Chapter 3

Filesystem Concepts

This chapter explains some important concepts about hard disk *filesystems*, the structure by which files and directories are organized in the IRIX system. The chapter describes the primary types of IRIX filesystems, the older Extent File System (EFS) and the newer XFS filesystem, and other disk filesystems. It explains concepts that are important to filesystem administration such as IRIX directory organization, filesystem features, filesystem types, creating filesystems, mounting and unmounting filesystems, and checking filesystems for consistency.

The major sections in this chapter are:

Even if you are familiar with the basic concepts of UNIX filesystems, you should read through the following sections. The IRIX EFS and XFS filesystems are slightly different internally from other UNIX filesystems and have slightly different administration commands and procedures.

Filesystem administration procedures are described in Chapter 4, “Creating and Growing Filesystems,” and Chapter 5, “Maintaining Filesystems.”

For information about floppy and CD-ROM filesystems, see the guide *IRIX Admin: Peripheral Devices*.

IRIX Directory Organization

Every IRIX system disk contains some standard directories. These directories contain operating system files organized by function. This organization is not entirely logical; it has evolved over time and has its roots in several versions of UNIX. Table 3-1 lists the standard directories that most systems have. It also lists alternate names for those directories in some cases. The alternate names are usually an older pathname for the directory and are provided (as symbolic links) to ease the transition from old pathnames to new pathnames as the IRIX directory organization evolves.

Table 3-1 Standard Directories and Their Contents

Directory	Alternate Name	Contents
/		The root directory, contains the IRIX kernel (/unix)
/dev		Device files for terminals, disks, tape drives, CD-ROM drives, and so on
/etc		Critical system configuration files and maintenance commands
/etc/config	/var/config, /usr/var/config	System configuration files
/lib		Critical compiler binaries and libraries
/lib32		Critical compiler binaries and libraries
/lib64		Critical compiler binaries and libraries for 64-bit systems (IP19, IP21, and IP26)

Table 3-1 (continued) Standard Directories and Their Contents

Directory	Alternate Name	Contents
/lost+found		Holding area for files recovered by the *fsck* command
/proc	/debug	Process (debug) filesystem
/sbin		Commands needed for minimal system operability
/stand		Standalone utilities (*fx*, *ide*, *sash*)
/tmp		Temporary files
/tmp_mnt		Mount point for automounted filesystems
/usr		On some systems, a filesystem mount point
/usr/bin	/bin	Commands
/usr/bsd		Commands
/usr/demos		Demo programs
/usr/etc		Critical system configuration files and maintenance commands
/usr/include		C header files
/usr/lib		Libraries and support files
/usr/lib32		Libraries and support files
/usr/lib64		Libraries and support files for 64-bit systems (IP19, IP21, and IP26)
/usr/local		Non-Silicon Graphics system commands and files
/usr/lost+found		Holding area for files recovered by the *fsck* command
/usr/people		Home directories
/usr/relnotes		Release Notes
/usr/sbin		Commands
/usr/share		Shared data files for various applications

Table 3-1 (continued) Standard Directories and Their Contents

Directory	Alternate Name	Contents
/usr/share/Insight		InSight books
/usr/share/catman		Reference pages (man pages)
/usr/var		Present if / and */usr* are separate filesystems
/var		System files likely to be customized or machine-specific
/var/X11		X11 configuration files
/var/adm	/usr/adm	System log files
/var/inst		Software installation history
/var/mail	/usr/mail	Incoming mail
/var/nodelock		NetLS nodelock license file
/var/preserve	/usr/preserve	Temporary editor files
/var/spool	/usr/spool	Printer support files
/var/tmp	/usr/tmp	Temporary files
/var/yp		NIS commands

General Filesystem Concepts

A filesystem is a data structure that organizes *files* and *directories* on a disk partition so that they can be easily retrieved. Only one filesystem can reside on a disk partition.

A file is a one-dimensional array of bytes with no other structure implied. Information about each file is stored in structures called *inodes* (inodes are described in the next section "Inodes"). Files cannot span filesystems.

