1986

Addition of Structured Records to the UNIX™ and MS-DOS™ File Systems

Nancy Wisotzke

Rochester Institute of Technology
School of Computer Science and Technology

Addition of Structured Records to the UNIX(tm) and MS-DOS(tm) File Systems

by
Nancy Wisotzke

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by: _____ 12/8/86
Professor Jeff Lasky

_____
Professor Warren Carithers

_____
Professor Chris Comte

Title of Thesis: ADDITION OF STRUCTURED RECORDS TO THE UNIX (tm) AND MS-DOS (tm) FILE SYSTEMS

I _____ hereby (grant/deny) permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

or

Nancy K Wolfe

UNIX is a registered trademark of AT&T Bell Laboratories

MS-DOS is a registered trademark of Microsoft Corporation

Digital Equipment Corporation, my employer, assumes no responsibility for the contents of this thesis.

CONTENTS

CHAPTER 1

INTRODUCTION

A file system consists of operating system code supporting the I/O operations that open, close, create, read and write to files. Record oriented I/O is not a feature of either the UNIX or the MS-DOS native file system. The data is stored by the file system as a sequence of bytes. It is the task of the user to design an application dependent structure on the file system. In the case of logically related fixed or variable length data, a record structure is desirable.

This thesis describes a set of extensions to the UNIX and the MS-DOS file systems that supply a record structure to these native file systems. The descriptions of both the UNIX and the MS-DOS native file systems are followed by the extensions to each file system. Program code is provided to demonstrate the MS-DOS extensions.

1.2 COMPARISON OF UNIX AND MS-DOS NATIVE FILE SYSTEMS

The UNIX and MS-DOS file systems have many similarities. Each file system provides minimal functionality; that is, the basic abilities to read, write, open, close and create files. Reading and writing are sequential operations under both file systems. Both the UNIX file system and the MS-DOS file system are hierachical in structure. Each contains a root directory with branches leading to leaves of alternate directories and files. A file is found by tracing a path leading from the root directory. No major differences were found between the two file systems.

1.3 IMPLEMENTATION OF THE EXTENSIONS

The extensions to the MS-DOS file system are implemented in KOALA, a Digital Equipment Corporation proprietary language. KOALA is a block structured language similar to ALGOL, ADA and PASCAL. KOALA offers user-defined statements, functions and operators. Data types consist of integers, bytes, strings, arrays, characters and records. A record can consist of any combination of data types. KOALA routines consist of functions and statements, where functions return a value, and both functions and statements accept parameters. When another program needs to call a

function or statement the term EXPORT is placed in front of that function or statement definition. The same holds true for constants, variables and types. The term EXTERNAL placed in front of a function or statement definition indicates that the function or statement is found in a macro assembly language module.

The extensions to the UNIX file system are described from the user point of view. The different methods which provide record structure are presented along with the advantages and disadvantages of each.

1.4 THESIS ORGANIZATION

The remaining chapters of this thesis are structured as follows:

Chapter two describes the native UNIX file system and its extensions. It details the internal structures present, the allocation of space on a disk, and the native file system functions. The extensions to these functions to provide record structures are then discussed.

Chapter three describes the native MS-DOS file system. The hierarchical structure, the disk space

allocation and each system I/O function are discussed. The system function descriptions are followed by the record structure extensions. Application programs which demonstrate the use of the extended MS-DOS file system are also presented. The code which extends the native MS-DOS file system is documented and accompanied by a user guide. The demonstration of the new file system and its record structure are implemented on a Rainbow personal computer running MS-DOS version 2.11.

Chapter four summarizes what was learned while structuring these two file systems. Future enhancements to the file systems are also discussed. The indexed record structure is described along with the B+tree indexing method. The integration of the B+tree index into the UNIX file system and the MS-DOS file system is presented.

Appendix A includes data structures, illustrations of both the UNIX and the MS-DOS file systems, and a glossary.

The bibliography follows the Appendix. Each entry includes a short description of the book or article.

CHAPTER 2

THE UNIX FILE SYSTEM

A file under UNIX is a named sequence of bytes. UNIX files are organized in a hierarchial directory structure. The root directory is at the top of the hierarchy. A file is found by searching through the directory tree. The pathname describes this search. The pathname consists of a series of directories, separated by slashes ('/'), followed by the name of the file. If a path name begins with a '/' then the search begins at the root directory, if not, the search begins at the current directory. The current directory is the directory from which you can directly access a file. To change the current directory, the cd command is used. Every process has a current directory. There is a ".." and a "." entry found in each directory. The ".." entry refers to the parent directory and the "." refers to the directory itself. In the case of the root directory both the ".." and the "." entries refer to the root.

The file system is accessed using block I/O. In UNIX, a disk is an array of blocks, divided into four or five areas (depending on the UNIX version). (See Figure 2-1). The first disk block is not used by the file system. Its use varies from system to system; however it often contains the initial UNIX system bootstrap(start-up) program.

The super block plays an important role in maintaining the file system. There are two versions of the super block, the on-disk and the in-core super block. The second disk block holds the on-disk super block. This block is read in at system initialization time and holds information on the size of the disk and the layout of the other areas on the disk. It also keeps track of free block and inode usuage on the disk.

The in-core super block is allocated when a mount command is issued and released when an unmount command occurs. Mount causes a file on the original file system to refer to the root directory of the file on the removable volume. Once the mount is executed, the files on the file system are accessed as if they are part of the original file system. The UNIX kernel's mount table holds the in-core super blocks of all mounted file systems. The unmount command removes the in-core super

block from the mount table.

Directories are considered files on UNIX with the restriction that they cannot be written to. Each file has an entry in either the root directory or a sub-directory. Since these sub-directories are considered files, they have an entry in their parent directory. Directory entries are 16 bytes in length and consist of a 2 byte i-number and a 14 byte file name. Each directory entry maps the file name to its corresponding inode.

The UNIX file system maintains an ilist (following the super block) on each disk. The ilist contains an inode for each file or directory on the disk. The offset of an inode into the ilist is the i-number.

The inode is the central structure for all file access in UNIX. There are two types of inode structures, the on-disk inode and the in-core inode. A file that resides on disk and is not currently being accessed has an inode entry in the ilist or on-disk inode table. The information found in the on-disk inode is read into an in-core inode when the file is accessed. Now both an on-disk and an in-core inode are associated

with the file. In-core inodes are allocated for each file that is active, for current directories, for each mounted-on file, for each text file, and for the root. The root directory's inode is at a known i-number and all files may be found by tracing a path from the root. This information is used when converting a path name to an in-core inode table entry. A search for a file's inode starts at some known inode (either the root or the inode of the current directory). This directory is searched for the next portion of the pathname. The directory entry found gives an i-number and a filename. When the filename found matches the last component of the pathname, the i-number of the directory entry maps to the desired in-core inode. Any modifications to the inode are made at this level. When the file is no longer accessed the in-core inode is copied back to the ilist and the in-core version is freed. The in-core inode is described next with the fields that are common to both inodes indicated.

UNIX Disk Structure

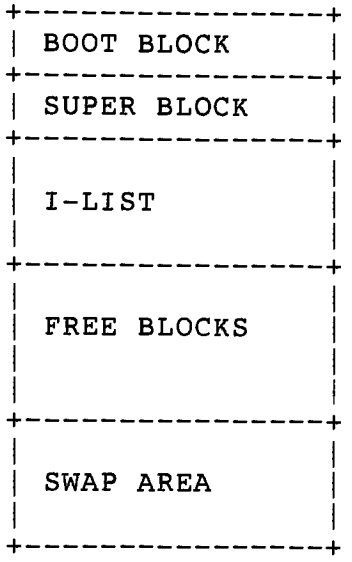


Figure 2-1

The inode structure holds the file description. The first four fields are unique to the in-core inode. The flag field indicates whether the inode is locked, has been modified, is mounted on, has a process waiting on the lock, or if the access time needs to be updated. The count field holds a reference count of the number of files which reference the inode. If this reference count is zero, the inode is unallocated. There may be several file table entries that point to one inode. The device on which the inode resides is held in a field in the inode. The internal file name for the inode, called the i-number on the UNIX file system, is held in the next field.

The information for the remaining inode fields is read into the in-core inode from the on-disk inode on the volume. The fields consist of a mode (which describes file usage, type, permissions), links to the directory entries, the owner, the group id of the owner, the file size, the addresses of the device that constitute the file, the time that the file was last accessed and last modified, and the time that the inode was last modified.

The device addresses in the inode point to free space where disk blocks for the file can be obtained. (See Figure 2-2). The free block area is the fourth section of a disk. The size of a block differs over the various versions of UNIX. A block is traditionally 512 bytes; however, under 4.1BSD its size is 1K bytes, and under 4.2BSD a block can be up to 8K bytes. Under version 6 there are 8 device address pointers in the inode. This number becomes 13 in version 7. Under version 7 UNIX, the first ten addresses in this field point to the first ten blocks of free space available for a file. The eleventh address points to a block that contains the addresses of the next 128 blocks of the file. The twelfth block points to 128 more blocks, each which in turn point to 128 blocks of free space for the file. The thirteenth address holds "triple indirect"

addressing. Each address points to 128 blocks, each which in turn point to 128 blocks that point to 128 blocks of free space. The maximum file size, in this case, is 2,113,673 blocks. Triple indirect addressing is not used. Most disks are smaller than the one billion byte file size limit.

File allocation occurs when space is reserved for a new inode in the memory resident inode table and the disk inode is copied into this entry.

There is a fifth disk area in some versions of UNIX. This is the swap area. This area holds the user processes that have been swapped out of main memory.

