

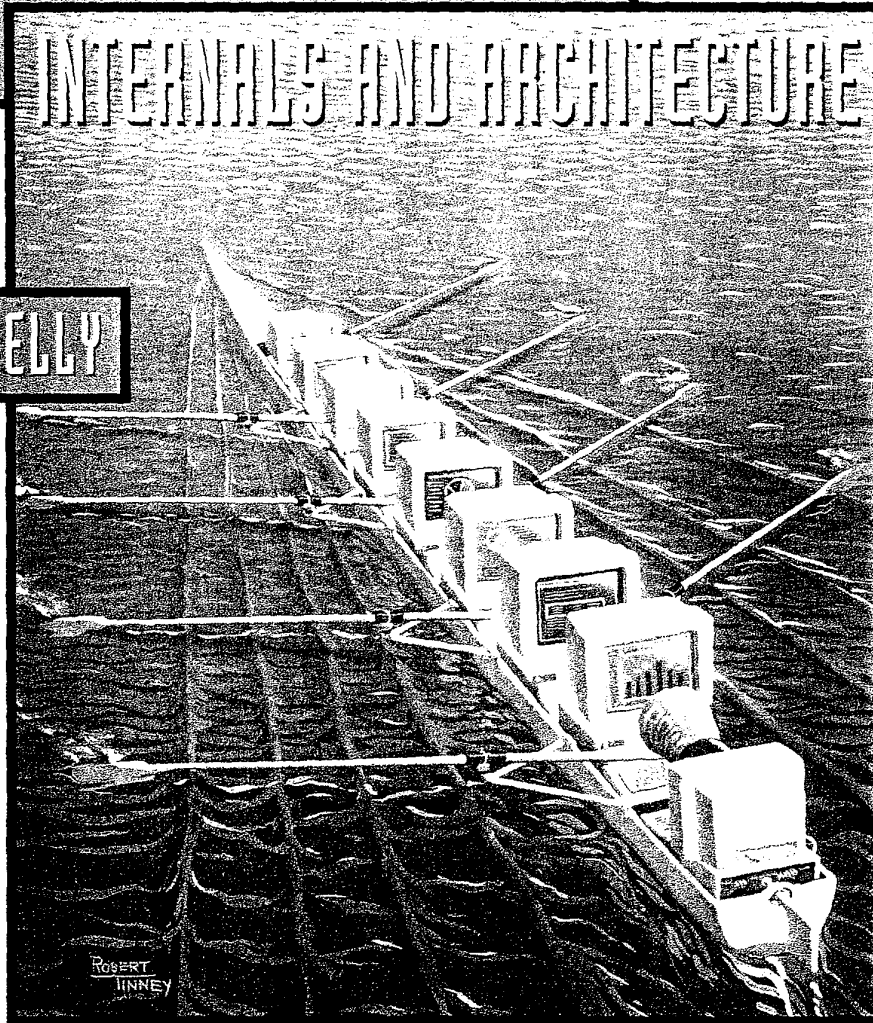
564

Ranade Workstation Series

AIX/6000

INTERNALS AND ARCHITECTURE

DAVID A. KELLY



Library of Congress Cataloging-in-Publication Data

Kelly, David A. (David Allen)

AIX/6000 internals and architecture / David A. Kelly.

p. cm.—(J. Ranade workstation series)

Includes index.

ISBN 0-07-034061-7

1. AIX (Computer file) 2. Operating systems (Computers) 3. IBM RS/6000 Workstation. I. Title. II. Series.

QA76.76.063K452 1996

005.4'469—dc20

95-25794

CIP

McGraw-Hill



A Division of The McGraw-Hill Companies

Copyright © 1996 by The McGraw-Hill Companies, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

1 2 3 4 5 6 7 8 9 0 AGM/AGM 9 0 0 9 8 7 6

ISBN 0-07-034061-7

The sponsoring editor for this book was Jerry Papke, the editing supervisor was Fred Bernardi, and the production supervisor was Pamela Pelton. It was set in Century Schoolbook by Renee Lipton of McGraw-Hill's Professional Book Group composition unit.

Printed and bound by Quebecor / Martinsburg.


This book is printed on acid-free paper.

McGraw-Hill books are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please write to the Director of Special Sales, McGraw-Hill, 11 West 19th Street, New York, NY 10011. Or contact your local bookstore.

Information contained in this work has been obtained by The McGraw-Hill Companies, Inc. ("McGraw-Hill") from sources believed to be reliable. However, neither McGraw-Hill nor its authors guarantees the accuracy or completeness of any information published herein and neither McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that McGraw-Hill and its authors are supplying information, but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

The Journaled File System

6.1 An Overview of File Systems

 The term “file system” has two distinct meanings for UNIX-based systems. The “global file system” refers to the file tree as viewed by the user. It includes the entire hierarchical arrangement of directories and files, from a logical perspective, regardless of the physical components that comprise the tree. In reality, the global file system is made up of one or more physical file systems, which reside on separate disk partitions or other storage media. These physical file systems are connected to form the global file system.

The global file system

Figure 6.1 illustrates the AIX 3.2 file tree and a simplified representation of the file systems on disk.

Author's Note: The details of the files found in each directory are appropriate for a system administration discussion and are not provided here. While Fig. 6.1 shows each file system as a contiguous disk partition, the AIX logical volume manager allows file systems to be fragmented and spread across one or more physical disk drives, as described in Chap. 2.