A directory is a container that stores files and other directories. It is merely another type of file that the user is permitted to use, but not allowed to write; the operating system itself retains the responsibility for writing directories. Directories cannot span filesystems. The combination of directories and files make up a filesystem.

The starting point of any filesystem is an unnamed directory that serves as the root for that particular filesystem. In the IRIX operating system there is always one filesystem that is itself referred to by that name, the Root filesystem. Traditionally, the root directory of the Root filesystem is represented by a single slash (/). Filesystems are attached to the directory hierarchy by the *mount* command. The result is the IRIX directory structure shown in Figure 3-1.

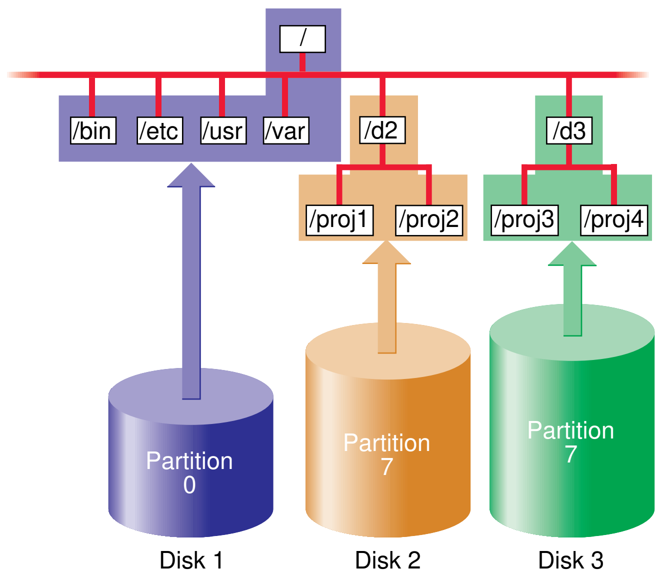

Figure 3-1 The IRIX Filesystem

You can join two or more disk partitions to create a *logical volume*. The logical volume can be treated as if it were a single disk partition, so a filesystem can reside on a logical volume and hence is the only way for a single filesystem to span more than one disk. Logical volumes are covered beginning in Chapter 6, “Logical Volume Concepts.”

The following subsections describe key components of filesystems.

Inodes

Information about each file is stored in a structure called an *inode*. The word inode is an abbreviation of the term *index node*. An inode is a data structure that stores all information about a file except its name, which is stored in the directory. Each inode has an identifying inode number, which is unique across the filesystem that includes the file.

An inode contains this information:

- the type of the file (see the next section, “Types of Files,” for more information)
- the access mode of the file; the mode defines the access permissions *read*, *write*, and *execute* and may also contain security labels and access control lists
- the number of hard links to the file (see the section “Hard Links and Symbolic Links” for more information)
- who owns the file (the owner’s user-ID number) and the group to which the file belongs (the group-ID number)
- the size of the file in bytes
- the date and time the file was last accessed, and last modified
- information for finding the file’s data within the disk partition or logical volume
- the pathname of symbolic links (when they fit and on XFS filesystems only)

You can use the *ls* command with various options to display the information stored in inodes. For example, the command *ls -l* displays all but the last two items in the list above in the order listed (the date shown is the last modified time).

Inodes do not contain the name of the file or its directory.

Types of Files

Filesystems can contain the types of files listed Table 3-2. The type of a file is indicated by the first character in the line of *ls -l* output for the file.

Table 3-2 Types of Files

Type of File	Character	Description
Regular files	–	Regular files are one-dimensional arrays of bytes.
Directories	d	Directories are containers for files and other directories.
Symbolic links	l	Symbolic links are files that contain the name of another file or a directory.
Character devices	c	Character devices enable communication between hardware and IRIX; data is accessed on a character by character basis.
Block devices	b	Block devices enable communication between hardware and IRIX; data is accessed in blocks from a system buffer cache.
Named pipes (also known as FIFOs)	p	Named pipes allow communication between two unrelated processes running on the same host. They are created with the *mknod* command (see the section "Creating Device Files With mknod" in Chapter 2 for more information on *mknod*).
UNIX domain sockets	s	UNIX domain sockets are connections between processes that allow them to communicate, possibly over a network.