Inode File Access

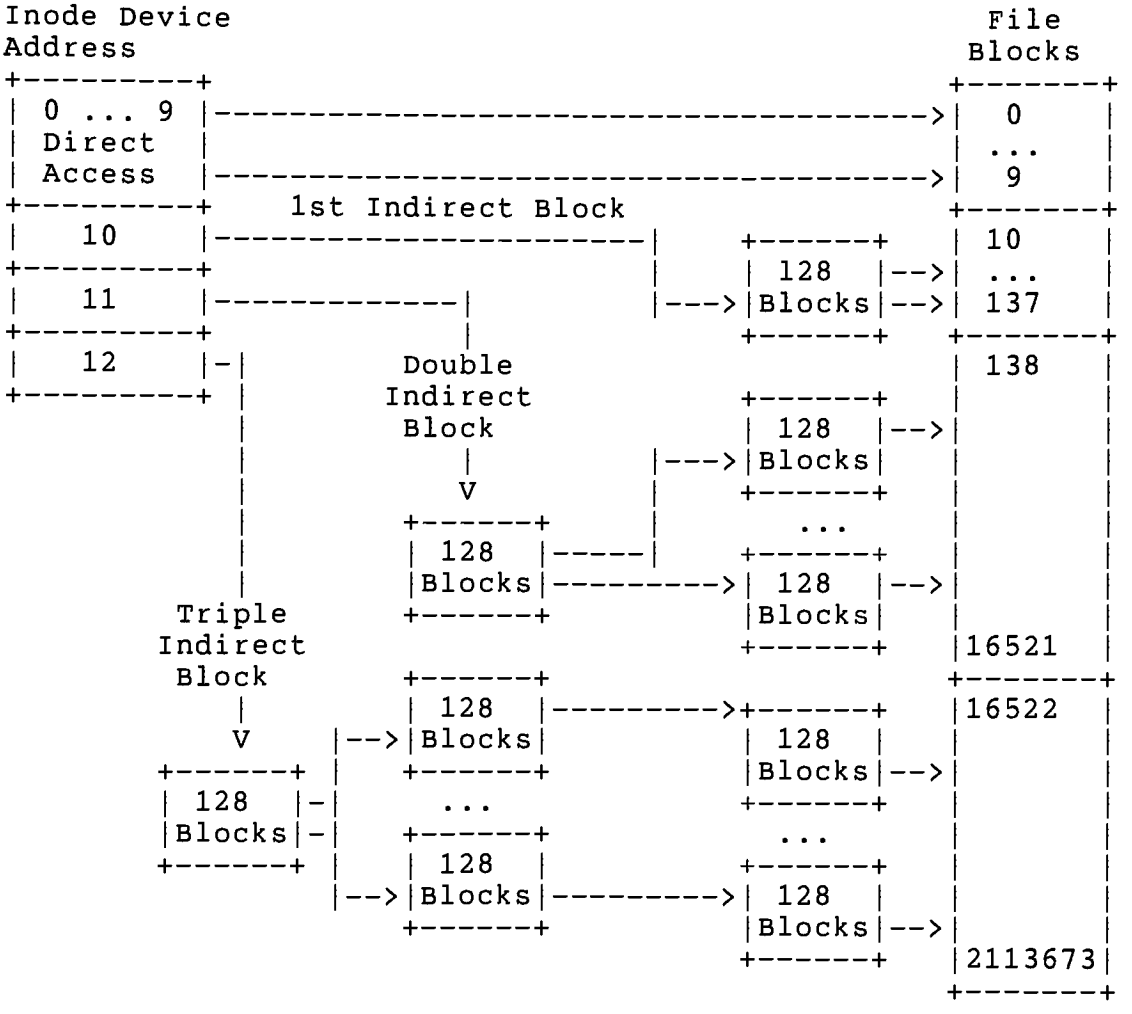


Figure 2-2

2.2 I/O SYSTEM IMPLEMENTATION

A user program is an application which has a need to access UNIX files. The user accesses a file by executing a create or open call to the UNIX operating system. The UNIX I/O system contains the code that manipulates the file system. A call to the UNIX I/O system to open or create a file causes the process' user descriptor to point to the open file array or table. This table holds a pointer to the entry in the inode table. The inode table points to the actual file. (See Figure 2-3).

UNIX File Access from User Descriptor to File

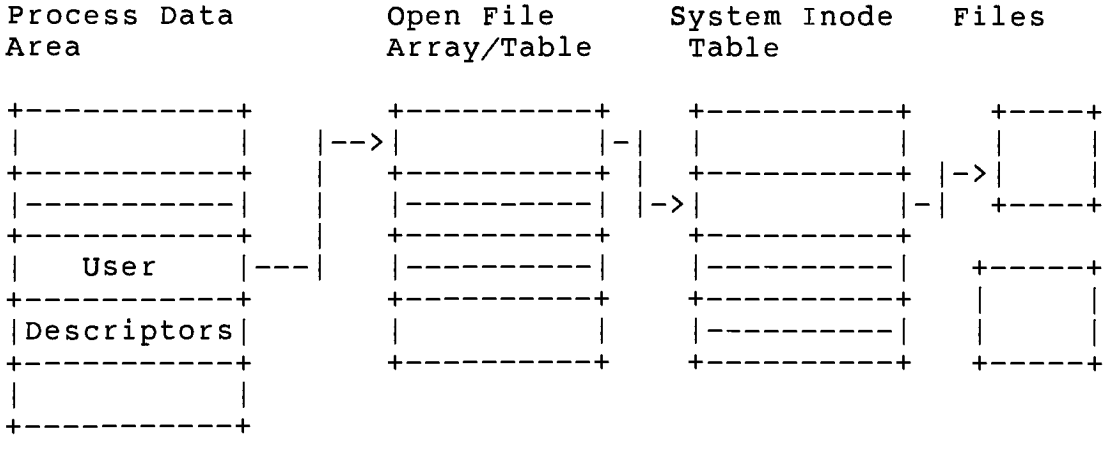


Figure 2-3

The open file array contains file structures which hold information necessary for all file activity under UNIX. Each time an open or creat system call is

successful a file descriptor is returned. The file descriptor is the file system's method of accessing the file. The file descriptor is a field in the file structure. The file structure also contains a pointer to the next character to be read or written, a flag field indicating whether the file is being read from or written to, and the location of the I/O buffer.

Different resources are needed for a file depending on its status. A file that exists but whose count field is zero needs an entry in the directory hierarchy, a disk inode entry, and zero or more blocks of disk storage. Once the file is referenced an in-core inode entry is also required. A file that is opened for reading or writing needs an entry in the file array and an entry in the user program's process data area (which points to a file array entry). Thus far, the allocation of storage, the file structure and the inode structure have been described. I/O operations on files are presented next; it is here that the record structure is implemented.

2.2.1 CREAT

To create a file that does not currently exist, or truncate an existing file to zero, the creat(path, mode) system call is used. The path parameter is a valid UNIX

pathname. The mode consists of three octal digits which indicate the file permissions of the owner, the group and the world. The permissions are specified by 4 for read, 2 for write, and 1 for execute and are added together to specify the desired permissions.

Creat checks the pathname to see if a file with that name already exists. In the case where the file does not exist a core inode must be allocated for the file. The super block keeps track of these free inodes. Only one hundred inodes are kept in the on-disk super block at a time and when the hundred are used, a linear search of the ilist is performed to obtain the next hundred inodes. The core inode fields (flag, count, device, and i-number) are then assigned. A directory entry is made with the pathname and the i-number. Creat then checks the permissions on the file, assigns values to the file structure, and calls a device open system call.

When a file of the same name already exists, access permissions are checked and any previous contents of the file are removed. This is done by a procedure which frees all disk blocks that are associated with the inode. The file descriptor and the file structure are allocated. The file descriptor is initialized to point

to the file structure and the file is opened.

If an error occurs during the creat, the inode is released and -1 is returned in place of the file descriptor. Errors occur when the directory is not found, the directory cannot be written to, the file exists and cannot be written to, the file is a directory, or there are too many open files.

2.2.2 OPEN

To open a file, the application uses the open(path, mode) system call. The path parameter is a valid UNIX pathname. The mode parameter indicates the type of file access desired and is 0 for read, 1 for write, and 2 for read and write access.

Open checks the pathname to see if a file with that name already exists. If the file exists, an entry is made into the inode table and an open file table entry holding the pointer to the inode table entry is created. Open associates a file descriptor with the file. This file descriptor must be used for subsequent I/O operations on the file. The I/O pointer is positioned at the beginning of the file (byte 0) and the file is opened according to the access permissions requested by

the user. If an error occurs during the open, a -1 is returned in place of the file descriptor. Errors occur if the file does not exist, if the directory does not exist, if the directory cannot be read, if the file cannot be read (if open for read), or written (if open for write), or if there are too many open files.

The open and creat system calls set up entries in three system tables. The entry in the open file table indicates whether the file is open for reading, writing or is a pipe. An entry is made in the inode table and in the user process table.

2.2.3 WRITE

To write to a file in UNIX, a call is made to the write(filedesc, buff, nobytes) system call. The filedesc argument is the file descriptor that is returned from the creat or open system call. The buff argument is the buffer to write to the file from, and the nobytes argument is the number of bytes to be written.

Write converts the user file descriptor into a pointer to the file structure. It then checks that the file was opened for write. The inode is accessed

through the file structure in the open file table.

For a character special file, the appropriate procedure is called to perform a write to the device. A loop is then begun to write the requested number of bytes. The physical block number of the file to be referenced is determined, the character offset in the file is set, and the minimum of the number of bytes left in the block and the number requested to be written is set. If the file is not a special block file, the physical block number is determined using the inode pointer and the block number determined previously. The number of bytes to be written is checked. If it is a full block, a buffer is assigned for the indicated block. If the desired block is located and already associated, it is returned. The available list holds buffers that are currently in use but may be reassigned for an alternate use. The first buffer from this list is returned. In the case of less than a full block being written, the contents currently in the buffer are read. This is done in order to save the contents in an area of the buffer that is not in use.

The number of bytes requested for transfer are moved from the user buffer area to the kernel file area. If the new value of the file offset points beyond end of

file, then the file size has increased. The inode modification flag is set and the loop to write to the file continues, as long as no error occurs and the count to be transferred is not zero. The number of bytes written is returned from the file system call. An error has occurred if this value is not the same as the number of bytes requested for writing.

If an error has occurred during the write, a -1 is returned from the system call. Errors occur when a bad file descriptor, buffer address, or count are passed to the system call, or when a physical I/O error occurs. On an error the buffer is released to the available list. No write ahead is present in UNIX; however, if the characters in the buffer have not changed the buffer is marked and released. If the buffer is taken for another use this mark indicates to the file system that it should be written out first.

2.2.4 READ

A read is done with the read(filedesc, buff, nobytes) system call. The filedesc parameter is the file descriptor returned from the creat or open system call. The buff parameter is the location of the buffer that holds the data that is to be read, and nobytes is the number of bytes to read.

Read converts the file descriptor into a pointer to the file structure. The access mode in the inode is checked to determine if the file is open for read. A check is then made to insure that the number of bytes to be read is not zero. If this count is zero, the procedure returns. The inode is accessed through the file structure in the open file table. A loop is then set up to transfer or read the data from the file.

The file offset of the block to be read is compared to the file size. The number of bytes to be read is set to the minimum of the number requested and the number of bytes left in the file. The read-ahead block is set. The blocks of the file may be read sequentially. In this case, once a block is read, I/O begins on the read-ahead block.

The number of bytes requested for transfer are moved from the kernel area to the user buffer area. The read loop continues until either an error occurs, or until all the bytes requested have been read. The number of bytes read is returned from the system call. If this value is zero, end of file has been reached. The number of bytes returned may be less than the number requested. This is not an error but indicates that fewer than the requested number of bytes remain in the

file.

If an error has occurred during the read, a -1 is returned by the system call. Errors occur when a bad file descriptor, buffer address, or count are passed to the system call, or when a physical I/O error occurs.

2.2.5 I/O BUFFERS

At this point, to conclude the description of the UNIX file system, the buffers are described in more detail. The I/O buffers are the method through which the kernel keeps track of the data from the user program and the data on the disk. Buffer headers are identified with a device number and a block number. These headers are kept in two lists, the b-list and the av-list. The b-list links together buffers that are associated with the device type. The av-list is the available list which holds buffers that can be released from their present use and allocated for different tasks. A busy flag is set when a buffer is allocated from the available list. A done flag is set when the information in the buffer is correct. Another flag is set when the buffer contents need to be written out before being reassigned. The buffers are not assigned to a file until needed. This allows for a small number of buffers to be used by many files. Nothing is removed from the

buffers until they are needed and if only part of the buffer is needed, the remainder of the information is not altered. The kernel controls writing of the buffers and a utility program forces buffers to be written out unconditionally twice a minute, in order to preserve their contents.