Each file system has its own root directory, which is mounted onto a stub directory in the file system above. The stub directory is called the mount point. Each mount point directory is shown as boxed in Fig. 6.1.

AIX local disk file systems must reside within disk partitions called “logical volumes.” Each logical volume is considered a device and thus includes a device file abstraction in the /dev directory. Table 6.1 lists the AIX 3.2 file systems, describes their general use, and indicates the device names for their logical volumes.

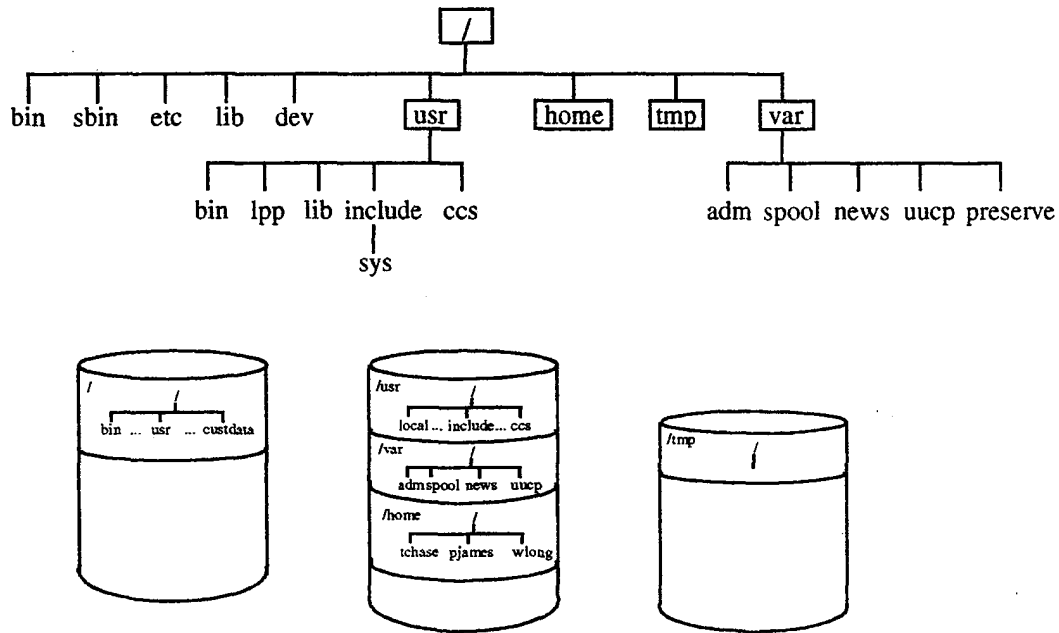


Figure 6.1 The AIX 3.2 file tree and disk file systems.

TABLE 6.1 AIX 3.2 File Systems

FS Name	Device Name	Description
/	/dev/hd4	Holds configuration and boot files specific to the system
/home	/dev/hd1	Holds users' HOME directory trees
/usr	/dev/hd2	Holds Licensed Program Products (LPPs)
/tmp	/dev/hd3	Holds temporary files
/var	/dev/hd9var	Holds transient system files such as mail and news

The phys

The virtu

6.2 Fil

Files

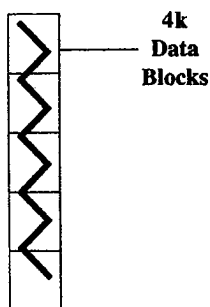


Figure 6.2 AIX 3.2 files.

The ordinary file is one type of AIX file. Other types of files include directories, symbolic links, block device special files, character device special files, named pipes (FIFOs), and sockets. Details of the structures and uses of each of these file types are provided in appropriate chapters of this book. Directories are described shortly.

The raw data of an ordinary file are stored in data blocks, as seen in Fig. 6.2. For local disk files (JFS), the data block size is 4096 bytes (4 kb). This means that the smallest disk allocation size for an ordinary file that is not empty is 4 kb. Also, the last data block of a file is always 4 kb, regardless of how much of the block is actually used. While this tends to waste disk space, especially in file systems that hold many small files (the JFS in AIX 3.2 does not support data block fragmentation), the benefit comes from the fact that file I/O is performed in 4-kb chunks via pageins and pageouts (see Chap. 4).

Inodes

Since the data blocks of an ordinary file contain nothing but raw data, the attributes of a file, such as the UID of the owner of the file, the GID of the group associated with the file, and the permission bits and file type, must be kept elsewhere. These attributes are stored in the file's inode (short for information node). Each file has an inode. Figure 6.3 illustrates the JFS inode.

The AIX header file `/usr/include/jfs/ino.h` contains the definition of the disk inode as a struct `dinode`. This file is one of the more difficult files to read for two reasons. The comments are found above the member definitions, instead of to the right of each definition, and, since there are many different types of files, the latter portion of the `dinode` structure is a complex set of nested unions and structures. An experienced C programmer should have little trouble untangling this convoluted mess, but the novice may need to spend more time on it.