Hard Links and Symbolic Links

As discussed in the section "Inodes" in this chapter, information about each file, except for the name and directory of the file, is stored in an inode for the file. The name of the file is stored in the file's directory and a link to the file is created by associating the filename with an inode number. This type of link is called a *hard link*. Although every file is a hard link, the term is usually used only when two or more filenames are associated with the same inode number. Because inode numbers are unique only within a filesystem, hard links cannot be created across filesystem boundaries.

The second and later hard links to a file are created with the *ln* command, without the **-s** option. For example, say the current directory contains a file called *origfile*. To create a hard link called *linkfile* to the file *origfile*, give this command:

```
% ln origfile linkfile
```

The output of *ls -l* for *origfile* and *linkfile* shows identical sizes and last modification times:

```
% ls -l origfile linkfile
-rw-rw-r--    2 joyce    user         4 Apr  5 11:15 origfile
-rw-rw-r--    2 joyce    user         4 Apr  5 11:15 linkfile
```

Because *origfile* and *linkfile* are simply two names for the same file, changes in the contents of the file are visible when using either filename. Removing one of the links has no effect on the other. The file is not removed until there are no links to it (the number of links to the file, the *link count*, is stored in the file's inode).

Another type of link is the *symbolic link*. This type of link is actually a file (see Table 3-2). The file contains a text string, which is the pathname of another file or directory. Because a symbolic link is a file, it has its own owners and permissions. The file or directory it points to can be in another filesystem. If the file or directory that a symbolic link points to is removed, it is no longer available and the symbolic link becomes useless until the target is recreated (it is called a *dangling symbolic link*).

Symbolic links are created with the *ln* command with the **-s** option. For example, to create a symbolic link called *linkdir* to the directory *origdir*, give this command:

```
% ln -s origdir linkdir
```

The output of *ls -ld* for the symbolic link is shown below. Notice that the permissions and other information don't match. The listing for *linkdir* shows that it is a symbolic link to *origdir*.

```
% ls -ld linkdir origdir
drwxrwxrwt  13 sys     sys  2048 Apr  5 11:37 origdir
lrwxrwxr-x   1 joyce   user    8 Apr  5 11:52 linkdir -> origdir
```

When you use ".." in pathnames that involve symbolic links, be aware that " .." refers to the parent directory of the true file or directory, not the parent of the directory that contains the symbolic link.

For more information about hard and symbolic links, see the ln(1) reference page and experiment with creating and removing hard and symbolic links.

Filesystem Names

Filesystems don’t have names per se; they are identified by their location on a disk or their position in the directory structure in these ways:

- by the block and character device file names of the disk partition or logical volume that contains the filesystem (see the section “Block and Character Devices” in Chapter 1)
- by a mnemonic name for the disk partition or logical volume that contains the filesystem (see the section “Creating Mnemonic Names for Device Files With ln” in Chapter 2)
- by the mount point for the filesystem (see the section “Filesystem Mounting and Unmounting” in this chapter)

The filesystem identifier from the list above that you use with commands that administer filesystems (such as *mkfs*, *mount*, *umount*, and *fsck*) depends upon the command. See the reference page for the command you want to use or examples in this guide to determine which filesystem name to use.

EFS Filesystems

The EFS filesystem is the original IRIX filesystem. It contains an enhancement to the standard UNIX filesystem called *extents* (defined below), and thus is called the Extent File System (EFS). The maximum size of an EFS filesystem is about 8 GB. It uses a filesystem block size of 512 bytes and allows a maximum file size of 2 GB minus 1 byte.

Advanced features of EFS are that it keeps multiple inode tables in close proximity to data blocks rather than a single inode table, and it uses a bitmap to keep track of free blocks instead of a list of free blocks.