2.3 STRUCTURING OF THE FILE SYSTEM

This section describes the extensions to the native UNIX file system which provide record structure. There are three different record formats provided to implement this structuring. The record format is the way that the record physically appears on the storage medium. The three record formats supported are:

- Fixed length
- Variable length counted
- Variable length stream

When all of the records in a file are the same length, fixed length record format is used. The size of the record is set at compile time to the size of the user data structure. Fixed length records are represented by fixed length data types such as, integers, characters, arrays, and structs containing fixed length fields.

When the size of the records in a file varies, the variable length record format is used. Variable length records are represented as strings. In a variable length counted record file the data is preceded by a two byte count field, which contains the size of the record. The variable length stream record format delimits each record with a newline character. Both the two byte count field and the newline character aid in the retrieval of the record from the file.

There are two methods of adding these record formats to the native UNIX file system. One method modifies the inode and some of the existing system calls. The other provides a set of library routines to the user, that implement the record I/O operations. Each of these structuring methods are now discussed.

2.3.1 Addition Of Library Routines

The addition of C library level functions to structure the file system has several advantages. This method makes the record formatting invisible to the user, also this method is portable. The library routines can be installed on other systems. One disadvantage to this method is that a header record is used to hold information about the file on the storage medium. This takes up space, and other utilities may

not know about this header.

The header can be attached to the data in the file, or located separately from the data. When the header is attached to the file, it is not necessary to keep track of the address of the first data record. It immediately follows the header. The disadvantage to this method is that the header takes up space, and there may not be enough space available on the device to write both the header and the data. Using the method where the header is detached from the data makes it necessary to keep a pointer to the first data block of the file.

2.3.1.1 Fixed Length Records

The record format of the file is decided at compile time by the format of the user data structure that is used throughout I/O operations on the file. The file header is set up in the creat and open library calls. Space for the header is allocated in the creat, and the record format field is filled in. The record length is also written to the header. This length is the size of the user data structure at compile time. The open system call checks the file header to see if the user data structure's record format matches the record format that was set up in the creat. If the record formats do

not match, an error occurs.

To write a record to the file, the library function rwrite(fd, buff) is used. The fd parameter is the file descriptor that is returned in the creat or open. Buff is the buffer containing the data to be written to the file. This library function makes a call to the write system call with the file descriptor and write buffer from the rwrite call, and the record length from the header record. Write returns the number of bytes written, or -1 if an error has occurred. When the number of bytes written is returned, it is checked against the record length. If the two are not equal, an error has occurred and a full record cannot be written. Rwrite returns 0 when the write completes successfully, and -1 when an error occurs.

To retrieve a record from a fixed length file the rread(fd, buff) library function is used. The fd parameter is the file descriptor that is returned from the creat or open call. Buff is the buffer that the data is read into. The rread function uses the system read call to transfer the data from the file. The system read is called with the file descriptor and the buffer from the rread call, and the record length from the file header. The number of bytes read is returned

from read, or -1 is returned when an error occurs. The number of bytes read is compared to the record length and if the two are not equal, an error has occurred. Rread returns 0 when the read completes successfully, and -1 when an error occurs.

2.3.1.2 Variable Length Records

The library routines to handle variable length records use the counted record format. When stream record format is to be used, calls are made to the existing I/O system calls. These calls currently implement stream I/O.

The creat library function allocates and sets up the file header. The record format field is set to variable. The record length field is set to 0, indicating that it is not used. This is because the record length varies with each I/O operation on the file. The open library function checks the record format of the user data type against the record format in the file header. If the two are not the same, an error has occurred and the open fails.

To write a variable length record to the file the rwrite(fd, buff) library function is used. The fd parameter is the file descriptor that is returned from an open or creat. The buff parameter is the buffer to write from. Rwrite uses the strlen function on the data in buff to determine the record length. The length bytes are concatenated onto the data string inside buff. A call is made to the write system call to transfer the data to the file. Write takes the file descriptor, the count/write buffer, and the record length plus two, as parameters. Two is added to the record length to accomodate the two byte count field. The number of bytes written is compared to the record length. An error occurs if the two are not equal. Rwrite returns the number of bytes written, or -1 if an error has occurred.

To read a variable length record from the file, the rread(fd, buff) library function is called. The fd parameter is the file descriptor that is returned from the creat or open. Buff is the buffer to transfer the data from the file into. The rread function first calls the read system function with the fd and buff parameters from rread, and two for the record length parameter. Two bytes are read from the file into buff, which now contains the record length. The read system call is

executed again with fd, buff, and the record length (that was just returned), as parameters. Read returns the number of bytes read, or -1 on an error. The number of bytes read is compared to the record length and an error is returned if they are not equal. Rread returns the number of bytes read when the read completes successfully, or -1 when an error occurs.

2.4 STRUCTURING AT THE SYSTEM LEVEL

An alternative to the library routine method of structuring is the addition of the record structuring code at the system level. This method is used when the applications need to know about record formats. The user must learn about the record formats available and choose the one that best suits each application. One disadvantage to this method is that changes are made to the inode and some system calls, which adds code and complexity to the native UNIX file system.

2.4.1 Data Structure Changes

At this point, a discussion of the data structure changes is presented, followed by a discussion of the changes to the system calls. The first change is to the in-core inode. A field is added to the inode which describes the record format of the file as either fixed, variable counted, or variable stream. A new system

call, set_rec_format(format), is added to the kernel code to set this field in the inode. The user makes a call to this function before issuing an open or creat system call, to indicate the type of variable record format that is to be used. The creat and open system functions are changed to set the record format field in the inode. If the set_rec_format call is not issued, a check is made inside the creat or open, to see if the record format is fixed (the user data structure is fixed length). If it is not, the variable record format defaults to stream. The inode record format field is set accordingly. The user interface to the creat and open system calls does not change.

2.4.2 Fixed Length Records

The write(fd, buff, reclen) system call is used to write records to the file. The user interface to this call does not change. The fd parameter is the file descriptor that is returned from the open or creat system call. The buff parameter holds the data which the user assigned to the record that is to be written. The reclen parameter is the length of this record, and remains constant throughout other writes to the file. The number of bytes written is returned by the write function and is checked against reclen. An error is returned if the two are not equal.

Random writes on fixed length records are also allowed. The record number is passed to the write by the user. The random write call uses a new parameter. This new system function is added to UNIX in a layer of code above the kernel code. The random write function provides a method of writing to a specific record in the file.

numwrit = randwrit(fildesc, buf, reclen, recno) ;

The record length(reclen) is multiplied by the record number(recno) and lseek is called to position the file pointer to the desired record. The write system call is then called to write the data to the file. The number of bytes written is checked against the record length and an error is returned if the two are not equal.

To read records from the file, the read(fd, buff, reclen) system call is used. The user interface to this call does not change. The fd is the file descriptor returned from the creat or open, the buff parameter is the buffer that the record is to be read into. For fixed length records, the count to be read is always the size of the user data record. The reclen parameter is set to this size and remains fixed during reads to the file. A read should always return a record of the same size. The number of bytes read is returned by the read

function. A check for the number of bytes transferred and the number of bytes requested is made. An error is raised if they are not equal. A comparison must be made between the number of bytes requested and the number remaining in the file. If the number requested is greater than the number of bytes remaining in the file, end of file is returned.

Random access reads add a new parameter to the read function call. The record number is passed to the function by the user. This new system call is added to UNIX in a layer of code on top of the kernel code. The random read function provides a method for reading a specific record of the file.

```
numread = randread(fildesc, buf, reclen, recno) ;
```

Random reads are executed by multiplying the record number(recno) by the record size(reclen) and calling lseek to position the file pointer. The read system function is then called to read the record. The number of bytes read is returned. This count is checked against the record size and an error is returned if the two do not match. The random read also tests for end of file. This is done by comparing the record length to the number of bytes remaining in the file. If the number of bytes remaining in the file is less than the

record length, end of file is returned.

2.4.3 Variable Length Records

A variable length record can be written to the file as a counted record or a stream record. To set the record format, the set_rec_format call is used. The record format field in the inode is then set in the open or creat system call. When the record format is set to stream, the current read and write system calls are not changed. These functions already provide stream record I/O. A discussion of the changes necessary to implement counted records is now presented.

To write a counted record to the file, a new system call, rwrite(fd, buff), is used. This system call takes the file descriptor, and the data buffer to write from as parameters. Inside this function the length of the record is determined by the strlen function. This length is concatenated to the front of the data to be written. The write system call is then used to write the data. The buff parameter holds the new data string (count and data), and the reclen parameter is the length of the data plus the two bytes for the count. The number of bytes written is returned and checked against reclen. If the two are not equal, an error is returned. The rwrite call returns the number of data bytes that

have been written to the file, or -1 if an error occurs.

A new system call, rread(fd, buff), is used to read variable length counted records. Inside this function, a call is made to the read system call to retrieve the first two bytes from the record. These bytes represent the length of the record. The read system call is executed again with this record length as a parameter. The record's data is then retrieved from the file. The number of data bytes read is returned from the rread call, or -1 is returned if an error has occurred.

2.5 CONCLUSIONS

The current Unix file system and two methods of structuring it have just been presented. The system administrator must weigh the advantages and disadvantages of each in order to decide which method to use. The amount of control that the user requires over record formatting must also be taken into consideration.

CHAPTER 3

THE MS-DOS FILE SYSTEM

A file under MS-DOS consists of a series of bytes. Access to the file is through a pathname. This chapter describes the pathnames which access the file, the storage of directories and files on the disk, and the organization of a disk. Next, the MS-DOS file system calls are discussed followed by the extensions to the native file system which implement a record structure.

The directory structure on MS-DOS is hierarchial. The directories are the branches and the files are the leaves. The root is at the top of the hierarchy and is the directory that is automatically created when the disk is formatted. A file is found by searching the directory tree. The pathname describes this search. A pathname is a series of directories, separated by backslashes (\), followed by the file name. If a pathname begins with a "\" then the search begins at the

root directory. If not, the search begins at the current directory. A path name is often preceded by a device name indicating the disk on which the file can be found. If no device is indicated, the default device is used. A device name identifies a disk by a single letter. It is separated from the pathname by a ":". A discussion on the storage of directories and files and the organization of a disk follow.

MS-DOS reserves one or more of the first tracks on a disk for storing a loader program. These reserved tracks are called system tracks. All of the remaining tracks are used for storing files. Data files are written wherever there are empty locations and there can be many locations containing portions of a file. An entry is made into a directory and the File Allocation Table (FAT) for each file on a disk. This tells MS-DOS where the file is located on the disk.

MS-DOS divides a disk into groups of logical records called clusters. This provides for better data management. The data storage area of each disk is addressed in terms of these clusters. They are numbered from zero on the outside of the disk sequentially inward with increasing cluster numbers. The number of clusters containing the file's data and the number available for

use by the file system are stored in the disk's directory and File Allocation Table. The first few clusters on each disk are reserved to store its directory, the number indicating the capacity of the disk and the cluster size.

Directories, other than the root directory, are considered files by MS-DOS. Each file has an entry in either the root directory or in one of these lower directories. Since these directories are considered files, they have an entry in their parent directory. All directory entries are 32 bytes in length and are formatted as follows:

0-7 Filename. Eight characters, left aligned and padded, if necessary, with blanks. The first byte of this field indicates the file status as follows:

00H The directory entry has never been used. This is used to limit the length of directory searches, for performance reasons.

