether
   * a newly mounted file system was V1 or V2.  The solution was to introduce
   * a new variable, s_zones, and copy the size there.
   *
   * Calculate some other numbers that depend on the version here too, to
   * hide some of the differences.
   */
  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 {
        sp->s_inodes_per_block = V2_INODES_PER_BLOCK;
        sp->s_ndzones = V2_NR_DZONES;
        sp->s_nindirs = V2_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;
  fsys_ver=version;  /*store in global variable, the version number */
  /* 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);
}




/**************************************************************************************************/
                                        write.c
/**************************************************************************************************/

/*===========================================================================*
  *                              new_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 ldce 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);
}


/*===========================================================================*
 *                              zero_block                                   *
 *===========================================================================*/
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;
}

/*******************************************************************************************/
                                        lirs.c
/*******************************************************************************************/

void lirs_init()
{
        int i;
        struct lirs_q *tmp;
        lirs_front = &lirs[0];

        for(tmp= &lirs[0]; tmp<&lirs[NR_BUFS]; tmp++)
        {
                tmp->block = NO_BLOCK;
                tmp->dev = NO_DEV;
                tmp->flag = NOT_IN_USE;
                tmp->next = tmp+1;
                tmp->prev = tmp-1;
        }
        lirs[NR_BUFS -1].next = &lirs[0];
        lirs[0].prev =&lirs[NR_BUFS-1];

        lirs_in_use = 0;
        lir_in_use = 0;
        lirs_rear = lirs_front;

}
void hir_init()
{
        int i;
        struct hir_q *tmp;
        hir_front = &hir[0];

        for(tmp= &hir[0]; tmp<&hir[NR_HIR]; tmp++)
        {
                tmp->block = NO_BLOCK;
                tmp->dev = NO_DEV;
                tmp->flag = NOT_IN_USE;
                tmp->next = tmp+1;
                tmp->prev = tmp-1;
        }
        hir[NR_HIR -1].next = &hir[0];
        hir[0].prev =&hir[NR_HIR-1];
        hir_in_use = 0;
        hir_rear = hir_front;
}

void put_end_lirs(*tmp)
struct lirs_q *tmp;
{
        struct lirs_q *prev_ptr, *next_ptr, *ltmp;
        short t_flag =FOUND;
        /* remove the block from its location*/

        next_ptr = tmp->next;
        prev_ptr = tmp->prev;
        prev_ptr->next = next_ptr;
        next_ptr->prev = prev_ptr;
        /* find the last element on lirs_q*/

        ltmp = lirs_front;
        while(ltmp->flag!= NOT_IN_USE)
        {
                ltmp = ltmp->next;
                if(ltmp == lirs_rear)
                {
                        t_flag = NOT_FOUND;
                        printf("not found in lirs");
                        break;
                }
        }
        if((t_flag == FOUND) || ((t_flag == NOT_FOUND) && (tmp->flag == LIR)))
        {
                prev_ptr = ltmp->prev;
                tmp->next = ltmp;
                tmp->prev = prev_ptr;
                prev_ptr->next = tmp;
                ltmp->prev = tmp;
        }
        else
        {
                printf("OOPS");
        }/*if((t_flag == NOT_FOUND) && (tmp->flag ==LIR))*/
}/* end of put_end_lirs*/

void put_end_hir(*tmp)
struct hir_q *tmp
{
        struct hir_q *prev_ptr, *next_ptr, *htmp;
        short t_flag =FOUND;
        /* remove the block from its location*/

        next_ptr = tmp->next;
        prev_ptr = tmp->prev;
        prev_ptr->next = next_ptr;
        next_ptr->prev = prev_ptr;
        /* find the last element on hir_q*/

        htmp = hir_rear = hir_front;
        while(htmp->flag!= NOT_IN_USE)
        {
                htmp = htmp->next;
                if(htmp == hir_rear)
                {
                        t_flag = NOT_FOUND;
```