Inodes are created when an EFS filesystem is created, not when files are created. When a file is created, an inode is allocated to that file. Thus, the maximum number of files in a filesystem is limited by the number of inodes in that filesystem. By default, the number of inodes created is a function of the size of the partition or logical volume. Typically one inode is created for every 4K bytes in the partition or logical volume. You can specify the number of inodes with the **-n** option to the filesystem creation command, *mkfs*. Inodes use disk space, so there is a tradeoff between the number of inodes and the amount of disk space available for files.

The first block of an EFS filesystem is not used. Information about the filesystem is stored in the second block of the filesystem (block 1), called the *superblock*. This information includes:

- the size of the filesystem, in both physical and logical blocks
- the read-only flag; if set, the filesystem is read only
- the superblock-modified flag; if set, the superblock has been modified
- the date and time of the last update
- the total number of index nodes (*inodes*) allocated
- the total number of inodes free
- the total number of free blocks
- the starting block number of the free block bitmap

After the superblock bitmap is a series of *cylinder groups*. A cylinder group is a group of 1 to 32 contiguous disk cylinders. Each cylinder group contains both inodes and data blocks. Each contiguous group of data blocks that make up a file is called an extent. There are 12 extent addresses in an inode. Extents are of variable length, anywhere from 1 to 148 contiguous blocks.

An inode contains addresses for 12 extents, which can hold a combined 1536 blocks, or 786,432 bytes. If a file is large enough that it cannot fit in the 12 extents, each extent is then loaded with the address of up to 148 *indirect* extents. The indirect extents then contain the actual data that makes up the file. Because EFS uses indirect extents, you can create files up to 2 GB, assuming you have that much disk space available in your filesystem.

The last block of the filesystem is a duplicate of the filesystem superblock. This is a safety precaution that provides a backup of the critical information stored in the superblock.

EFS filesystems can become *fragmented* over time. Fragmented filesystems have small contiguous blocks of free space and files with poor layouts of the file extents. The *fsr* command reorganizes filesystems to improve file extent layout and compact the filesystem free space. By default, *fsr* is run once a week automatically from *crontab*.

XFS Filesystems

XFS is a new IRIX filesystem designed for use on most Silicon Graphics systems—from desktop systems to supercomputer systems. Its major features include

- full 64-bit file capabilities (files larger than 2 GB)
- rapid and reliable recovery after system crashes because of the use of journaling technology
- efficient support of large, sparse files (files with "holes")
- integrated, full-function volume manager, the XLV Volume Manager
- extremely high I/O performance that scales well on multiprocessing systems
- guaranteed-rate I/O for multimedia and data acquisition uses
- compatibility with existing applications and with NFS®
- user-specified filesystem block sizes ranging from 512 bytes up to 64 KB
- small directories and symbolic links of 156 characters or less take no space

At least 32 MB of memory is recommended for systems with XFS filesystems.

XFS supports files and filesystems of 2^{40}-1 or 1,099,511,627,775 bytes (one terabyte) on 32-bit systems (IP17, IP20, and IP22). Files up to 2^{63}-1 bytes and filesystems of unlimited size are supported on 64-bit systems (IP19, IP21, and IP26). You can use the filesystem interfaces supplied with the IRIS Development Option (IDO) software option to write 32-bit programs that can track 64-bit position and file size. Many programs work without modification because sequential reads succeed even on files larger than 2 GB. NFS allows you to export 64-bit XFS filesystems to other systems.

XFS uses database journaling technology to provide high reliability and rapid recovery. Recovery after a system crash is completed within a few seconds, without the use of a filesystem checker such as the *fsck* command. Recovery time is independent of filesystem size.

XFS is designed to be a very high performance filesystem. Under certain conditions, throughput exceeds 100 MB per second. Its performance scales to complement the CHALLENGE™ MP architecture. While traditional filesystems suffer from reduced performance as they grow in size, with XFS there is no performance penalty.