2EH The entry is for a directory.

E5H The file was used, but it has been erased.

Any other character is the first character of a

filename.

8-0A Filename extension.

0B File attribute; the attribute values are as follows:

01H File is marked read-only.

02H Hidden file; the file is excluded from normal directory searches.

04H System file; the file is excluded from normal directory searches.

08H The entry contains the volume label in the first 11 bytes. No other usable information is present (except the date and time of creation) and the entry exists only in the root directory.

10H The entry defines a sub-directory and is excluded from normal directory searches.

20H Archive bit; the bit is set to "on" whenever the file has been written to and closed.

0C-15 Reserved.

16-17 The time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes.

18-19 Date the file was created or last updated. The year, month, and day are mapped into two bytes.

1A-1B The cluster number of the first cluster in the file. The cluster number is stored with the least significant byte first.

1C-1F File size in bytes. The first word of this field is the low-order part of the size.

* See diagram of MS-DOS Directory entry in Appendix A

The MS-DOS operating system maintains a File Allocation Table (FAT) on each disk. The FAT stores the cluster numbers allocated to files on the disk. There are usually two copies of the FAT to insure against loss or corruption. If this table cannot be read, the files stored on the disk cannot be located. The FAT is an array of 12-bit entries for each cluster on the disk. The first two FAT entries usually contain a disk identifier (the size and format of the disk). The second and third bytes of the FAT are always FFH. This insures that the first FAT cluster entry begins on a byte boundary. Three bytes store two FAT entries. The third FAT entry, which starts at byte four, begins the mapping of the data area. Files in the data area are not always written sequentially on the disk. The data area is allocated one cluster at a time and the first

free cluster found is the next to be allocated.

The first cluster number of a file is contained in the file's directory entry. When a cluster becomes filled, MS-DOS searches the FAT table starting from the top of the table, for an entry of 000H indicating an empty cluster. It places this cluster number into the FAT location of the cluster pointed to by the directory entry. Whenever a new cluster is assigned to a file its number is placed into the FAT location of the previous cluster. This links a file from cluster to cluster. The FAT entry for the last cluster of a file is set to a value in the range FF8H to FFFH. Any clusters which have diskette defects should have their FAT entries set to FF7H.

The File Allocation Table is found on the first section of the disk after the reserved sectors. If it is larger than one sector, the sectors it resides on are contiguous. The FAT is read into one of the MS-DOS buffers whenever needed. This can be during reads, writes, opens, or other I/O operations.

MS-DOS also keeps a File Control Block (FCB) (see Figure 3-1) to hold information about the file which is

needed by the operating system during I/O operations. A FCB consists of a 37 or 44 byte area in memory located wherever the program finds convenient. The FCB contains fields which hold the drive number, filename, extension, current block, record size, file size, date of last write, time of last write, the current record and the relative record. The current record field points to one of the records in the current block and must be set before doing a sequential read or write to the file. The relative record (from block zero) points to the currently selected record. It must be set before doing a random read or write on the file. FCBs are considered either opened or unopened. An unopened FCB must be set up before an open system function is executed.

MS-DOS File Control Block

Byte	Field
0	Drive Number
1-8	Filename
9-11	Filename Extension
12-13	Current Block Number
14-15	Logical Record Size
16-19	File Size
20-21	Date of Creation or Last Update
22-23	Time of Creation or Last Update
24-31	Reserved by MS-DOS
32	Record within Current Block
33-36	Random Access Record Number

For Extended File Control Blocks
add the following prefix:

-7	Flag Byte
-6- -2	Reserved
-1	File Attribute

Figure 3-1

To access a disk using the MS-DOS file system, a pathname, file name and file handle need to be specified. The file handle is returned when a file is created or opened. It must be passed to all file access routines. The file system routines called with the file

handle first set up or retrieve the FCB information. MS-DOS looks at a file as an address space. The file handle is the file system's method of accessing this space.

3.2 I/O SYSTEM CALLS

MS-DOS system calls are executed from macro level code. The next layer is a KOALA layer which allocates and sets up I/O buffers and makes the calls to the macro routines. The top layer of code is the level through which the application program accesses the file system. (See Figure 3-2).

MS-DOS Module Hierarchy

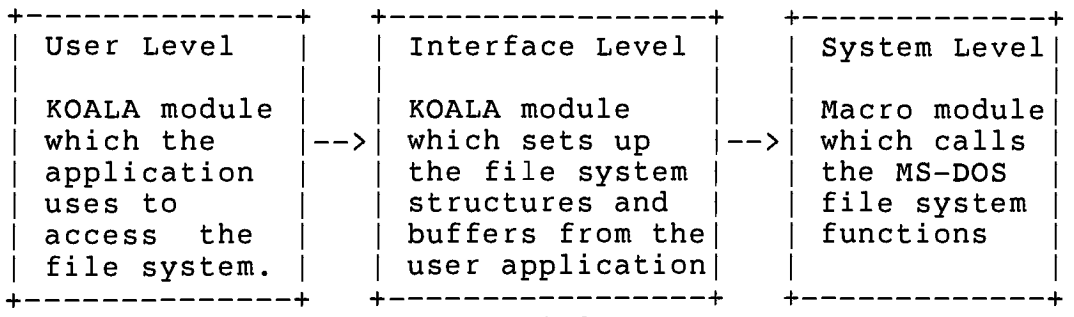


Figure 3-2

The file information is held in a record data structure. This file record holds the name of the file, the file handle, several buffer areas, the organization of the file, the access mode of the file, a count field and the file attribute. (See Appendix A)

3.2.1 CREATE

To create a file that does not currently exist, the CREATE set NAMED string DEFAULT_NAME string command is used. The set in the call is a user defined structure. The string values are valid MS-DOS file names. This routine makes a call to setup_file which allocates space for the file on the storage media and checks that the name passed in by the user is valid. It then calls the sys$create function with the file data structure record as a parameter. This function calls the macro create function to create the file with the file name, file attribute and the file handle as parameters. The MS-DOS system create is now executed. The create system call creates a new file or if a file already exists with the same name, create truncates it to zero. The new file is given the attribute and name that were passed into it. The file handle is returned, and provides a reference to the new file and the file is opened for read/write access.

3.2.2 OPEN

To open a file, the application uses the OPEN set NAMED string DEFAULT_NAME string statement. The set is a user defined data structure that must be previously defined in a VARIABLE declaration. The string values are valid MS-DOS file names. This routine calls

setup_file which allocates space for the file on the storage media. It checks that the name passed in by the user is a valid MS-DOS file name by a call to call_parse. A call is then made to sys$open with the file data structure record as the parameter. This function then calls the macro open with the file name, file access mode and file handle as parameters. The MS-DOS system open is executed from the macro function. Open associates a file handle with the file. This file handle must be used for subsequent I/O to the file. The read/write pointer is set at the first byte of the file and the record size of the file is set to one byte.

3.2.3 WRITE

The WRITE set statement is used to write to a file. This statement checks that the file is open. A call is made to the sys$put function with the file data structure as a parameter. This function then calls the macro write system call with the file handle, the number of bytes to be written, the buffer to write from and the number of bytes actually written, as parameters. The MS-DOS system write is executed from this macro function. Write transfers the number of bytes requested from the buffer into the file indicated by the file handle. The number of bytes transferred is passed back to the sys$put function and an error occurs if this

count is not equal to the number of bytes requested.

3.2.4 READ

A read is done with the READ set statement. A check is made that the file is open. A buffer is allocated to hold the data that is to be read. Sys$get is called with the file data structure record as a parameter. This function calls the macro read function with the file handle, the number of bytes to read, the buffer to read from and a variable to hold the number of bytes actually read as parameters. The MS-DOS system read is executed from the macro read function. Read transfers the number of bytes requested from the file referenced by the file handle, into the buffer location indicated. The number of bytes actually transferred is passed back to the sys$get routine and if it is not equal to the number of bytes requested an error has occurred.

3.3 STRUCTURING OF THE FILE SYSTEM

The MS-DOS system calls are the basis for the structuring of the file system. The actual code to structure the file system is done in a high level language. These programs are linked to the macro code that contains the calls to the MS-DOS system functions.

The macro module interfaces directly with the MS-DOS system calls. It sets up the necessary registers and calls MS-DOS to perform file system functions. The next module level is a KOALA module which allocates and sets up I/O buffers and calls the macro routines. Many of the file system changes are found in this KOALA module. The top level module is also written in KOALA. This module is the user interface to the file system. Changes are also made here in order to implement the record structuring extensions.

This section describes the extensions to the MS-DOS file system which provide record structure. These extensions describe the actual program code changes. The first change is to the file data structure record. (See Figure 3-3). Several fields which describe the file are added to this record. A field describes the record structure of the file as either fixed, variable, or stream. A stream buffer field is used to read and write to stream files and a stream index is used to hold the position in the stream file.

```
!*** MS-DOS File record

EXPORT TYPE file_access IS ACCESS
    RECORD
   EXPORT name  : STRING;
```

```
EXPORT ifi   : INTEGER;
EXPORT buffer  : char_access;
EXPORT user_buff_len : INTEGER;
EXPORT user_buff_area : char_access;
EXPORT buff_size : INTEGER;
EXPORT organization : (org_sequential,
            org_indexed);
EXPORT access_mode : dos_access_type;
EXPORT attribute : dos_attribute;
EXPORT current_key : INTEGER;
EXPORT count   : INTEGER;
EXPORT dos_buffer : dos_buffer_type;
        EXPORT at_end   : BOOLEAN;
EXPORT is_open   : BOOLEAN;
EXPORT rec_format : range 0 .. 255 ;
EXPORT stream_buffer     : char_access ;
EXPORT stream_index      : INTEGER ;
EXPORT stream_buf_size   : INTEGER ;
 END RECORD;
```

Figure 3-3

3.3.1 FIXED LENGTH RECORDS

A write or read of a fixed length record file should always return the same number of bytes. The field in the file data structure record describing the type of record should be set to fixed in this case. The

compiler sets this value by looking at the user defined set in the create or open call. If this set is integer or a record consisting of fixed length fields the record format is fixed. The compiler also determines the size of this user structure. This value is placed into the buffer length field of the file data structure record during a create and remains constant throughout operations on the file.

