OS-9 File System

Disk File Organization

RBF supports a tree-structured file system. The physical disk organization is designed for efficient use of disk space, resistance to accidental damage, and fast file access. The system also has the advantage of relative simplicity.

Basic Disk Organization

RBF supports logical sector sizes in integral binary multiples from 256 to 32768 bytes. If you use a disk system that cannot directly support the logical sector size (for example, 256 byte logical sectors on a 512-byte physical sector disk), the driver module must divide or combine sectors as required to simulate the required logical size.

Many disks are physically addressed by track number, surface number, and sector number. To eliminate hardware dependencies, OS-9 uses a ***logical sector number*** (LSN) to identify each sector without regard to track and surface numbering.

It is the responsibility of the disk driver module or the disk controller to map logical sector numbers to track/surface/sector addresses. OS-9's file system uses LSNs from 0 to (n-1), where "n" is the total number of sectors on the drive.

NOTE: All sector addresses discussed in this section refer to LSNs.

The format utility initializes the file system on blank or recycled media by creating the track/surface/sector structure. format also tests the media for bad sectors and automatically excludes them from the file system.

Every OS-9 disk has the same basic structure. An ***identification sector*** is located in logical sector zero (LSN 0). It contains a description of the physical and logical format of the storage volume (disk media). A ***disk allocation map*** usually begins in logical sector one (LSN 1). This indicates which disk sectors are free for use in new or expanded files. A ***root directory*** of the volume begins immediately after the disk allocation map.

Identification Sector

LSN zero always contains the identification sector (see Figure 7-1). It describes the physical format of the disk, the size of the allocation map, and the location of the root directory. It also contains the volume name, date and time of creation, etc. If the disk is a bootable system disk it also has the starting LSN and size of the OS9Boot file.

Addr	Size	Name	Description
$00	3	DD_TOT	Total number of sectors on media
$03	1	DD_TKS	Track size in sectors
$04	2	DD_MAP	Number of bytes in allocation map
$06	2	DD_BIT	Number of sectors/bit (cluster size)
$08	3	DD_DIR	LSN of root directory file descriptor
$0B	2	DD_OWN	Owner ID
$0D	1	DD_ATT	Attributes
$0E	2	DD_DSK	Disk ID
$10	1	DD_FMT	Disk Format; density/sides Bit 0: 0 = single side 1 = double side Bit 1: 0 = single density (FM) 1 = double density (MFM) Bit 2: 1 = double track (96 TPI/135 TPI) Bit 3: 1 = quad track density (192 TPI) Bit 4: 1 = octal track density (384 TPI)
$11	2	DD_SPT	Sectors/track (two byte value DD_TKS)
$13	2	DD_RES	Reserved for future use
$15	3	DD_BT	System bootstrap LSN
$18	2	DD_BSZ	Size of system bootstrap
$1A	5	DD_DAT	Creation date
$1F	32	DD_NAM	Volume name
$3F	32	DD_OPT	Path descriptor options
$5F	1		Reserved
$60	4	DD_SYNC	Media integrity code
$64	4	DD_MapLSN	Bitmap starting sector number (0=LSN 1)
$68	2	DD_LSNSize	Media logical sector size (0=256)
$6A	2	DD_VersID	Sector 0 Version ID

Allocation Map

The allocation map shows which sectors are allocated to files and which are free for future use. DD_MapLSN specifies the allocation map start address, which is usually 1. If this field is 0, assume an address of 1. The size of the map varies according to how many bits are needed. Each bit in the allocation map represents a cluster on the disk. If a bit is set, the cluster is considered to be in use, defective, or non-existent. DD_MAP (see Figure 7-1) specifies the actual number of bytes used in the map.

NOTE: The DD_Bit variable specifies the number of sectors per cluster. The number of sectors per cluster is always an integral power of two.

The format utility sets the size of the allocation map depending on the size and number of sectors per cluster. You can select the number of sectors per cluster on the command line when invoking the format utility.

Root Directory

The root directory file is the parent directory of all other files and directories on the disk. It is the directory accessed using the physical device name (such as /d1). Usually, it immediately follows the allocation map. The location of the root directory file descriptor is specified in DD_DIR (see Figure 7-1).

Basic File Structure

OS-9 uses a multiple-contiguous-segment type of file structure. Segments are physically contiguous sectors that store the file's data. If all the data cannot be stored in a single segment, additional segments are allocated to the file. This may occur if a file is expanded after creation, or if a sufficient number of contiguous free sectors is not available.