You can create filesystems with block sizes ranging from 512 bytes to 64 KB. For real-time data, the maximum *extent* size is 1 GB. Filesystem extents, which provide contiguous data within a file, are configurable at file creation time using the **fcntl()** system call and are multiples of the filesystem block size. Inodes are created as needed by XFS filesystems. You can specify the size of inodes with the **-i** option to the filesystem creation command, *mkfs*. You can also specify the maximum percentage of the space in a filesystem that can be occupied by inodes with the *mkfs* **-i maxpct=** option.

Most filesystem commands, such as *du*, *dvhtool*, *ls*, *mount*, *prtvtoc*, and *umount*, work with XFS filesystems as well as EFS filesystems with no user-visible changes. A few commands, such as *df*, *fx*, and *mkfs* have additional features for XFS. The filesystem commands *clri*, *fsck*, *findblk*, and *ncheck* are not used with XFS filesystems.

For backup and restore, the standard IRIX commands *Backup*, *bru*, *cpio*, *Restore*, and *tar* and the optional software product NetWorker® for IRIX can be used for files less than 2 GB in size. To dump XFS filesystems, the new command *xfsdump* must be used instead of *dump*. Restoring from these dumps is done using *xfsrestore*. See Table 3-1 and Table 3-2 in Chapter 3, "Dumping and Restoring XFS Filesystems," for more information about the relationships between *xfsdump*, *xfsrestore*, *dump*, and *restore* on XFS and EFS filesystems.

Network File Systems (NFS)

NFS filesystems are available if you are using the optional NFS software. NFS filesystems are filesystems that are exported from one host and mounted on other hosts across a network.

On the hosts where the filesystems reside, they are treated just like any other EFS or XFS filesystem. The only special feature of these filesystems is that they are exported for mounting from other workstations. Exporting NFS filesystems is done with the *exportfs* command. On other hosts, these filesystems are mounted with the *mount* command or by using the automount facility of NFS.

Tip: The sections "Using Disk Space on Other Systems" and "Making Your Disk Space Available to Other Systems" in Chapter 6 of the *Personal System Administration Guide* provide instructions for mounting and exporting NFS filesystems.

NFS filesystems are described in detail in the *ONC3/NFS Administrator's Guide*, which is included with the NFS software option.

Cache File Systems (CacheFS)

The Cache File System (CacheFS) is a new filesystem type that provides client-side caching for NFS and other filesystem types. Using CacheFS on NFS clients with local disk space can significantly increase the number of clients a server can support and reduce the data access time for clients using read-only file systems.

The *cfsadmin* command is used for managing CacheFS filesystems. A special version of the fsck command, *fsck_cachefs* is used to check the integrity of a cache directory. It is automatically invoked when a CacheFS filesystem is mounted. When mounting and unmounting CacheFS filesystems, the **-t cachefs** option must be used. For more information on these commands, see the cfsadmin(1M), fsck_cachefs(1M), and mount(1M) reference pages.

CacheFS filesystems are available if you are using the optional NFS software. They are described in detail in the *ONC3/NFS Administrator's Guide*, which is included with the NFS software option.

/proc Filesystem

The */proc* filesystem, also known as the debug filesystem, provides an interface to running IRIX processes for use by monitoring programs, such as *ps* and *top*, and debuggers, such as *dbx*. The debug filesystem is usually mounted on */proc* with a link to */debug*. To reduce confusion, */proc* is not displayed when you list free space with the *df* command.

The "files" of the debug filesystem are of the form */proc/nnnnn* and */proc/pinfo/nnnnn*, where *nnnnn* is a decimal number corresponding to a process ID. These files do not consume disk space; they are merely handles for debugging processes. */proc* files cannot be removed.

See the proc(4) reference page for more information on the debug filesystem.

Filesystem Creation

To turn a disk partition or logical volume into a filesystem, the *mkfs* command must be used. It takes a disk partition or logical volume and divides it up into areas for data blocks, inodes, and free lists, and writes out the appropriate inode tables, superblocks, and block maps. It creates the filesystem's root directory and, for EFS filesystems only, a *lost+found* directory.

An example *mkfs* command for making an EFS filesystem is:

```
mkfs -t efs /dev/rdsk/dks0d2s7
```

You can use the **-n** option to *mkfs* to specify the number of inodes created.