The following KOALA program is used to illustrate the activities of the MS-DOS file system. The file record (see Figure 3-3) is also referenced.

```
PROGRAM fixed IS

VARIABLE stock : STRING ;
VARIABLE itemcost : STRING ;
VARIABLE numavail : STRING ;

VARIABLE stock_rec : SET OF RECORD

   stock_num : INTEGER ;
   cost_per_item : INTEGER ;
   num_available : INTEGER ;

END RECORD ;

!Screen prompts for data

CONSTANT request : STRING := "Please type the information
```

```
                        after each prompt " ;
    stock_num : STRING := "Stock Number : " ;
    cost : STRING := "Cost per Item : " ;
    avail : STRING := "Number Available in Stock : " ;

STATEMENT add_stock_rec IS

  BEGIN
   ERASE_SCREEN ;
   WRITE_LINE request ;
   WRITE stock_num ;
   READ_LINE stock ;
   stock_rec.stock_num := INTEGER stock;
   WRITE cost ;
   READ_LINE itemcost ;
   stock_rec.cost_per_item := INTEGER itemcost ;
   WRITE avail ;
   READ_LINE numavail ;
   stock_rec.num_available := INTEGER numavail ;
  END add_stock_rec ;

STATEMENT create_file IS

  BEGIN
   ALLOCATE stock_rec ;
   CREATE stock_rec NAMED "create.dat" ;
  END create_file ;

STATEMENT write_file IS
```

```
  BEGIN
   create_file ;
   FOR i IN 1 .. 5 LOOP

     add_stock_rec ;
     WRITE stock_rec ;

   END LOOP ;
   CLOSE stock_rec ;
  END write_file ;


STATEMENT read_file IS

  BEGIN
   OPEN stock_rec NAMED "create.dat" ;
   ERASE_SCREEN ;
   FOR i IN 1 .. 5 LOOP
     READ stock_rec ;

     WRITE stock_num ;
     WRITE_LINE STRING stock_rec.stock_num ;
     WRITE cost ;
     WRITE_LINE STRING stock_rec.cost_per_item ;
     WRITE avail ;
     WRITE_LINE STRING stock_rec.num_available ;

     END LOOP ;
   CLOSE stock_rec ;
```

```
    END read_file ;

BEGIN

    write_file ;
    read_file ;

END fixed ;
```

Figure 3-4

In the above program, stock_rec is a set which contains records consisting of three integer fields. The record format of the file is set to fixed and the record size is six bytes. An integer is stored as two bytes in KOALA.

The WRITE set statement puts the data that the user assigned to the file record and the length of that record into the file data structure record fields. The data is put into the buffer field of the file record and the record size is put into the user_buff_len field. The sys$put routine is then called with the file data structure record and the record format, which is fixed, as parameters. This routine passes the file handle, the number of bytes to write (which is equal to the user_buff_len), and the file buffer field (which

contains the data), to the macro write function. The number of bytes to write and the number of bytes written should always be the same unless an error has occurred. The data is written to the file as a sequence of bytes with no delimiters.

If the user enters 1, 2, and 3 respectively in response to the prompts in the programming example, (see Figure 3-4) the call to WRITE stock_rec illustrates what occurs in the file system. The data is placed in the buffer field of the file record in hexadecimal as 310032003300. This represents two bytes per integer. The user_buff_len field is given the value 6. The sys$put routine is called, which calls the macro write function, and the six bytes are written to the file.

The READ set statement allocates a user buffer into which the data is returned to the user. This buffer is the size of one record. This size is determined by the user defined set and remains the same during reads and writes on the file. The size of this buffer is found in the user_buff_len field of the file record. The sys$get function is called with the file data structure record and the fixed record type as parameters. Sys$get sets the record size to the user_buff_len field of the file record. The macro read is called with the file handle,

the record size and the file buffer as input. The system read is executed which returns the same number of bytes each time unless an error occurs. The number of bytes read from the file is returned from the MS-DOS system call. This value is checked against the record size and an error is returned if they are not equal.

When READ stock_rec is called, the data buffer is allocated with a record size of six bytes. The sys$get function is called, which executes the macro read function. The user data buffer is returned containing the hexadecimal values of the integers, 310032003300.

There were very few problems implementing fixed length records on the MS-DOS file system. The major areas of concern were in setting up the buffers and determining proper space allocations. The length of the record determined the buffer size and this needed to be allocated before the first write. The size of the records had to be retrieved and placed into the file record where it could be easily accessed. The length of the record was determined at compile time and returned through a KOALA data structure record.

3.3.2 VARIABLE LENGTH RECORDS

The size of a variable length record varies with the data written to it and read from it. Variable length records are written with the length placed into the first two bytes of the record. This length is used to read the record from the file. A variable length record file consists of a set of strings.

```
PROGRAM varia IS

CONSTANT request : STRING := "TYPE STRING : " ;
VARIABLE varfile_line : STRING ;

VARIABLE varfile : SET OF STRING ;

STATEMENT add_to_file IS

  BEGIN
   ERASE_SCREEN ;
   WRITE request ;
   READ varfile_line ;
  END add_to_file ;

STATEMENT create_file IS

  BEGIN
   SET_PARAMETER varfile, VARIABLE_REC_FORMAT TO counted_recs;
   CREATE varfile NAMED "test.dat" ;
   ALLOCATE varfile ;
  END create_file ;

STATEMENT write_file IS
```

```
  BEGIN
   create_file ;
   FOR i IN 1 .. 5 LOOP

     add_to_file ;
          varfile.all := varfile_line;
     WRITE varfile ;

   END LOOP ;
   CLOSE varfile ;
  END write_file ;

STATEMENT read_file IS

  BEGIN

   SET_PARAMETER varfile, VARIABLE_REC_FORMAT TO counted_recs;
   OPEN varfile NAMED "test.dat" ;
   ERASE_SCREEN ;
   FOR i IN 1 .. 5 LOOP
     READ varfile ;
   WRITE_LINE varfile.all ;
     END LOOP ;
   CLOSE varfile ;

   END read_file ;

BEGIN
```

```
   write_file ;
   read_file ;

END varia ;
```

Figure 3-5

The create sets the record format of the file to variable. The SET_PARAMETER statement sets the variable length record format to counted records. This indicates that the records in the file are preceded by a two byte length field. The open also sets this field to variable. There are no further changes in the create function. The open calls sys$open with the file data structure record and the variable record format as parameters. Sys$open allocates a default buffer of 256 bytes to be used by other I/O operations. The default record size is set to 256. The macro open is then called and the file is opened. The is_open field of the file record is set to true.

Strings are preceded by a two byte string length in KOALA. This length is used to read and write the records. The WRITE set statement takes the data from the user input for the record and then converts it into a string. The data is now two length bytes followed by the actual data bytes. The buffer length is set to the

length of the data plus two bytes for the length of the record. Sys$put is now called with the file data structure record and the variable record type as parameters. The sys$put routine takes the buffer length as the number of bytes to write and checks that the record starts on a word boundary. The macro write routine is now called with the file handle, the user_buff_len and the file record's user buffer, as parameters. This writes the actual data to the file.

If the user entered 123 as the input string in the above program, the WRITE varfile call causes the following steps to occur. The file buffer converts the data into string format. This causes an integer 3 to precede the data. The user_buff_len is the length of the string plus the two bytes for the length of the record. This assigns 5 to the user_buff_len field of the file record. The sys$put function is called which calls the macro write function. The write call writes five bytes to the file. The length bytes are written to the file followed by the data 123. A dump of the file shows the record written in hexadecimal as 0300313233. The first two bytes represent the record length of 3 and the following three bytes represent the user data 123.

The READ set statement calls the sys$get function to read a record from the file. Sys$get first sets the number of bytes to be read to 2 and calls the macro read. The MS-DOS system read is executed which reads the first two bytes. These bytes indicate the length of the data record. This value is returned to sys$get and is assigned as the number of bytes of data to be transferred. A buffer is allocated to hold the returned data in the case that the default buffer that was allocated in the open is too small. The number of bytes to be read is also made even since all records must start on an even boundary. The macro read is called again and the read system call transfers the data from the file. The user buffer is then checked to make sure it has enough space for the data. The data is then placed into the user buffer and made available to the user. If there is insufficient space in the user data buffer, more space is allocated.

The READ varfile statement is executed by the program. (See Figure 3-5). This causes the sys$get function to be called with both the file record and variable record format as parameters. The macro read is called first to read two bytes from the file. It returns with the integer 3. This value is then passed back to the macro read function as the number of bytes

of data in the record. Three bytes are read from the file and the users data buffer now contains the record. The hexadecimal value of this record is the string 0300313233.

There were several problems while implementing variable length records on MS-DOS. The major problems are with the allocation of buffers of the right size on both the read and writes of records. The number of bytes written to the file is always two more than the number of bytes of data. This is the number that is stored in the user_buff_len field of the file record. This number has to be recognized as different from the actual number of data bytes. The count of the actual data bytes only is stored as the first two bytes of the data string. The read has to get the first two bytes of data from the file in order to determine the amount of data. If the read used the user_buff_len that was kept in the file record, two bytes too many would be read and the macro routine returns an end of file error. Along with this the write to the file has to be checked to make sure that the first two bytes only represent the number of bytes of actual data and do not include the two bytes for the length. The write buffer has to be allocated to the size of the data plus the two bytes for the length field. If this buffer is only allocated to

the size of the data, then, when the record is written to the file the last two bytes are truncated. An error occurs if you passed the macro write function the correct size of the data plus the length, and has a buffer of the wrong size.

3.3.3 STREAM I/O

In order to implement stream I/O, the record format field of the file is set to stream in the create and open. This is done by using the SET_PARAMETER statement before the create and open calls. Stream files are treated as variable length record files in their allocation of buffers. The default buffer of 256 bytes is allocated in the open and the default record size is set to 256 bytes.

```
PROGRAM stream IS

VARIABLE
    stream_file : SET OF STRING ;

BEGIN

    SET_PARAMETER stream_file, VARIABLE_REC_FORMAT to stream_recs;
    CREATE stream_file NAMED "stream.txt";
    allocate file;

    FOR i IN 1..10 LOOP
```

```
    stream_file.all := "Hello " & STRING i;
    WRITE stream_file;
     END LOOP;

     CLOSE stream_file;

     OPEN stream_file NAMED "stream.txt";
     FOR i IN 1 .. 10 LOOP
    READ stream_file ;
    WRITE_LINE stream_file.all ;
     END LOOP;

     CLOSE stream_file;

END stream ;
```

Figure 3-6

The WRITE set statement converts the data to a string and sets the buffer length to the length of the data plus two for the delimiter bytes. The sys$put function is then called with the file data structure record and the record format as parameters. To write the record the length bytes are taken off the string. The macro write and the MS-DOS write system call are used to write the data portion of the string only to the file. Sys$put then places a two into the number of bytes of data field in the write call and calls the macro write and the MS-DOS system write. A carriage

return character and line feed character is written to the file. This delimits the record.

The first time WRITE stream_file is called in the above loop (see Figure 3-6), Hello 1 is converted to a KOALA string. It is stored as 070048656C6C6F2031 in hexadecimal form. This represents the length of the data which is the first two bytes, followed by the data bytes. The user_buff_len is set to 9, which is the length of the data plus the two bytes which are used to write the delimiters. Sys$put is then called. The macro write function is called with the file handle, the number of bytes to be written and the user buffer which contains the data. The number of bytes to be written is the number of bytes of actual data. This is found by subtracting two from the user_buff_len. The data is then written to the file. The macro write function is called again to write the two byte delimiter to the file. The file now contains the following record in hexadecimal format: 48656C6C6F20310D0A. Notice that the data is not preceded by a length field and that the data is followed by the carriage return character and the line feed character.

The READ set statement calls the sys$get function. Sys$get reads a block of 256 bytes and then searches

these bytes for the carriage return and line feed characters indicating the end of the record. A stream index value is kept in the stream index field of the file data structure record. This index is used to keep track of the place that the last record left off and to determine if another block of data needs to be read in. The data is returned to the user with the carriage return and line feed characters stripped off. The returned data is preceded by the length field.