The OS-9 segmentation method was designed to keep a file's data sectors in as close physical proximity as possible to minimize disk head movement. Frequently, files (especially small files) have only one segment. This results in the fastest possible access time. Therefore, it is good practice to initialize the size of a file to the maximum expected size during or immediately after its creation. This allows OS-9 to optimize its storage allocation.

All files have a sector called a file descriptor sector, or FD. FD contains a list of the data segments with their starting LSNs and sizes. This is also where information such as file attributes, owner, and time of last modification is stored. Only the system uses this sector; it is not directly accessible by the user. The table in Figure 7-2 describes the contents of a file descriptor.

NOTE: ***Offset*** refers to the location of a field, relative to the starting address of the file descriptor. Offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: sys.l or usr.l.

Offset	Size	Name	Description
$00	1	FD_ATT	File Attributes: D S PE PW PR E W R
$01	2	FD_OWN	Owner's User ID
$03	5	FD_DAT	Date Last Modified: Y M D H M
$08	1	FD_LNK	Link Count
$09	4	FD_SIZ	File Size (number of bytes)
$0D	3	FD_CREAT	Date Created: Y M D
$10	240	FD_SEG	Segment List: see below

Figure 7-2: File Descriptor Content Description

The attribute byte (FD_ATT) contains the file permission bits. Bit 7 is set to indicate a directory file, bit 6 indicates a non-sharable file, bit 5 indicates public execute, bit 4 indicates public write, etc.

The date last modified (FD_DAT) changes when a file is opened in write or update mode. This is useful for making date-dependant backups.

The segment list (FD_SEG) consists of a series of five-byte entries, continuing until the end of the logical sector. For 256-byte sectors, this results in 48 entries. These entries have the size and address of each block of storage used by the file in logical order. Each entry has a three-byte logical sector number that specifies the beginning of the block and a two-byte block size (in sectors). Unused segments must be zero.

The RBF file manager maintains the file pointer, logical end-of-file, etc., used by application software and converts them to the logical disk sector number using the data in the segment list.

You do not have to be concerned with physical sectors. OS-9 provides fast random access to data stored anywhere in the file. All the information required to map the logical file pointer to a physical sector number is packaged in the file descriptor sector. This makes OS-9's record-locking functions very efficient.

Segment Allocation

Each device descriptor module has a value called a ***segment allocation size***. It specifies the minimum number of sectors to allocate to a new segment. The goal is to avoid a large number of tiny segments when a file is expanded. If your system uses a small number of large files, this field should be set to a relatively high value, and vice versa.

When a file is created, it has no data segments allocated to it. Write operations past the current end-of-file (the first write is always past the end-of-file) cause allocation of additional sectors to the file. Subsequent expansions of the file are also generally made in minimum allocation increments.

NOTE: An attempt is made to expand the last segment before attempting to add a new segment.

If not all of the allocated sectors are used when the file is closed, the segment is truncated and any unused sectors are de-allocated in the bitmap. This strategy does not work very well for random-access data bases that expand frequently by only a few records. The segment list is rapidly filled with small segments. A provision has been added to prevent this from being a problem.

If a file (opened in write or update mode) is closed when it is not at end-of-file, the last segment of the file is not truncated. To be effective, all programs that deal with the file in write or update mode must ensure that they do not close the file while at end-of-file, or the file will lose any excess space it may have. The easiest way to ensure this is to do a seek(0) before closing the file. This method was chosen because random access files are frequently somewhere other than end-of-file, and sequential files are almost always at end-of-file when closed.

Directory File Format

Directory files have the same physical structure as other files with one exception: RBF must impose a convention for the logical contents of a directory file.

A directory file consists of an integral number of 32-byte entries. The end of the directory is indicated by the normal end-of-file. Each entry consists of a field for the file name and a field for the file's file descriptor address.

The file name field (DIR_NM) is 28 bytes long (bytes 0-27) and has the sign bit of the last character of the file name set. The first byte is set to zero, indicating a deleted or unused entry. The file descriptor address field (DIR_FD) is three bytes long (bytes 29-31) and is the LSN of the file's FD sector. Byte 28 is not used and must be zero.

When a directory file is created, two entries are automatically created: the dot (.) and double dot (..) directory entries. These specify the directory and its parent directory, respectively.