Important members of the `dinode` structure include:

Directories

di_mode. A `mode_t` data type that includes the file type and permission bits

di_nlink. The number of hard links for the inode (described shortly)

di_uid and **di_gid.** The user ID of the owner and the group ID of the associated group

di_size. An `off_t` data type that holds the logical file size in bytes

di_nblocks. The number of data blocks actually used by the file

di_atime. The timestamp of the file's last access

di_mtime. The timestamp of the file's last modification

di_ctime. The timestamp of the inode's last modification

di_acl. A pointer to the file's access control list, if any (described shortly)

Author's Note: The "ctime" timestamp is wrongly commented in some UNIX-based system header files as holding the file's creation time. Rather, it holds the date and time of the last change made to the inode itself, such as changed permissions via the `chmod` command.

The remainder of the inode structure is specific to the file type. For ordinary files, this portion of the inode provides an array of logical block numbers for the data blocks of the file. A detailed explanation of how the data block addresses are maintained is found in Sec. 6.5.

The size of an AIX 3.2 inode is 128 bytes. Each file system has its own set of inodes. A file system's inodes are maintained in an array allocated to a set of data blocks within the file system. This is known as an inode table. Each inode has a number which corresponds to the inode's index within the table.

Directories

It's important to note that nowhere in the discussion of the file or the inode has the file name appeared. That's because file names are not found within the files (or at least as far as the operating system is aware), nor are they found within

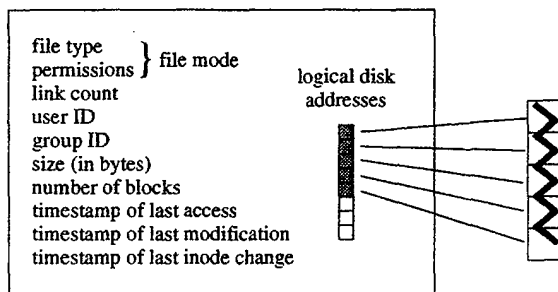


Figure 6.3 The JFS disk inode.

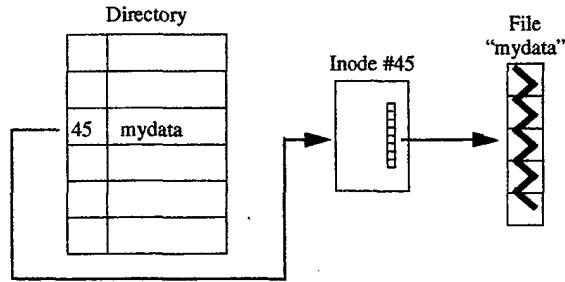


Figure 6.4 Directories and inodes.

the inode. Actually, file names exist only to provide the user with a symbolic reference to a file. The system performs all file operations based on inode numbers. File names are found only within directories.

A directory is a type of file that does have a structure recognized by the operating system. Each directory is made up of slots that hold file names and their corresponding inode numbers. In other words, a directory serves as a lookup table for converting file names to inode numbers. Figure 6.4 provides a simplified view of the relationship of a directory to an inode to a file.

Figure 6.5 shows how a complete pathname to a file is represented through directories and inodes. Since a directory is a type of file, it will have an inode which is referenced in that directory's "parent" directory. Thus a linked list, of sorts, is formed from the root directory to the file. Note that each directory contains a file named "." which represents that directory. The "." (dot) name is useful when issuing commands such as `cp /etc/motd ./mymotd`. Each directory also contains a file named ".." (dot-dot) which represents the directory's parent directory. This is used in commands such as `cd ..` to change the current directory to the parent directory. These file names are shown in Fig. 6.5.

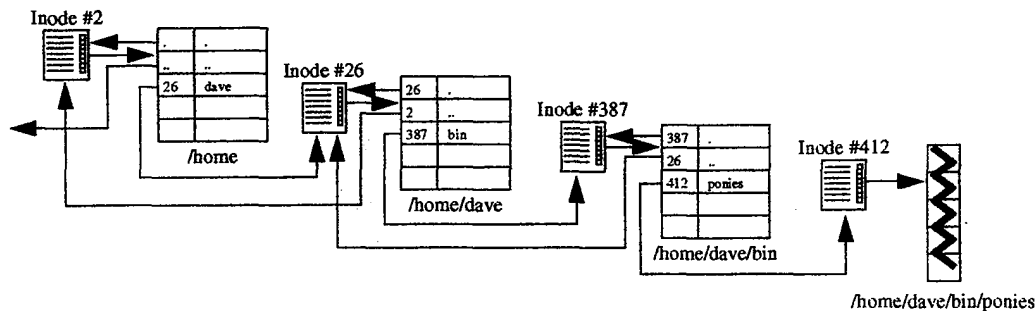


Figure 6.5 A pathname example.

Links

4 bytes	
d_ino	
47	
118	
6	n

Figure 6.6 A

The old style directory, as was used in AIX Version 2 for the RT, had a simple directory structure that consisted of two fields, a short integer for the inode number and a character array of 14 characters for the file name. The file name limit, therefore, was 14 characters. AIX 3.2 employs a directory structure similar to that of the BSD version of UNIX, which provides for variable-length file names.

The AIX 3.2 directory structure is defined as a struct `direct` in the file `/usr/include/jfs/dir.h`. The structure has four fields:

- `d_ino`. An unsigned long for the inode number
- `d_reclen`. An unsigned short for the length of the entire directory entry
- `d_namlen`. An unsigned short for the length of the file name
- `d_name[_D_NAME_MAX + 1]`. A character array for the file name

`_D_NAME_MAX` is defined in the `dir.h` file as 255 and is the size of the longest allowed file name. The `MAXNAMELEN` symbolic constant is also defined within the same file for BSD compatibility. Since AIX file names are terminated with a null character (`\0`), the `d_name[]` array is 256 characters. The `d_namlen` field contains the length of the file name. The file name is always padded up to the next 4-byte boundary. Figure 6.6 illustrates an example of a directory.