The READ stream_file is executed by the program. This causes the sys$get function to be called. A block of data is read from the file. This block is 256 bytes if the file is sufficiently large or the amount of data bytes that are left in the file. This data is read into the stream_buffer field of the file record. It is then searched for the carriage return and line feed delimiter. The data is written to the user buffer one byte at a time until the carriage return, followed by a line feed is found. A count of the number of bytes written to the user buffer is kept. If no carriage return and line feed is found, either another block of the file is read and the search continues, or, if no more data exists, an end of file error is returned. On a successful read, the data is returned to the user as a string. The data returned in this case, represented as

a hexadecimal string, is 070048656C6C6F2031.

There were several problems with the implementation of stream records. The first is with the size of the buffers. It worked out that the allocation of buffers is the same as that used for variable length records since the two byte delimiter cancels out with the two byte length field. The user buffers and the user_buff_len fields remain the same. There is a need for a new buffer and a new length field. These have to be added to the file record. This is to solve the problem of how to read the data. No field indicating the length of the data in the file exists and this information is not found in the file. The block size of 256 bytes is chosen as the stream buffer size and the data present in the file is read in 256 byte blocks. The check that the number of bytes read matched the number of bytes requested has to be removed. Otherwise, when less than 256 bytes remained in the file, end of file is returned and the read fails. Once this is fixed the byte compares are performed. Here care has to be taken to keep an accurate count of the data that is copied to the file buffer and of the amount of data left in the stream buffer. The compare has to look for both a carriage return and a line feed since they can appear singularly in the data. Another problem occurs when the

carriage return and line feed bytes straddle the block boundary. Care has to be taken to remember that the carriage return character had been found before reading a new block and checking for the line feed character.

3.4 CONCLUSIONS

The MS-DOS file system user now has the choice of stream I/O, fixed length record I/O, or variable length record I/O. Stream I/O is written to the file as a sequence of bytes delimited by a carriage return character and a line feed character. Stream files are read by searching for the carriage return and line feed characters. The data bytes are saved in the read buffer and the delimiters are stripped off. Fixed length record format files always write the same number of bytes. This number is determined by the size of the user record at compile time. Variable length record files write the user data to the file as KOALA strings. These strings are preceded by a two byte length. The length field is read during a read. The length value determines the number of bytes that are retrieved from the file.

CHAPTER 4

LESSONS LEARNED

The UNIX native file system and the MS-DOS native file system both are unstructured. Data is written to the native file as a series of bytes. Both file systems allow the user to provide more structure. In the case of large amounts of data being written to a file, as in a database application, the user can provide a record structure for the file system.

The UNIX file system can be structured to provide fixed length record I/O and variable length record I/O to the user. Stream I/O is available when implementing ascii text files under the native UNIX file system. The high level language I/O calls are modified to provide fixed and variable length record I/O.

The MS-DOS file system contains a series of system calls that the user executes to implement the file system. These system calls are written into a macro module to perform the MS-DOS I/O operations for the file system. The macro routines are called from a high level language program and it is at this level that the record structure is implemented. Stream I/O, fixed length record I/O and variable length record I/O files are now available to the user.

4.2 FUTURE ENHANCEMENTS

4.2.1 Indexed File System

In addition to the variable length record I/O, fixed length record I/O and stream I/O that have been discussed for the UNIX file system and for the MS-DOS file system, a user may desire additional structure. This structure may be provided in the form of indexed record I/O.

In an indexed sequential file, each record contains a key value. This key value may be used for direct or sequential access to the record. A record in the indexed file is variable length or fixed length, with a unique key. Retrieval of records in an indexed sequential file is either successive by order of key

value or direct given a specific key value. This file structuring method can be implemented on the UNIX and the MS-DOS file systems. Recommendations for adding indexed records to the UNIX file system and the MS-DOS file system now follow.

An indexed file is implemented by building an index on top of a sequential data file. Each record holds a key field that is defined by its location and length. A unique primary key is the least key required to identify each record. The records are stored sequentially by key value order. A record is retrieved either sequentially from the file or randomly by the key value. A key is represented by a string, a character or an integer. To determine the size of a string key the string length function that was previously written to determine the size of a variable length record is used. The sizes of the other data types are determined by the operating system that is being used. An integer is usually represented by 2 bytes and a character by one byte. Several changes to the UNIX and the MS-DOS file systems have to be implemented. A discussion of the changes to the current I/O routines in order to implement indexed sequential I/O follows.

4.2.1.1 Data Structures - UNIX

The current UNIX data structures require some changes and additions in order to implement an indexed file organization. The inode file type field that was previously added now recognizes indexed as a file type. A field is added to the inode that identifies the current key. This field indicates whether a primary or a secondary key is being used. A second field is added to the inode structure which points to a key structure. This key structure holds the information on the keys for each record. The first field of the key structure indicates the number of keys in the record. The second field is an array of structures describing the keys. These key descriptor structures contain: a field describing the type of the key, a flag field indicating whether or not the key value can change, whether duplicate keys are allowed or if the key is null, a field indicating the position of the key in the record, and a field specifying the size of the key.

4.2.1.2 Data Structures - MS-DOS

The data structures for the MS-DOS file system require some changes in order to recognize that indexed I/O is the current file organization. A field is added

to the MS-DOS file record to identify the current key. The organization field of this record must be changed to recognize that the file organization may be indexed. The set descriptor record holds a pointer to a key descriptor. The key descriptor is a data structure record that describes the keys for each record. It contains a field indicating the number of keys in the record and a list of data structure records describing the keys. The key data structure record contains: a field describing the data type of the key, a flag field which indicates whether or not the key value can change, whether duplicate keys are allowed or if the key is null, a position field indicating where the key is found in the record, and a size field indicating the length of the key. These data structures are illustrated in Appendix A. This ends the alterations and additions to the MS-DOS data structures. The I/O functions and how they change to accommodate indexed records are described next.

4.2.1.3 Create

The create function now looks at the organization of the file. An indexed file builds the list of key descriptors, one for each record, during the create. The create call from the application does not change.

4.2.1.4 Build Key List

The build key list function is added to the UNIX and the MS-DOS file systems to build the key descriptors. This function fills in a key descriptor for each key of the record and places them in the key descriptor array. The information for these key descriptors is taken from the user declaration of the file. These key descriptor records are used to build the index for the file.

4.2.1.5 Open

The open function also now checks to see if the file organization is indexed. When the file is indexed, the list of key descriptors is built the same as on a create. The key descriptors returned by the open are checked against those the user indicated when defining the file. An error is returned if they do not match. The open call at the application level does not change.

4.2.1.6 Close

The close function requires no changes in order to implement indexed I/O.

4.2.1.7 Indexing Methods

4.2.1.7.1 B Tree

Before the write and the read functions for indexed files are discussed, the B tree and the B+ tree indexing methods are described. Prior to the time of the first write to the indexed file, an index must be created to hold its key values. This index becomes the path to follow when a random read or write of a record is requested. An indexed file is first loaded sequentially by order of the primary key value. The record is written to the data block and the key value is placed into the index. The B tree and the B+ tree indexing methods demonstrate how data is retrieved from a file or inserted into a file using keyed records.

The B tree is an indexing method based on a tree data structure. The B tree is one of the most widely used methods for organizing an indexed file and its data. It is a standard organization for indexes in a database system.

The standard definition of a B tree states that a B tree of order M is an M-way search tree that provides

the following properties (see 18, Knuth):

- Every node has <= M children.
- Every node of the tree, except the root and the leaves, has >= M/2 children.
- The root has at least two children unless it is a leaf.
- All leaves of the tree are found on the same level.
- An internal node with x children holds x-1 key values.

The operations that the B tree performs are direct search, sequential search, insertion of key values and deletion of key values. Each is now described.

A direct search on a B tree for a specific key value starts at the root node. This node is searched, and if the key is not found, the traced search path leads to a node in the next level. This node is then searched; the search continues to the next level (as long as one exists), until the key is found or a leaf node is reached.

Sequential access to the key values in a B tree is done by traversing the tree, node by node, in the same order that the data was entered into the tree. Nodes containing more than one key are visited more than once. The child node must be searched before the next key of the node can be accessed. This requires up and down and left to right searching of the tree and can be very time consuming. Sequential searches on a B tree do not provide for very impressive system performance.

To insert a key value into the B tree, the tree must be searched as in the direct search method described above. The search does not succeed and will end at a leaf node. All keys are inserted at the leaf node level. If this leaf node is full, then the node is split. The original node and the new key are divided into two new nodes. A pointer to the new node is inserted into the level above. If the node above becomes full, it too must be split. This can propagate up to the root node. When the root node needs to be split, a new root node is created and the tree increases by one level.

Deletion of a key from a B tree involves a direct search of the tree for the key. When the key is found it is removed from the node. If the node is not a leaf

node the next sequential key must take its place in the node. This is to insure that the paths between nodes do not change.

4.2.1.7.2 B+ Tree

The B+ tree is a variation of a B tree which allows for faster sequential searching. The leaf nodes of the B+ tree contain a sequential list of keys. A B tree path leads to this sequential list. Sequential processing becomes quick and easy since the keys are linked together sequentially in the leaf nodes.

A direct search in a B+ tree uses the B tree to follow a path through the levels to a leaf node. When this leaf node is reached, it is searched for the key. The same number of nodes are read on each search. This does not aversely effect performance and provides a consistency when searching.

To delete a key from a B+ tree, the B tree path nodes need not change. The key is deleted from the leaf node; the index portion does not change.

Both the B tree and the B+ tree provide an indexing method for organized searching of keys in a file. Next, the B+ tree implementation on UNIX and MS-DOS is discussed.

4.2.1.8 Write

The write function first sets the file access by key mode. Records are written to the file in sequential order according to ascending primary key values. Once the record is written to a data block, the key value is placed into the index. The key descriptor is inserted into the index by performing a direct search on the B tree and placing the node in the proper place in the sequential list of leaf nodes. When the file is created and the initial data is loaded into the file, space is saved in both the B+ tree index and in the data blocks to allow for additional insertions. Should a data block become full, a data block split occurs. In this case, half of the data stays in the data block and the other half goes into the new data block. This splitting of data blocks slows system performance. The number of accesses to retrieve a record increases and the system takes up resources and time to handle the splitting operation. Because of this, splitting of data blocks should be avoided whenever possible. This can be done

by planning out the amount of data that the file needs and by preparing for additional data insertions at the initial data load time. The write call at the application level does not change.

4.2.1.9 Read

Two read functions are supplied. This is to provide for random reads and for sequential reads of records. The sequential read takes a key of reference as input. This key of reference indicates whether the primary key index or an alternate key index is to be used as the basis for record retrieval. The selected key index is then used to retrieve one record for each read request. This is done by searching the sequential list of leaf nodes in the B+ tree index. This list makes the sequential read fast and efficient. The read call for a sequential read at the application level does not change. The system uses the primary key index unless a SETUP_KEY with the key number as a parameter is called. This new function is supplied to go into the key descriptor array and chose another key descriptor as the key of reference.

The random read function requires the key value and the key of reference as parameters. The index selected

by the key of reference is searched for the specified key value. This requires a direct search of the B+ tree index. The B tree provides the path to the leaf node where the key descriptor for the desired record resides. When this key value is found, the record is returned to the user. The format for this function call on UNIX is: ranread(fildes, filbuf, readnum, keynum, match). The keynum parameter is the number of the key to be used. The match parameter indicates the item in the file to retrieve. The format for this function call on MS-DOS is: READ set KEY integer MATCH match_type. The set parameter is the name that was declared in a VARIABLE declaration for the file. The integer value indicates the key number to be used. The match_type indicates the item in the set to read. KEY_EQ causes the first item that matches the indicated key value to be read. KEY_GT causes the first item greater than the indicated key value to be read and KEY_GE causes the first item that is greater than or equal to the indicated key value to be read.