An example *mkfs* command for making an XFS filesystem with a 1 MB internal log section is:

```
mkfs -l size=1m /dev/rdsk/dks0d2s7
```

An example *mkfs* command for making an XFS filesystem on a logical volume with log and data subvolumes is:

```
mkfs /dev/rdsk/xlv/a
```

After using *mkfs* to create an EFS filesystem, run the *fsck* command to verify that the disk is consistent.

For more instructions on making filesystems see Chapter 4, "Creating and Growing Filesystems," and the mkfs(1M), mkfs_efs(1M), and mkfs_xfs(1M) reference pages.

Filesystem Mounting and Unmounting

Filesystems must be *mounted* to be used. Figure 3-2 illustrates this process. When a filesystem is mounted, the name of the device file for the filesystem (*/dev/rdsk/dks0d2s7* in Figure 3-2) and the name of a directory (*/proj* in Figure 3-2) are given. This directory, */proj,* is called a *mount point* and forms the connection between the filesystem containing the mount point and the filesystem to be mounted. Mounting a filesystem tells the kernel that the mount point is to be considered equivalent to the top level directory of the filesystem when pathnames are resolved. In Figure 3-2, the files *a*, *b*, and *c* in the */dev/rdsk/dks0d2s7* filesystem become */proj/a*, */proj/b*, and */proj/c* as shown in the bottom of the figure.

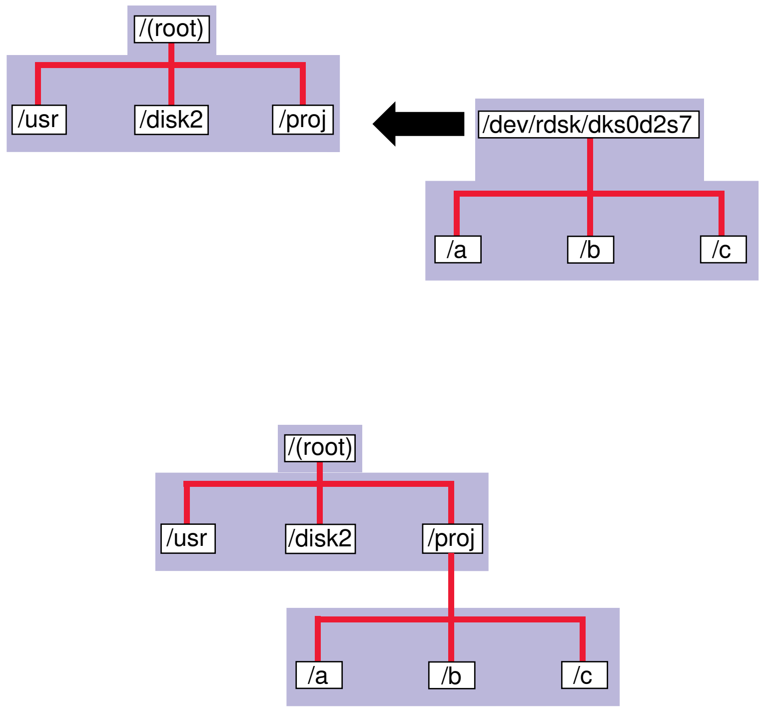

Figure 3-2 Mounting a Filesystem

When you mount a filesystem, the original contents of the mount point directory are hidden and unavailable until the filesystem is unmounted. However, the mount point directory owner and permissions are not hidden. Restricted permissions can restrict access to the mounted filesystem.

Unlike other filesystems, the Root filesystem (/) is mounted as soon as the kernel is running and cannot be unmounted because it is required for system operation. The Usr filesystem, if it is a separate filesystem from the Root filesystem, must also be mounted for the system to operate properly. System administration that requires unmounting the Root and Usr filesystem can be done in the miniroot. See the section “Filesystem Administration From the Miniroot” in this chapter for more information.