Raw Physical I/O on RBF Devices

You can open an entire disk as one logical file. This allows access of any byte(s) or sector(s) by physical address without regard to the normal file system. This feature is provided for diagnostic and utility programs that must be able to read and write to ordinarily non-accessible disk sectors.

A device is opened for physical I/O by appending the at (@) character to the device name. For example, you can open the device /d2 for raw physical I/O under the pathlist /d2@.

Standard open, close, read, write, and seek system calls are used for physical I/O. A seek system call positions the file pointer to the actual disk physical address of any byte. To read a specific sector, perform a seek to the address computed by multiplying the LSN by the logical sector size of the media. You can find the logical sector size in the PD_SctSiz field of the path descriptor (if 0, assume a value of 256 bytes). For example, on 1024-byte logical media, to read sector 3, perform a seek to address 3072 (1024 * 3), followed by a read system call requesting 1024 bytes.

If the number of sectors per track of the disk is known or read from the identification sector, any track/sector address can be readily converted to a byte address for physical I/O.

WARNINGS: Use extreme care with the special "@" file in update mode. To keep system overhead low, record locking routines only check for conflicts on paths opened for the same file. The "@" file is considered different from any other file, and therefore only conforms to record lockouts with other users of the "@" file.

Improper physical I/O operations can corrupt the file system. Take great care when writing to a raw device. Physical I/O calls also bypass the file security system. For this reason, only super-users are allowed to open the raw device for write permit. Non-super-users are only permitted to read the identification sector (LSN 0) and the allocation bitmap. Attempts to read past this return an end-of-file error.

Record Locking

Record locking is a general term that refers to preserving the integrity of files that more than one user or process can access. OS-9 record locking is designed to be as invisible as possible to application programs.

Most programs may be written without special concern for multi-user activity.

Simply stated, record locking involves:

- ¿ Recognizing when a process is trying to read a record that another process may be modifying.
- ¡ Deferring the read request until the record is safe.

This is referred to as conflict detection and prevention. RBF record locking also handles non-sharable files and deadlock detection.

Record Locking and Unlocking

Conflict detection must determine when a record is in the process of being updated. RBF provides true record locking on a byte basis. A typical record update sequence is:

OS9 I$Read	*program reads record*	*RECORD IS LOCKED*
.		
.	*program updates record*	
.		
OS9 I$Seek	*reposition to record*	
OS9 I$Write	*record is rewritten*	*RECORD IS RELEASED*

When a file is opened in update mode, *ANY* read causes the record to be locked out because RBF does not know in advance if the record will be updated. The record remains locked until the next read, write, or close occurs. Reading files that are opened in read or execute modes does not cause record locking to occur because records cannot be updated in these two modes.

A subtle but nasty problem exists for programs that interrogate a data base and occasionally update its data. When a user looks up a particular record, the record could be locked out indefinitely if the program neglects to release it. The problem is characteristic of record locking systems; you can avoid it by careful programming.

NOTE: Only one portion of a file may be locked out at one time. If an application requires more than one record to be locked out, multiple paths to the same file may be opened with each path having its own record locked out. RBF notices that the same process owns both paths and keeps them from locking each other out. Alternatively, the entire file may be locked out, the records updated, and the file released.

Non-sharable Files

You may use file locking when an entire file is considered unsafe for use by more than one user. On rare occasions, you need to create a ***non-sharable file***. A non-sharable file can never be accessed by more than one process at a time. Make a file non-sharable by setting the single user (S) bit in the file's attribute byte. You can set the bit when you create the file, or later using the attr utility.

If the single-user bit is set, only one process may open the file at a time. If another process attempts to open the file, error (#253) is returned.

More commonly, a file needs to be non-sharable only during the execution of a specific program. Accomplish this by opening the file with the single-user bit set in the access mode parameter.

For example, if a file is opened as a non-sharable file, when it is being sorted it is treated as though it had a single-user attribute. If the file was already opened by another process, an error (#253) is returned.

A necessary quirk of non-sharable files is that they may be duplicated using the I$Dup system call, or inherited. A non-sharable file could therefore actually become accessible to more than one process at a time. Non-sharable only means that the file may be opened once. It is usually a very bad idea to have two processes actively using any disk file through the same (inherited) path.