```
                                        break;
                                }
                }
                if((t_flag == FOUND) || ((t_flag == NOT_FOUND) && (tmp->flag == IN_USE)))
                {
                                prev_ptr = htmp->prev;
                                tmp->next = htmp;
                                tmp->prev = prev_ptr;
                                prev_ptr->next = tmp;
                                htmp->prev = tmp;
                }
}/* end of put_end_lirs*/

void set_lirs_front_ptr()
{
                /* to reorganize the front pointer of lirs queue*/
                struct lirs_q *tmp;

                tmp = lirs_front->next;

                while((tmp->flag != LIR) && (tmp != lirs_rear))
                {
                                tmp->flag= NOT_IN_USE;
                                tmp= tmp->next;
                }

                lirs_rear = lirs_front = tmp;        /* put lisr_front at lir block */

}

void set_lirs_last(block, dev, flag)
int block; int dev; short flag;
{
/*set the last block of lirs_q with data*/
                struct lirs_q *ltmp, *ltmp1, *ltmp2;
                short t_flag = FOUND;
                struct hir_q *htmp;
/*              if(lir_in_use == NR_HIR) return();*/

                ltmp = lirs_front;
                while(ltmp->flag != NOT_IN_USE)
                {
                                ltmp = ltmp->next;

                                if(ltmp ==lirs_rear)
                                {
                                        t_flag = NOT_FOUND;
                                        break;
                                }
                }
                if(t_flag ==FOUND)
                {
                                ltmp->block = block;
                                ltmp->dev = dev;
                                ltmp->flag = flag;
                }
                else
                {

                                ltmp2 = lirs_front;
                                ltmp1 = lirs_front->next;

                                while(ltmp1->flag != HIR_P)
                                        ltmp1 = ltmp1->next;

                                ltmp1->flag = LIR;

                                htmp = hir_rear = hir_front;

                                while((htmp->block != ltmp1->block)
                                        ||(htmp->dev != ltmp1->dev)
                                        ||(htmp->flag == NOT_IN_USE))
                                {
                                        htmp = htmp->next;
                                }

                                if(htmp == hir_front)
                                        hir_front = hir_rear= htmp->next;

                                put_end_hir(htmp);
                                htmp->flag = NOT_IN_USE;
                                hir_in_use--;

                                set_lirs_front_ptr();
                                put_end_lirs(ltmp2);

                                ltmp2->block = block;
                                ltmp2->dev = dev;
                                ltmp2->flag = flag;
                                printf("End");
                }
}


void set_hir_last(block, dev)
int block; int dev;
{
/*set the last block of hir with data*/
                struct hir_q *htmp;

/*      if(hir_in_use == NR_HIR) return();*/

                htmp =hir_rear= hir_front;
                while(htmp->flag != NOT_IN_USE)
                {
                                htmp = htmp->next;
                                if(htmp == hir_rear) break;
                }
                htmp->block = block;
                htmp->dev = dev;
                htmp->flag = IN_USE;
                hir_in_use++;
}

void set_hir_full(block,dev)
int block; int dev;
{
/* fill the entry if hir_q is full*/
                struct lirs_q *ltmp1;
                int t_flag = FOUND;
                hir_rear = hir_front;
                /* set the first block of hir_q*/
                /* block as HIR_NP in lisr q*/
                ltmp1 = lirs_front;
                while((ltmp1->block != hir_rear->block)||
                                        (ltmp1->dev != hir_rear->dev)||
                                                (ltmp1->flag == NOT_IN_USE))
                {
                                ltmp1 = ltmp1->next;
                                if(ltmp1 == lirs_rear)
                                {
                                        t_flag = NOT_FOUND;
                                        break;
                                }
                }
                if(t_flag == FOUND)
                                ltmp1->flag = HIR_NP;
                /* now remove the front element of hir*/
                hir_front = hir_front->next;
                hir_rear->block = block;
        hir_rear->dev = dev;
        hir_rear->flag = IN_USE;
        hir_rear = hir_front;

}

void lirs_found(*tmp)
struct lirs_q *tmp;
{
/* if the element found in lirs_q */
        struct lirs_q *tmp2, *ltmp1, *next_ptr, *prev_ptr;
        struct hir_q *htmp;

        if(tmp->flag == LIR)
        {   /* free the lir block and put at the rear end of lisr*/
                if(tmp == lirs_front)
                {
                /* reorganize the front ptr of lisr_q*/
                        set_lirs_front_ptr();
                }
                put_end_lirs(tmp);

        }/* if flag == lir */
        else if(tmp->flag == HIR_P)
        {
                /* if the block is an hir present block */
                /* free the tmp(just found in lirs) block and put
                 at the end of the lirs q. set its flag LIR*/
                put_end_lirs(tmp);
                tmp->flag = LIR;