You can mount filesystems several ways:

- manually with the *mount* command (discussed in the section "Manually Mounting Filesystems" in Chapter 5)
- automatically when the system is booted, using information in the file */etc/fstab* (discussed in the section "Mounting Filesystems Automatically With the /etc/fstab File" in Chapter 5)
- automatically when the filesystem is accessed (called *automounting*, this applies to NFS (remote) filesystems only; see the section "Mounting a Remote Filesystem Automatically" in Chapter 5)

You can unmount filesystems in these ways:

- shut the system down (filesystems are unmounted automatically)
- manually unmount filesystems with the *umount* command (see the section "Unmounting Filesystems" in Chapter 5)

The *mount* and *umount* commands are described in detail in the section "Mounting and Unmounting Filesystems" in Chapter 5.

Filesystem Checking

The *fsck* command checks EFS filesystem consistency and data integrity. Filesystems are usually checked automatically when the system is booted. Except for the Root filesystem, filesystems must be unmounted while being checked. You might want to invoke *fsck* manually at these times:

- before making a backup
- after doing a restore
- after doing disk maintenance
- before installing software
- before manually mounting a dirty filesystem
- when *fsck* runs automatically and has many errors

Several procedures for invoking *fsck* manually are described in the section "Checking EFS Filesystem Consistency With fsck" in Chapter 5. A detailed explanation of the checks performed by *fsck* and the options it presents when it finds problems are provided in Appendix A, "Repairing EFS Filesystem ProblemsWith fsck."

The *xfs_check* command checks XFS filesystem consistency. It is normally used only when a filesystem consistency problem is suspected. See the xfs_check(1M) reference page for more information.

The *fsck_cachefs* command checks CacheFS filesystem consistency. It is automatically run when CacheFS filesystems are mounted. See the fsck_cachefs(1M) reference page and the *ONC3/NFS Administrator's Guide* for more information.

Filesystem Reorganization

EFS filesystems can become fragmented over time. When a filesystem is fragmented, blocks of free space are small and files have many extents (see the section "EFS Filesystems" in this chapter for information about extents). The *fsr* command reorganizes filesystems so that the layout of the extents is improved and free disk space is coalesced. This improves overall performance. By default, *fsr* is run automatically once a week from *crontab*. See the fsr(1M) reference page for additional information.

Filesystem Administration From the Miniroot

When filesystem modifications or other administrative tasks require that the Root filesystem not be mounted or not be in use, the miniroot environment provided by the software installation tools included on IRIX system software release CDs can be used. When using the miniroot, a limited version of IRIX is installed in the swap partition in a filesystem mounted at /. The system runs this version of IRIX rather than the standard IRIX in the Root and Usr filesystems. The Root and Usr filesystems are available and mounted at */root* and */root/usr*. Thus the pathnames of all files in the Root and Usr filesystems have the prefix */root*.

How to Add Filesystem Space

You can add filesystem space in three ways:

- Add a new disk, create a filesystem on it, and mount it as a subdirectory on an existing filesystem.
- Change the size of the existing filesystems by removing space from one partition and adding it to another partition on the same disk.
- Add another disk and grow an existing filesystem onto that disk with the *growfs* or *xfs_growfs* command.

These three methods of adding filesystem space are discussed in the following subsections.

Mount a Filesystem as a Subdirectory

To mount a filesystem as a subdirectory, you simply add a new disk with a separate filesystem and create a new mount point for it within your filesystem. This is generally considered the safest way to add space. For example, if your Usr filesystem is short of space, add a new disk and mount the new filesystem on a directory called */usr/work*. A drawback of this approach is that it does not allow hard links to be created between the original filesystem and the new filesystem.

See Chapter 2, "Performing Disk Administration Procedures," for full information on partitioning a disk and making filesystems on it.

"Steal" Space From Another Filesystem

To move disk space from one filesystem on a disk to another filesystem on the same disk, you must back up your existing data on both filesystems, run the *fx* command to repartition the disk, then remake both filesystems with the *mkfs* command. This method has serious drawbacks. It is a great deal of work and has certain risks. For example, to increase the size of a filesystem, you must remove space from other filesystems. You must be sure that when you are finished changing the size of your filesystems, your old data still fits on all the new, smaller filesystems. Also, resizing your filesystems may at best be a stop-gap measure until you can acquire additional disk space.