4.2.1.10 Remove

The remove statement is added to the file functions available under indexed I/O. There are two different calls to remove a record from the file. The following

statements are added to the UNIX file system. The remove(fildes, keynum) function will search the file corresponding to the fildes parameter. The key value equal to that found in the key descriptor indicated by the keynum parameter is searched for. The remove(filedes) function removes the record that the file is currently pointing at. The following statements are added to the MS-DOS file system to provide for deletion of records. The REMOVE set KEY integer statement searches the file for the key value equal to that found in the key descriptor referenced by the key parameter. If the record is found, it is deleted from the set. The REMOVE set statement deletes the record that is currently being pointed to in the indicated set.

4.2.2 Conclusions

An indexed file provides the user with sequential retrieval of records and direct record retrieval by key value. This method is desirable when large amounts of data need to be stored by a user and a field in the record is identified as the primary key for random retrieval. Many database systems use this file format to store personnel or supply records. The addition of the indexed sequential file to the UNIX file system and the MS-DOS file system provides the user with more options when using the file system.

4.3 NET.UNIX OPINIONS ON UNIX FILE SYSTEM

A discussion was started on the USENET group net.unix from article <3287@decwrl.UUCP>["unix file system"],by jcampbell@mrfort.DEC, which involved using the UNIX file system for FORTRAN. He stated that there is no place in UNIX to keep information about the contents of the file. This posting elicited numerous responses from the net.unix community during July and August of 1985. The responses agreed that the structuring is necessary in this case however, the UNIX operating system is not the medium for the addition of this kind of structure.

There were several arguments for this. Operating systems with several types of file support are large and their source listings difficult to follow. Along with the expansion of the operating system to support additional file system capabilities goes the dedication of system resources for possibly unused operations. The utility programs involved with copying files would all have to be changed and several other aspects of the kernel would need to be modified as stated in the section on UNIX above. System maintenance would become more complicated and the chance of file corruption would become greater. The general opinion of the community was to keep unnecessary code out of the operating

system. Some felt the place to keep information on the file was in the file itself. Any support that is not currently supplied by the kernel should be tailored to the application or if a standardized package is needed, as in the case of database support, building a set of library routines and maintenance programs is recommended.

APPENDIX A

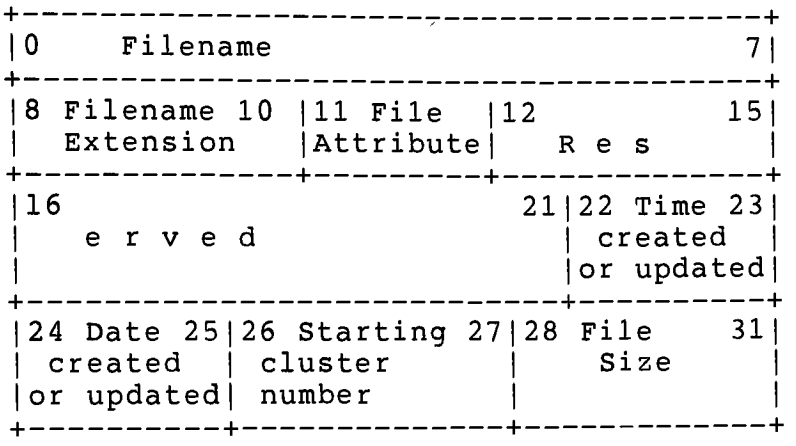
DATA STRUCTURES AND GLOSSARY

A.1 MS-DOS DATA STRUCTURES

```
      !*** File record

EXPORT TYPE file_access IS ACCESS
    RECORD
   EXPORT name   : STRING;
   EXPORT ifi    : INTEGER;
   EXPORT buffer   : char_access;
   EXPORT user_buff_len : INTEGER;
   EXPORT user_buff_area : char_access;
   EXPORT buff_size : INTEGER;
   EXPORT organization : (org_sequential,
              org_indexed);
   EXPORT access_mode : dos_access_type;
   EXPORT attribute : dos_attribute;
   EXPORT current_key : INTEGER;
   EXPORT count   : INTEGER;
   EXPORT dos_buffer : dos_buffer_type;
          EXPORT at_end   : BOOLEAN;
   EXPORT is_open   : BOOLEAN;
   EXPORT rec_format : range 0 .. 255 ;
   EXPORT stream_buffer     : char_access ;
   EXPORT stream_index      : INTEGER ;
   EXPORT stream_buf_size   : INTEGER ;
    END RECORD;

!***  Access types ***

EXPORT TYPE
    dos_access_type IS
   (dos_read_access,
   dos_write_access,
   dos_update_access);
```

```
EXPORT TYPE char_buffer IS ARRAY (0..32767) OF CHARACTER;
EXPORT TYPE char_access IS ACCESS char_buffer;


!*** Attributes ****

EXPORT TYPE dos_attribute IS INTEGER;

EXPORT CONSTANT
    dos_attr_normal : INTEGER := 0;
    dos_attr_read_only : INTEGER := 1;
    dos_attr_hidden : INTEGER := 2;
    dos_attr_system : INTEGER := 4;
    dos_attr_changeable : INTEGER := 7;
    dos_attr_vol_id : INTEGER := 8;
    dos_attr_dir : INTEGER := 10;
    dos_attr_hard : INTEGER := 22;
    dos_attr_archive : INTEGER := 32;

!******** Key descriptor *****

Type key_descriptor is access
    record
    num_keys    : integer;
    keys    : array (1..255) of
                record
                key_type : (kd_stg,kd_in2,kd_bn2,kd_in4,kd_bn4);
                flags : byte;
                p_offset : integer;
                u_offset : integer;
                size : integer;
                end record;
    end record;


! *********** Set descriptor ******

Type set_descriptor is access
    record
    class : set_class;
    num_var : range 0 .. 255;
    size  : integer;
    keys  : key_descriptor;
    variants : array (0 .. 255) of variant_descriptor;
    end record;
```

MS-DOS Directory Entry



A.2 GLOSSARY

B tree - A B tree is the standard indexing organization, based on a tree data structure, used for databases. In a B tree more than one branch can come from each node. Each node holds primary key values, index pointers (which are used to branch to the next lower level of nodes) and associated data. The associated data is used as a set of pointers to locate the data records whose keys are in the index node.

B+ tree - A B+ tree is a variation of the B tree. It contains a B tree as an index followed by the leaves, where all the keys and associated data are found. In this structure the index guides the search to the leaf nodes where the key is found. The leaf nodes are linked together, left-to-right to allow for quick, sequential searching.

C - This is a high level programming language designed by Dennis Ritchie. The UNIX operating system has been largely rewritten from assembly language into C.

close - This is a UNIX system command that frees all structures built with the open or creat commands. (i.e. inode table entries, file table entries, etc.)

close a file handle (function 3EH) - This is the MS-DOS system call that closes the file associated with the file handle and flushes the internal buffers.

cluster - A cluster is a way of organizing the data storage area of a disk under the MS-DOS operating system. Each cluster consists of a number of logical records. The default size of a logical record is 128 bytes.

creat - A UNIX system command that creates a new inode, makes a directory entry that contains the name of the file and the i-number, then does an open.

create a file (function 3CH) - This MS-DOS system call will create a new file or truncate an old file to zero length. If the file did not exist it is created and a directory entry is made. The file handle is returned and the file is opened for reading and writing.

directory - A UNIX directory is a type of file. It is used the same as ordinary files except that the system controls its contents. Unprivileged programs or users may not write on it. Directories place a tree or hierarchial structure on the file system, providing a mapping between the file name and the actual file. A MS-DOS directory is also a type of file. Its contents are other directories or files. Directory entries hold information on files such as the name, attributes and size.

file - A file in UNIX and MS-DOS is an unstructured sequence of bytes. The system imposes no structure to the files, any structuring is done by the user. Files are attached to the directory hierarchy.

file allocation table - This MS-DOS operating system table stores the cluster numbers associated with the files on the disk.

file control block - A file control block is a way of communicating information on the file (such as the name, drive, etc.) to the MS-DOS operating system in order to open a file, find its directory, or access its data records.

file handle - A file handle is an internal file identifier for the MS-DOS operating system. It refers to a file on a storage media. More than one file handle may point to the same file.

inode - The inode is found in the system's ilist and holds the file description. This description is made up of the user and group ID of the owner, the file protection bits, the physical disk or tape addresses for the contents of the file, its size, the time of creation, last use and last modification of the file, the number of links to the file, and a code indicating the type of file. An inode is allocated each time a new file is created.

i-number - The i-number is the offset of an inode within the ilist. The i-number along with the device and read/write pointer are stored in a system table and serve to uniquely identify that file.

ilist - The ilist is the system table of which the i-number is an index and where the inode is stored. The ilist is one of the four regions that the file system breaks the disk down into, and is a list of file definitions.

I/O device - An I/O device is a device used for input and/or output. Such as tape or disk.

kernel - The UNIX kernel is the operating system. It is made up of around 10,000 lines of C code and around 1,000 of assembly code. The code in the kernel is based on simplicity rather than efficiency.

leaf nodes - The leaf nodes are the terminal nodes in a tree structure. In the case of B+ trees the leaf nodes hold the key values and associated data.

link - A link is a directory entry for a file (in UNIX). To make a link to an existing file a directory entry is created with the new name, the i-number is copied from the original file and the link count field of the inode is incremented.

MS-DOS - Microsoft's Disk Operating System, a disk operating system for microcomputers using 8086/8088 microprocessors.

open - Open is a system command that takes the file system path name and converts it to an inode table entry. A pointer to this table entry is put into the newly created open file table entry and a pointer from this file table entry is put into the system data segment for the process.

open a file (function 3DH) - This MS-DOS system call takes a given path name and access mode and associates a 16 bit file handle with the file. The read/write pointer is set to the first byte of the file and the record size of the file is 1 byte. The returned file handle is needed for any future I/O on the file.

path - A path is the sequence followed to get from the root, through the subdirectories, to a file. The path name consists of directory names separated by "/"

(slashes on UNIX) and "\" (backslashes on MS-DOS) until the file is reached.

read - Read is a system I/O command that transmits a specified number of bytes between the file and the buffer. Read accesses the file through the inode. When a read returns with no data, the end of file has been reached.

read from file/device (function 3FH) - This MS-DOS system call transfers count bytes from a file into a buffer location. It takes a pointer to the buffer, the file handle and the number of bytes to read as input. When the value returned is zero the program tried to read from the end of the file.

record - A record is a division often found in files but which is completely absent from the UNIX file system. A record is a fixed sequence of bytes or a count along with count bytes, used to structure files.

remove - Remove is a UNIX system command that decrements the link count on the inode corresponding to the directory entry. It also erases the entry. Remove does not remove inodes only the links until the link count becomes zero. When this occurs, the system removes the inode and the file.

root - The root is a directory maintained by the system. All files in the system can be found by chaining through a sequence of directories starting at the root.

search tree - An M-way search tree is a tree in which each node has <= M paths from the node.

seek - Seek is a UNIX system command that does no physical seeking, it manipulates the I/O pointer.