                /* search the tmp block in HIR Q and*/
                /* put it at the end of hir to free it*/
                htmp = hir_front;
                while((tmp->block != htmp->block)||
                                (tmp->dev != htmp->dev)||
                                        (htmp->flag == NOT_IN_USE))
                        htmp = htmp->next;
                /* put this HIR_P block of hir_q at end of hir_q
                 and set its flag not in use*/
                if(htmp == hir_front)
                        hir_front = hir_rear= htmp->next;

                put_end_hir(htmp);
                htmp->flag = NOT_IN_USE;
                hir_in_use--;
                                /* place the content of front element of lirs_q
                at the rear end free block of the hir_q*/
                set_hir_last(lirs_front->block, lirs_front->dev);

                /*reorganize the front pointer of lirs_q*/
                lirs_front->flag = NOT_IN_USE; /* free the front block of lirs_q*/
                set_lirs_front_ptr();
        }
        else if(tmp->flag == HIR_NP)/* if part of HIR_P */
        {
                /* if the block is an hir_q non present block */
                /* free the tmp(just found in lirs) block and put*/
                /*at the end of the lirs q. set its flag LIR*/
                put_end_lirs(tmp);
                tmp->flag = LIR;
                                /* place the content of front element of lirs_q
                at the rear end free block of the hir_q*/
                if(hir_in_use >= NR_HIR)
                {
                        set_hir_full(lirs_front->block, lirs_front->dev);
                }
                else
                {
                        set_hir_last(lirs_front->block, lirs_front->dev);
                }
                                        /*reorganize the front pointer of lirs_q*/
                lirs_front->flag = NOT_IN_USE; /* free the front block of lirs_q*/
                set_lirs_front_ptr();

        }/* else part of HIR_NP*/

}/*end of lirs_found*/

void lirs_not_found(block,dev)
int block;int dev;
{
        struct lirs_q *ltmp, *next_ptr, *prev_ptr;
        struct hir_q *htmp;
        short t_flag = FOUND;
        if(lir_in_use < NR_LIR)
        {
                set_lirs_last(block,dev,LIR);
                lir_in_use++;
        }
        else if(hir_in_use < NR_HIR)
        {
                set_lirs_last(block, dev, HIR_P);
                htmp =hir_rear =hir_front;
                while((block != htmp->block)||
                                        (dev != htmp->dev)||
                                                (htmp->flag == NOT_IN_USE))
                {
                        htmp = htmp->next;
                        if(htmp == hir_rear)
                        {
                                t_flag = NOT_FOUND;
                                break;
                        }
                }
                /* put this HIR_P block of hir_q at end of hir_q
                 and set its flag not in use*/
                if(t_flag == FOUND)
                {
                        if(htmp == hir_front)
                                hir_front = hir_rear= htmp->next;

                        put_end_hir(htmp);
                }
                else
                {
                        set_hir_last(block, dev);
                }

        }
        else if(hir_in_use >= NR_HIR)
        {
                set_lirs_last(block, dev, HIR_P);

                htmp =hir_rear =hir_front;
                while((block != htmp->block)||
                                        (dev != htmp->dev)||
                                                (htmp->flag == NOT_IN_USE))
                {
                        htmp = htmp->next;
                        if(htmp == hir_rear)
                        {
                                t_flag = NOT_FOUND;
                                break;
                        }
                }
                /* put this HIR_P block of hir_q at end of hir_q
                 and set its flag not in use*/
                if(t_flag == FOUND)
                {
                    if(htmp == hir_front)
                        hir_front = hir_rear= htmp->next;

                    put_end_hir(htmp);
                }
                else
                {
                    set_hir_full(block, dev);
                }
        }
}

void lirsmain(int block, int dev)
int block; int dev;
{

        struct lirs_q *tmp;


        short t_flag= FOUND;
        /* try to search in lirs q */
        tmp = lirs_front;

        while( (tmp->block !=block)||
                    (tmp->dev != dev) ||
                        (tmp->flag == NOT_IN_USE))
        {
                tmp = tmp->next;
                if(tmp == lirs_rear)
                {
                    t_flag = NOT_FOUND;
                    break;
                }
        }

        if(t_flag == FOUND)
        {
                printf("\t found");
                lirs_found(tmp);
        }/* if t_flag == found*/
        else
        {
                lirs_not_found(block, dev);
        }/*else t_flag == found*/
}
```