AIX allocates directory space in 512-byte blocks (see `DIRBLKSIZ` in `dir.h`). A directory consists of one or more of these blocks. Directory entries claim all of the bytes in a block. This is accomplished by having the last directory entry claim all of the remaining free bytes in the block within that entry's `d_reclen` field. When an entry is deleted from a directory, the free bytes from the deleted entry are claimed by the previous entry's `d_reclen` field. Figure 6.7 represents a before and after example. The `dir.h` header file includes a macro called `DIRSIZ` that indicates the minimum record length required to hold a directory entry. The `dir.h` header file also provides information on many library routines for manipulating directories, including `opendir()`, `readdir()`, `closedir()`, and `rewinddir()`.

Links

AIX 3.2 supports hard links and symbolic links. A hard link exists between a directory entry and an inode within the same file system. The `ln` command

4 bytes d_ino	2 bytes d_reclen	2 bytes d_namlen	4 bytes d_name[]	4 bytes d_ino	2 bytes d_reclen	2 bytes d_namlen	4 bytes d_name[]
47	12	1	.0	49	12	2	..0
118	32	21	a_very_long_file_name0	121	16		
6	mydata0	...	d_name[] 24 bytes				

Figure 6.6 A directory example.

Before:

47	12	1	..0	XX	49	12	2	..0	XX
121	16	6	mydata\0		XX	168	20	9	
inventory\0		XX	210	32	21	a_very_long_file_name\0	XX		
231	420	6	ponies\0	XX					

512 Bytes

After: `rm inventory`

47	12	1	..0	XX	49	12	2	..0	XX
121	20	6	mydata\0		XX				
		210	32	21	a_very_long_file_name\0	XX			
231	420	6	ponies\0	XX					

512 Bytes

Figure 6.7 Directory space reallocation.

allows users to create additional links for an inode by establishing additional directory entries which reference that inode. For example, two users can create links to a data file or executable file such that the file appears to exist in each of their home directories. Figure 6.8 illustrates such an example. Hard links also allow a single file to have multiple names within the same directory. For instance, AIX 3.2 links /usr/bin/sh to /usr/bin/ksh to treat the Korn shell as the default shell.

The disk inode structure (dinode) includes a field named `di_nlink` which indicates the number of hard links to the file represented by the inode. In the example in Fig. 6.8, the inode has a link count of three. When a user removes a file via the `rm` command, the directory entry for that file is deleted and the `di_nlink` value is decremented for that inode. When an inode's `di_nlink` count reaches zero, the file is considered completely removed. The inode is then placed back on the free list of inodes and the file's data blocks are released back to the free list of data blocks.

The `di_nlink` count shows up in the output from `ls -l`, as shown in Fig. 6.9. File inode numbers can be determined with the `ls -li` command. The `ncheck` command converts inode numbers to file path names. The `find` command also includes a switch for determining file names for inode numbers. Hard links can only exist between directory entries and inodes within the same file system. Symbolic links were added to UNIX to provide links between directory entries

ln /ho
ln /ho

Figure

#	
/hc	
#	
...	
-rw	
-rw	
...	
#	
...	
38'	
38'	
#	
/de	
#	
/hc	
/hc	
/hc	

Figure

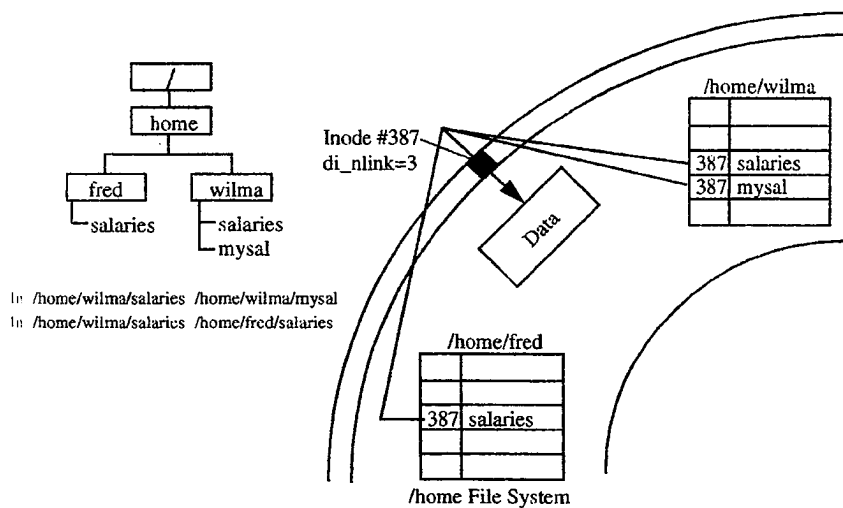


Figure 6.8 A hard link example.

```
# pwd
/home/wilma
# ls -l
...      ...      ...      ...      ...
-rw-r--  3      wilma payroll 38755  June 12 08:31  mysal
-rw-r--  3      wilma payroll 38755  June 12 08:31  salaries
...      ...      ...      ...      ...
# ls -i
...
387  mysal
387  salaries
# ncheck -i 387 /home
/dev/hd2:
/home/fred/salaries
/home/wilma/mysal
/home/wilma/salaries
# find /home -inum 387 -print
/home/fred/salaries
/home/wilma/mysal
/home/wilma/salaries
```

Figure 6.9 File names and inode numbers.

and inodes that are in different file systems. The `ln -s` command is used to create a symbolic link.