UNIX - UNIX is an operating system created at Bell Laboratories in 1969. It is written mainly in C, is portable and based on simplicity.

write - Write is a UNIX system I/O command that transmits a specified number of bytes from a buffer to a

file. Write accesses the file through an inode. It is considered an error if the number of bytes requested for transfer is not equal to the number actually written to the file.

write to a file or device (function 40H) - This is an MS-DOS system call that transfers count bytes from a buffer into a file. It is an error if the number of bytes written is not the same as the number requested. It takes a pointer to the buffer, the number of bytes to write and the file handle as input.

BIBLIOGRAPHY

1. Bass, John L., "UNIX Device Drivers", Dr. Dobb's Journal, Dec. 1984, p38-48.

   This article provides an overview to the UNIX I/O system and to UNIX device drivers. It describes the file I/O system including the open, close, read and write functions.

2. Bayer, R. and McCreight, E., "Organization and Maintenance of Large Ordered Indexes", Acta Informatica, 1972, p173-189.

   Bayer and McCreight discuss B trees as a method for organizing and accessing indexes in large files. This paper is the first to introduce the concept of B trees and is noted as a source for almost all other research on the subject.

3. Bourne, S.R., "UNIX Timesharing System: The UNIX Shell", The Bell System Technical Journal, vol.57, no.6, July-August 1978, p1971-1990.

   This article describes the commands and capabilities of the UNIX shell. The shell is both a command language and a programming language that can be used both interactively and non-interactively through files.

4. Christian, Kaare, The UNIX Operating System, New York, NY: John Wiley and Sons, Inc., 1983.

   This book provides an overview of UNIX for all levels of users. Part one contains the history of UNIX and a guide to the most commonly used features. Part two covers the shell, utilities and the kernel.

5. Claybrook, Billy G., File Management Techniques, New York, NY: John Wiley and Sons, 1983.

   This book covers all topics on file management. These include file structures, file systems, different file organizations and I/O operations on files.

6. Comer, Douglas, "The Ubiquitous B-Tree", Computing Surveys, vol.11, no.2, June 1979, p121-137.

This article describes B trees in detail, then goes into variations on them, including the B+ tree. Finally, it concludes with a description of VSAM which is IBM's B+ tree based access method.

7. Deitel, H.M., An Introduction to Operating Systems, Reading, MA: Addison-Wesley Publishing Co., 1983, p481-501.

This book contains a case study on the UNIX operating system. It has a section on the similiarities and differences of stream and record I/O.

8. Freeman, Donald E. and Olney R. Perry, I/O Design: Data Management in Operating Systems, Rochelle Park, NJ: Hayden Book Company, Inc., 1977.

This book describes I/O design and data management. It describes ways of organizing data, including the sequential organization and the indexed organization. It also discusses opening, closing, and accessing files.

9. Held, G. and Stonebraker, M., "B-trees Re-examined", CACM, vol.21, no.2, February 1978, p139-143.

This paper briefly describes B-trees as a storage structure for multi-user database applications. It then goes into some of the problems that can occur when using this method. The problems described include secondary indexing, concurrency and locking, and directory height.

10. Holt, R.C., Concurrent Euclid, The UNIX System, and TUNIS, Reading, MA: Addison-Wesley Publishing Co., 1983.

This book contains several chapters on the UNIX system. These chapters describe the files, the UNIX shell and the UNIX kernel which Holt calls the UNIX nucleus. These chapters provide a simple, easy to comprehend overview of these features of UNIX.

11. IBM, "Appendix A: VSAM Background", OS PL/I Optimizing Compiler: Programmers Guide, 1981, p258.12-258.23.

This appendix is a section of an IBM manual that describes some of the features and capabilities of the Virtual Sequential Access Method. This method is an example of an already existing B+ tree access method.

12. Johnson, S.C., "Portability of C Programs and the UNIX System", The Bell System Technical Journal, vol.57, no.6, July-August 1978, p2021-2048.

This article discusses portability in reference to C programs and the UNIX system. It describes benefits and advantages of portability and then gives examples. This article shows additional advantages available to the C programmer and the UNIX user.

13. Kamal-Neimat, Marie Anne, Search Mechanisms for Large Files, Ann Arbor, MI: UMI Research Press, 1981, p4-11.

This book briefly describes B-trees. It also covers the indexed sequential access method.

14. Kernighan, Brian W. and Ritchie, Dennis M., The C Programming Language, Englewood Cliffs, NJ: Prentice Hall, Inc., 1978.

This book is a reference to the C programming language. It describes what is available to the programmer and gives program examples. This book will be used to aid in the programming portion of this thesis.

15. Kernighan, Brian and Dennis M.Ritchie, "UNIX Programming - Second Edition", UNIX Timesharing System: UNIX Programmers Manual, vol. 2, second edition, Murray Hill, NJ: Bell Telephone Laboratories, Inc., 1983, p301-322.

This is a paper that introduces programming on the UNIX operating system. It includes topics on file access and low-level I/O, including the read, write, open, creat, and close functions. It also describes calls on the standard I/O library.

16. Kernighan, Brian W. and Pike, Rob, The UNIX Programming Environment, Englewood Cliffs, NJ: Prentice Hall, Inc. 1984.

This book is a very good reference to the UNIX system. All aspects of the UNIX system are covered including chapters on the file system and standard I/O. The file system is shown at the programming level as well as an overview of the kernel level.

17. King, Richard Allen, The MS-DOS Handbook, Berkley, CA: SYBEX Inc. 1985.

This book is a reference to the MS-DOS operating system. It describes the system calls, files, disk layouts and many other aspects of the MSDOS operating system. It discusses both the system level and the user level. It is a complete guide to the internal structure of MSDOS.

18. Knuth, Donald E., The Art of Computer Programming: Sorting and Searching, vol.3, Reading, MA: Addison Wesley Publishing Co., p473-480.

This article describes B-trees as a data structure used to search and update large files. There are illustrations and examples of B-trees, their insertion and deletion and suggestions on some changes to the structure.

19. Lesk, M.E., "Some Applications of Inverted Indexes on the UNIX System", UNIX Programmers Manual, vol.2, Murray Hill, NJ: Bell Telephone Laboratories, 1983, p175-187.

This article describes programs in the UNIX directory /usr/lib/refer, that are used in making and searching an index and isolating keys.

20. Lewis, T.G., and Smith, M.Z., Applying Data Structures, Boston, MA: Houghton Mifflin Co., 1976, p212-218.

This book contains a chapter on B-trees which describes their structure, how to search them, and insertions and deletions into B-trees.

21. Microsoft MS-DOS Operating System Users Guide, Bellevue: Microsoft, June 1983.

This manual describes the MS-DOS operating system, how to use it and the system functions available on it.

22. Rainbow MS-DOS V2.05 Programmers Guide, Maynard: Digital Equipment Corporation, November 1984.

This manual describes the MS-DOS operating system as it relates to the Rainbow computer.

23. Ritchie, D.M., "UNIX Time-Sharing System: A Retrospective", The Bell System Technical Journal, vol.57, no.6, July-August 1978, p1947-1969.

This article contains a section on the file system that discusses the lack of structure and lack of records on the UNIX file system. It also contains sections on I/O devices files, processes, and directories.

24. Ritchie, Dennis M., "The UNIX I/O System", UNIX Timesharing System: UNIX Programmers Manual, vol.2, seventh edition, Murray Hill,NJ: Bell Telephone Laboratories, Inc., 1983, p522-528.

This article gives an overview of the UNIX I/O system. It discusses the open, creat, read, write and close I/O calls. It also describes the data structures associated with a file.

25. Ritchie, D.M., Johnson, S.C., Lesk, M.E., and Kernighan, B.W., "UNIX Timesharing System: The C Programming Language", The Bell System Technical Journal, vol.57, no.6, July-August 1978, p1991-2019.

This article is an overview of the C language. It starts with the history of C and then covers various aspects of the C language including pointers, arrays, program structure and scope of variables. It is a shortened version of the book written by Kernighan and Ritchie.

26. Ritchie, D.M., and Thompson, K., "The UNIX Timesharing System", The Bell System Technical Journal, vol.57, no.6, July-August 1978, p1905-1929.

This article has a section on the file system that covers the three types of files, file protection, I/O and the implementation of the file system. This included describing inodes, i-lists and i-numbers, and system calls that referred to them.

27. Rogers, John, "A UNIX Bibliography", Dr. Dobb's Journal, December 1984, p50-59.

This article breaks the UNIX kernel into pieces, describes them briefly, and gives sources of information available on each topic.

28. Silvester, Peter P., The UNIX System Guidebook, New York, NY: Springer- Verlag, 1984.

This book is an introduction to Version 7 UNIX. It gives a history of UNIX, a description of the file system, the shell and the kernel. It also gives an introduction to text processing, languages and compilers and it gives an explanation of some of the UNIX commands.

29. Stonebraker, M., Wong, E., Kreps, P., and Held, G., "The Design and Implementation of INGRES", ACM Transactions on Database Systems, vol.1, no.3, September 1976, p189-222.

This article describes INGRES (Interactive Graphics and Retrieval System) which is a relational database on UNIX and is primarily programmed in C. This is an example of an already existing database using UNIX files.

30. Stunk, William Jr., The Elements of Style, NY, NY: Macmillian Publishing Co.,Inc., 1979.

This is a guide to writing. It includes misused words and expressions and the principles of composition.

31. Teorey, Toby J. and Fry, James R., Design of Database Structures, Englewood Cliffs, NJ: Prentice-Hall Inc., 1982, p293-333.

This book contains a chapter on the workings of binary trees, B-trees, B+ trees and their variations. It describes insertions and deletions

in each and compares the various methods. It contains many diagrams which aid in the understanding of these database structures.

32. Thompson, K., "UNIX Timesharing System: UNIX Implementation", The Bell System Technical Journal, vol.57, no.6, July-August 1978, p1931-1946.

The UNIX file system from the kernel level was described in this article. It tells how the file system defines disk structure, how files are opened, created, written and read, and all about the inodes, i-lists and i-numbers.

33. Thompson, K., "UNIX Implementation", UNIX Timesharing System: UNIX Programmers Manual, vol.2, seventh edition, Murray Hill, NJ: Bell Telephone Laboratories, Inc., 1983, p512-521.

This paper describes the implementation of the UNIX kernel. It contains a section on the I/O system and a section on the UNIX file system.

34. Tichy, H.J., Effective Writing For Engineers Managers Scientists, NY, NY: John Wiley and Sons, Inc., 1966.

This is a guide to writing from the start to finish of a paper. It gives examples of common errors and presents correct and incorrect writing examples under each topic.

35. Ullman, Jeffrey D., Principles of Database Systems, Potomac, MD: Computer Science Press, Inc., 1980, p42-51.

This chapter describes an index hierarchy and then goes on to describe B-trees. It discusses look-up, insertion and deletion along with illustrations.

36. Wirth, N., Algorithms + Data Structures = Programs, Englewood Cliffs, NJ: Prentice Hall Inc., 1976, p245-265.

This chapter from the data structure text describes B-trees and gives actual programming examples in Pascal pseudo-code. This is very useful to someone developing a B-tree or B+ tree program.