End of File Lock

An EOF lock occurs when the user reads or writes data at the end of file. The user keeps the end of file locked until a read or write is performed that is not at the end of the file. EOF lock is the only time that a write call automatically causes lock out of any part of the file. This avoids problems that could occur when two users try to simultaneously extend a file.

An extremely useful side effect occurs when a program creates a file for sequential output. As soon as the file is created, EOF lock is gained, and no other process is able to pass the writer in processing the file.

For example, if you redirect an assembly listing to a disk file, a spooler utility can open and begin listing the file before the assembler has written even the first line of output. Record locking always keeps the spooler one step behind the assembler, making the listing come out as desired.

Deadlock Detection

A deadlock can occur when two processes attempt to gain control of the same two disk areas simultaneously. If each process gets one area (locking out the other process), both processes are stuck permanently, waiting for a segment that can never become free. This situation is a general problem that is not restricted to any particular record locking method or operating system.

If this occurs, a deadlock error (#254) is returned to the process that caused it to be detected. It is easy to create programs that, when executed concurrently, generate lots of deadlock errors. The easiest way to avoid them is to access records of shared files in the same sequences in all processes that may be run simultaneously. For example, always read the index file before the data file, never the other way around.

When a deadlock error does occur, it is not sufficient for a program to simply re-try the operation in error. If all processes used this strategy, none would ever succeed. At least one process must release its control over a requested segment for any to proceed.

Record Locking Details for I/O Functions

Open/Create: The most important guideline to follow when opening files is: Do not open a file for update if you only intend to read. Files open for read only do not cause records to be locked out, and they generally help the system to run faster. If shared files are routinely opened for update on a multi-user system, users can become hopelessly record-locked for extended periods of time.

Use the special "@" file in update mode with extreme care. To keep system overhead low, record locking routines only check for conflicts on paths opened for the same file. The "@" file is considered different from any other file, and therefore only conforms to record lockouts with other users of the "@" file.

Read/ReadLine: Read and ReadLine cause lock out of records only if the file is open in update mode. The locked out area includes all bytes starting with the current file pointer and extending for the number of bytes requested.

For example, if you make a ReadLine call for 256 bytes, exactly 256 bytes are locked out, regardless of how many bytes are actually read before a carriage return is encountered. EOF lock occurs if the bytes requested include the current end-of-file.

A record remains locked until any of the following occur:

- Another read is performed
- A write is performed
- The file is closed
- A record lock SetStat is issued

Releasing a record does not normally release EOF lock. Any read or write of zero bytes releases any record lock, EOF lock, or File lock.

Write/WriteLine: Write calls always release any record that is locked out. In addition, a write of zero bytes releases EOF lock and File lock. Writing usually does not lock out any portion of the file unless it occurs at end of file when it will gain EOF lock.

Seek: Seek does not effect record locking.

SetStatus: There are two SetStat codes to deal with record locking: SS_Lock locks or releases part of a file. SS_Ticks sets the length of time a program will wait for a locked record. See the I$SetStat entry in ***OS-9 System Calls*** (chapter 2) for a description of the codes.

File Security

Each file has a group/user ID that identifies the file's owner. These are copied from the current process descriptor when the file is created. Usually, a file's owner ID is not changed.

An attribute byte is also specified when a file is created. The file's attribute byte tells RBF in which modes the file may be accessed. Together with the file's owner ID, the attribute byte provides (some) file security.

The attribute byte has two sets of bits to indicate whether a file may be opened for read, write, or execute by the ***owner*** or the ***public***. In this context, the file's owner is any user with the same group ID as the file's creator. Public means any user with a different group ID.

Whenever a file is opened, access permissions are checked on all directories specified in the pathlist, as well as the file itself. If you do not have permission to read a directory, you may not read any files in that directory.

Any ***super-user*** (a user with group ID of zero) may access any file in the system. Files owned by the super-user cannot be accessed by users of any other group unless specific access permissions are set. Files containing modules owned by the super-user must also be owned by the super-user. If not, the modules contained within the file are not loaded.

CAVEAT: The system manager should exercise caution when assigning group/user IDs. The RBF File Descriptor stores the group/user ID in a two byte field (FD_OWN). The group/user ID that resides in the password file is permitted two bytes for the group ID and two bytes for the user ID. RBF only reads the low order byte of both the group and user ID. Consequently, a user with the ID of 256.512 is mistaken for the super user by RBF.

End of Chapter 7

NOTES