Figure 6.10 illustrates how a symbolic link is created between two file systems. The symbolic link (`/home/carolyn/bin/ponies`) is a special type of file. Its inode contains the full path name of the target file (`/usr/local/bin/ponies`). The `dinode` struct defined in `/usr/include/jfs/ino.h` includes a character array called

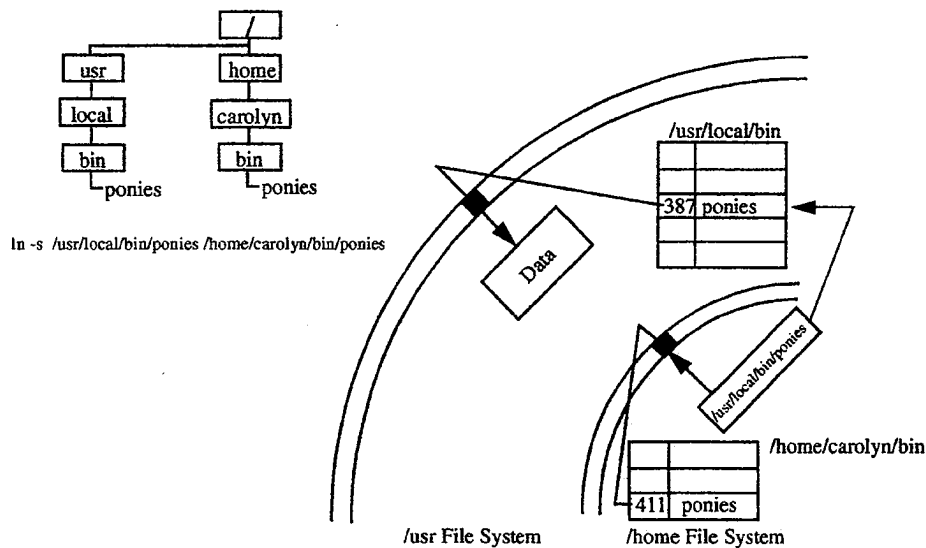


Figure 6.10 A symbolic link example.

`_s_private[D_PRIVATE]` which holds the path name of the target file as long as the path name is less than `D_PRIVATE`, which is defined as 48 characters. If the target file's path name is 48 characters or more (don't forget the null termination character), the path name is stored in a data block pointed to by the inode. This is all defined in the `di_sym` union.

The `ls -l /home/carolyn/bin` command shows the following (Fig. 6.11): The "l" character for the file type indicates the symbolic link. Notice the permissions for the symbolic link file. They are "wide open" since the target file's inode holds the actual permissions for the target file. Symbolic links do not increment the target file's link count.

Symbolic links, while necessary, include a number of potential problems. First, if the target file of a symbolic link is remove, moved, or renamed, the link is broken. For instance, if the file named `/usr/local/bin/ponies` is renamed to `/usr/local/bin/horses`, trying to execute `/home/carolyn/bin/ponies` would result in the error message "Cannot open /home/carolyn/bin/ponies." The error mes-

```
$ ls -l /home/carolyn/bin
... ..
lrwxrwxrwx 1 carolyn staff 0 July 18 09:43 ponies->/usr/local/bin/ponies
... ..
```

Figure 6.11 Symbolic links and the ls command.

sage is misleading because it does not indicate the fact that the system cannot traverse the link. Care must also be taken when executing any type of file manipulation command on a symbolic link file. The results depend on whether or not the command knows how to deal with symbolic links.

Symbolic links offer one big advantage over hard links, aside from allowing links across file systems. Symbolic links can be established for entire directories. For instance, the `/bin` directory in AIX 3.2 is actually a symbolic link to the `/usr/bin` directory. This way, not every file in the `/bin` directory has to be established as a link.

Author's Note: Some UNIX-based systems allow hard links of entire directories as long as the directories are within the same file system. AIX 3.2 does not allow this option for hard links.

Author's Note: AIX 3.2 introduced the symbolic links of the `/bin` directory to `/usr/bin` and the `/lib` directory to `/usr/lib` to help maintain backward compatibility with AIX 3.1. Most of the `/bin` and `/lib` directory contents were moved to `/usr/bin` and `/usr/lib`, respectively, in AIX 3.2 to reduce the size of the root file system for support of diskless workstations.

Finally, if a user moves from the parent directory of a symbolically linked directory, down through the symbolically linked directory, then references the `..` directory name or issues the `pwd` command, the results can be surprising. For instance, consider the example in Fig. 6.12. If a user starts at `/home/carolyn` and moves to `/home/carolyn/lib` (which is really `/usr/local/lib`), then issues the `pwd` command, the output is `/usr/local/lib` if the user is using the Bourne shell or