Repartitioning is documented in "Repartitioning a Disk With fx" in Chapter 2. For additional solutions when the filesystem is the Root filesystem, see "Running Out of Space in the Root Filesystem" in Chapter 5.

Grow a Filesystem Onto Another Disk

Growing an existing filesystem onto an additional disk or disk partition is another way to increase the available space in that filesystem. The original disk partition and the new disk partition become either an *lv* logical volume or an XLV logical volume (your choice). The *growfs* command (EFS filesystems) or *xfs_growfs* command (XFS filesystems) preserves the existing data on the hard disk and adds space from the new disk partition to the filesystem. This process is simpler than completely remaking your filesystems. The one drawback to growing a filesystem across disks is that if one disk fails, you may not recover data from the other disk, even if the other disk still works. If your Usr filesystem is a logical volume, you will be unable to boot the system into multiuser mode. For this reason, it is preferable, if possible, to mount an additional disk and filesystem as a directory on the Root or Usr or filesystems (on / or */usr*).

For instructions on growing a filesystem onto an additional disk, see the section "Growing an EFS Filesystem Onto Another Disk" or "Growing an XFS Filesystem Onto Another Disk" in Chapter 4.

Disk Quotas

If your system is constantly short of disk space and you cannot increase the amount of available space, you may be forced to implement disk quotas. Quotas allow a limit to be set on the amount of space a user can occupy, and there may be a limit on the number of files (inodes) each user can own. IRIX provides the *quotas* system to automate this process on EFS filesystems (the quotas system cannot be used on XFS filesystems). You can use this system to implement specific disk usage quotas for each user on your system. You may also choose to implement *hard* or *soft* limits. Hard limits are enforced by the system, soft limits merely remind the user to trim disk usage.

With soft limits, whenever a user logs in with a usage greater than the assigned soft limit, that user is warned (by the *login* command). When the user exceeds the soft limit, the timer is enabled. Any time the quota drops below the soft limits, the timer is disabled. If the timer is enabled longer than a time period set by the administrators, the particular limit that has been exceeded is treated as if the hard limit has been reached, and no more resources are allocated to the user. The only way to reset this condition is to reduce usage below the quota. Only *root* may set the time limits and this is done on a per-filesystem basis.

Several options are available with the *quotas* subsystem. You can impose limits on some users and not others, some filesystems and not others, and on total disk usage per user, total number of files, or size of files. The system is completely configurable. You can also keep track of disk usage through the process accounting system provided under IRIX.

The importance of managing disk quotas carefully cannot be over emphasized. It is strongly recommended that if disk quotas are imposed, they should be soft quotas, and every attempt should be made to otherwise rectify the situation before removing someone's files. Before using the *quotas* subsystem to enforce disk usage, carefully read the material on disk quotas in the section "Disk Quotas" in this chapter.

The *quotas* system is described completely in the quotas(4) reference page. The procedure for imposing disk quotas is described in the section "Imposing Disk Quotas" in Chapter 5.

Filesystem Corruption

Most often, a filesystem is corrupted because the system experienced a panic or didn't shut down cleanly. This can be caused by system software failure, hardware failure, or human error (for example, pulling the plug). Another possible source of filesystem corruption is overlapping partitions.

There is no foolproof way to predict hardware failure. The best way to avoid hardware failures is to conscientiously follow recommended diagnostic and maintenance procedures.

Human error is probably the greatest single cause of filesystem corruption. To avoid problems, follow these rules closely:

- Always shut down the system properly. Do not simply turn off power to the system. Use a standard system shutdown tool, such as the *shutdown* command.
- Never remove a filesystem physically (pull out a hard disk) without first turning off power.
- Never physically write-protect a mounted filesystem, unless it is mounted read-only.

The best way to insure against data loss is to make regular, careful backups. See the guide *IRIX Admin: Backup, Security, and Accounting* for complete information on system backups.