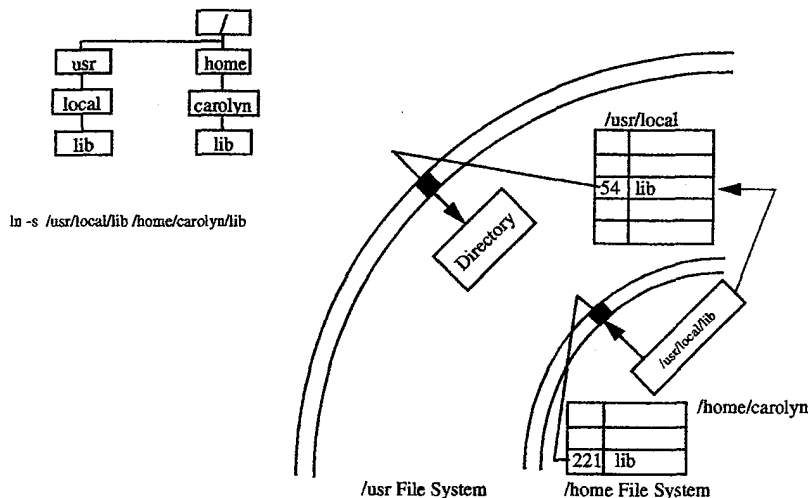


Figure 6.12 Symbolic links and directories.

the C shell. If the user is using the Korn shell, however, the output is `"/home/carolyn/lib"`! Similar results occur when the user issues a `"cd .."` command.

Access control lists

AIX 3.2 implements access control lists (ACLs) as a way of providing extended permission capability. The owner of a file can issue the commands `acledit`, `aclget`, and `aclput` to look at or modify the extended permissions of that file. (See the manual pages or InfoExplorer for more details on these commands.) Since the access control lists allow permission sets to be created for any user or group, additional space is required within the file's control data structure to hold this information. To avoid increasing the size of the disk inode, since ACLs might only be applied to a small number of files on most systems, IBM created a structure called an extended inode. The extended inode is an additional amount of disk space allocated only for files that require it. The dinode structure defined in `/usr/include/jfs/ino.h` has a field called `di_acl` which points to the data block holding the file's extended inode. If the value of `di_acl` is null, the file has no extended inode. The header file `/usr/include/sys/acl.h` defines the structures used to maintain ACLs. The header file lacks comments, but the structures are easy to understand once one understands the application of ACLs (see the manual page for `acledit`).

6.4 TI

6.3 The Journaled File System (JFS)

The JFS represents one of IBM's finest contributions to the open systems marketplace, although it is not without some controversy. The JFS applies a data base logging approach to file system control structures. In this way, if a system crashes while these control structures are being updated, a log redo utility allows the file system to be returned to a known state. It is important to understand that the JFS does not log changes made to user data. Therefore, a system crash might still result in the loss of user data. The JFS attempts to assure that the file system maintains its integrity through a crash.

Prior to the JFS, a system administrator relied on tools such as `fsck` or `fsdb` to fix a corrupted file system. The `fsck` utility looks for inodes that have non-zero link counts yet are not claimed by any directory entry, or data blocks which are not on the free list yet are not claimed by any inode. When `fsck` finds such "orphans," it has little choice but to place the inodes (files) or data blocks (assigned to a file by `fsck`) into the lost+found directory located in the root directory of the file system. Finding the owners of the contents of the lost+found directory after running `fsck` relies upon the UNIX savvy of the system administrator and is usually not an easy task.

Incidentally, AIX 3.2 includes the `fsck` utility which still can be used to check the integrity of a file system. In fact, if the JFS log redo fails for any reason, `fsck` may be the only way to fix a corrupted file system, short of restoring from a backup. AIX 3.2 also offers the `fsdb` (file system debugger), which allows a user to examine and change the data found in file system control structures.

Each journaled file system within a volume group (see Sec. 2.3 for information on volume groups) usually shares a common JFS log. The JFS log for the root volume group is in the /dev/hd8 logical volume. It is a 4-megabyte circular log which is updated by the operating system at regular intervals. The details of the JFS logging are described in Sec. 6.6.

One unfortunate aspect of the JFS is that it is not supported for floppy diskette file systems.

6.4 The JFS Architecture

As previously mentioned, the JFS design is similar to other UNIX-type file systems. It includes a boot block, a super block, inodes, indirect blocks, and data blocks. Anyone familiar with other types of UNIX file systems might be tempted to skip this section of the book. However, the JFS supplies a few interesting twists to the traditional UNIX file system paradigm. This section explores those nuances. It also serves as an introduction to the fundamental concepts of file systems.

Before describing the design on the JFS it's necessary to say a few things about how AIX 3.2 manages disk space. As explained in Chap. 2, the LVM carves all physical disk space up into contiguous physical partitions. The default size of these partitions is 4 megabytes on most RISC System/6000 models. Some smaller RISC System/6000 models use a 2-megabyte default physical partition size. In any case, the size of a volume group's physical partitioning is constant throughout the volume group and can be set by the system administrator when the volume group is created. IBM recommends 4 megabytes as the optimal size.

When a journaled file system is created, the system allocates it to one or more physical partitions. This means that the smallest file system that can be created in a volume group is equal to the size of the physical partitioning of that volume group. Figure 6.13 shows a volume group with 4-megabyte physical partitions. It also shows a couple of 4-megabyte file systems and a 12-megabyte file system. Another nice feature of the JFS (with help from the LVM) is the ability to easily extend a file system. When a file system becomes full, the system administrator

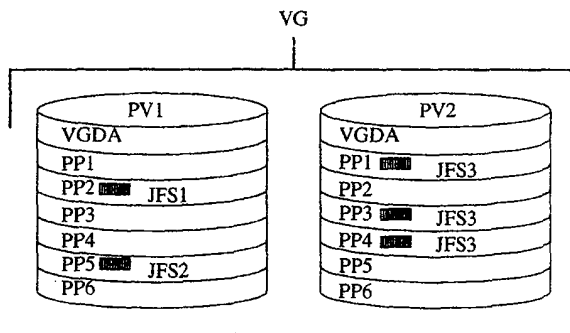


Figure 6.13 The JFS and the LVM.

The JF:

Each JFS consists of the following components, as illustrated in Fig. 6.14:

Block 1. The super block, which contains control information for the entire file system.

Block 31. The spare super block copy, used in the event of a corrupted super block. While many UNIX file systems allocated a large number of spare super blocks, the JFS only allocates one. This is because the JFS logging makes it less likely for the super block to become corrupted. Also, as described shortly, the information found in the JFS super block is not as important to the integrity of the file system as the information found in the super block of other UNIX file systems.

Blocks 32 through 63. The inode table for this allocation group.

Blocks 2 through 31, and 64 through the end of the file system. Data blocks.

Name _____
 4th 1st Nodes _____
 Some One _____
 Reserved _____

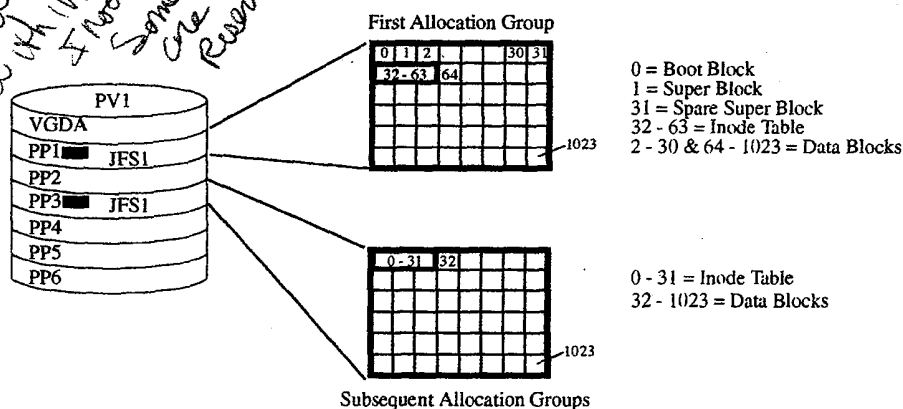


Figure 6.14 JFS block allocation.

The ing

The JFS super block

Block one of the JFS is the super block for that file system. The structure of the super block is defined in the header file `/usr/include/jfs/filsys.h`. The JFS super block is not as “super” as its counterparts in other UNIX-based file systems. Traditionally, the super block contains pointers to the linked lists of free inodes and data blocks for the file system it represents. For this reason, if the super block became corrupted, the file system was trashed. This is why many UNIX-based file systems have spare super block copies scattered throughout the file system. As you will see shortly, IBM has implemented another method of maintaining inode and data block-free lists. The method does not rely on the JFS super block.

The JFS super block contains a few interesting fields, such as:

`s_fsize`. The file system size (in 512-byte blocks). AIX 3.2 supports a maximum file system size of 2 gigabytes.

`s_bsize`. The block size for this file system.

`s_fname[]`. The name of the file system.

`s_logdev`. A `dev_t` type (device major and minor numbers) for the JFS log of this file system.

`s_ronly`. A character field set if the file system is mounted as read-only.

`s_time`. A `time_t` type that holds the timestamp of the last super block update.

`s_fmod`. A character flag that indicates the state of the file system. Values include:

0. File system is clean and unmounted.
1. File system is mounted.
2. File system was mounted when dirty or commit failed.
3. Log redo processing attempted but failed.

Beyond these fields, the JFS super block isn't very interesting.

The inode table and inode allocation

Blocks 32 through 63 of a JFS hold the inode table for the first allocation group. The JFS in AIX 3.2 dynamically allocates another inode table for each new allocation group when the file system is extended. The inode table occupies blocks 0 through 31 of each allocation group after the first allocation group. The JFS attempts to assign data blocks for inodes from the same allocation group whenever possible. Unlike traditional UNIX-based file systems that allow the system administrator to specify the number of inodes for a file system when the file system is created, the JFS in AIX 3.2 defaults to one inode per data block

within the allocation group. Since the size of an allocation group is 4 megabytes, and each data block is 4 kilobytes, the JFS creates 1024 inodes for each allocation group. This is illustrated in Fig. 6.14.

In theory, this inode allocation scheme should mean that a file system will never run out of inodes as long as there are still data blocks because of the one-to-one ratio. In practice, however, it is possible to run out of inodes and still have data blocks left over because zero-length files, such as symbolic links and device special files, require an inode without ever allocating a data block.

Author's Note: The latter example occurred for one of my students who was working with AIX 3.2 on IBM's SP/2 system. She needed to define a large number of device special files in the /dev directory, only to find that the system ran out of inodes in the root file system. The only solution was to extend the root file system to create more inodes.

Reserved inodes

The first 16 inodes of every JFS file system (inodes 0 through 15) are reserved by the JFS. A description of each reserved inode is found in the comments of the /usr/include/jfs/filsys.h header file. Most of these reserved inodes have file names that begin with a "." (dot) character because they are hidden files, but unlike the hidden files found throughout the system directories, these files are "really hidden," as they do not appear in any directory. This is accomplished by handcrafting the inodes so that they do not require a directory entry to support their link count value. Remember that every open file is represented by a segment in the VMM (see Chap. 4). Many of the reserved inodes are associated with files that are only present in the VMM when a file system is mounted and never actually exist on disk.

Author's Note: The /usr/include/jfs/filsys.h header file erroneously comments that inodes 9–16 are reserved for future use, when actually inodes 9–15 are reserved. This error can be tested by creating a new file system and creating a file within the new file systems. The first file created should have an inode number of 16. This just goes to show that you can't always trust the comments!

Inode 0 of the JFS is never used.

Root directory. Inode 2 of the JFS is always used for the root directory of the file system. An interesting thing happens if one performs the `ls -la` command in the root file system, then compares the output with that of an `ls -la` command in the /home directory. The inode number for /home within the root file system is 2. In other words, since the file system is mounted, the `ls` command reports the /home directory as the root of the /home file system. In addition, the inode number of "." in the /home directory is also 2 since that is the inode for the root directory of the /home file system, not the /home mount point.

128
Buh

.superblock. Inode 1 of the JFS is reserved for a file named `.superblock`. This virtual file simply refers to the super block and spare super block. Examining the inode reveals that the file consists of two data blocks, blocks 1 and 31.

.inodes. Inode 3 of the JFS is reserved for a file named `.inodes`. This file keeps track of all file system data blocks being used to hold inodes. This is similar to a disk or cylinder group map in other UNIX-based file systems.

.indirect. Inode 4 of the JFS is reserved for a virtual file named `.indirect`. The VMM uses this file to map the pages of indirect blocks for the entire file system. An explanation of indirect blocks is given shortly.

.inodemap. Inode 5 of the JFS is reserved for a virtual file named `.inodemap`. This file takes the place of the traditional linked list of free inodes via a bit map where each inode of the file system is represented by a bit flag. When an inode is in use (`di_nlink > 0`), the bit flag is turned on for the corresponding bit in the map. When a new file is created in the file system and a new inode is needed, the file system scans the `.inodemap` segment for the first bit flag that is turned off. The bit flag is turned on and the corresponding inode is assigned to the new file. This technique is much faster than manipulating linked lists.

.diskmap. Inode 6 of the JFS is reserved for a virtual file named `.diskmap`. As the `.inodemap` file keeps track of the free and allocated inodes, the `.diskmap` file keeps track of free and allocated data blocks within the file system. The `.diskmap` file, however, is not a simple bit map. The file uses a set of hash buckets built on a binary tree principle to point to chains of contiguous free data blocks. This way, when a new file is allocated with a known size, such as when a large file is copied to another file, the JFS can search for a large enough set of contiguous data blocks to allow the file to be stored with the least amount of fragmentation.

.inodex. Inode 7 of the JFS is reserved for a virtual file named `.inodex`. The file contains information about inode extensions, as used by access control lists (see Sec. 6.2).

.inodexmap. Inode 8 of the JFS is reserved for a virtual file named `.inodexmap`. The file contains a bit map used to keep track of free and allocated inode extensions. It is similar to the `.inodemap` file.

Many of these virtual files are created when the file system is mounted.

As mentioned earlier, inodes 9 through 15 of the JFS are reserved for future use. Figure 6.15 recaps the reserved inodes.

0	Not Used
1	.superblock
2	Root directory of the File System
3	.inodes
4	.indirect
5	.inodemap
6	.diskmap
7	.inodex
8	.inodexmap
9-15	Reserved

Figure 6.15 Reserved inodes.

6.5 JFS Storage Schemes

The earlier discussion of inodes alluded to the fact that a file's inode contains the logical disk addresses for the data blocks of that file. One might wonder how the inode, which is only 128 bytes, can hold enough logical disk addresses to accommodate large files. This is done through a series of storage schemes, each designed to efficiently handle files of various sizes. While the JFS method of addressing data blocks is similar to the direct, single indirect, double indirect, and triple indirect schemes used by many UNIX file systems, it differs slightly in its implementation.

The disk inode contains an array of eight logical addresses for the first eight data blocks of a file. The term "logical address" refers to the data blocks' numbers within the file system. The actual disk addresses (i.e., physical sector numbers) are derived by the logical volume manager during disk I/O. The array of eight direct block pointers, called `_di_raddr[NDADDR]`, is defined within the dinode structure of the `/usr/include/jfs/ino.h` header file. NDADDR is defined by AIX 3.2 as eight. Each pointer holds the logical block number for a 4-kilobyte data block. This scheme, as shown in Fig. 6.16, supports a file size of 32 kb. Since support for larger files is required, a file with a size of >32 kb must

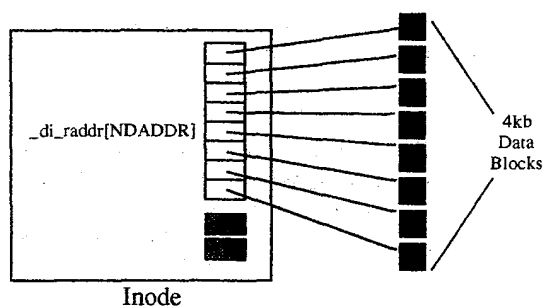


Figure 6.16 Direct data block accessing scheme.

_di_vi
_di_ri

Figure 6.1

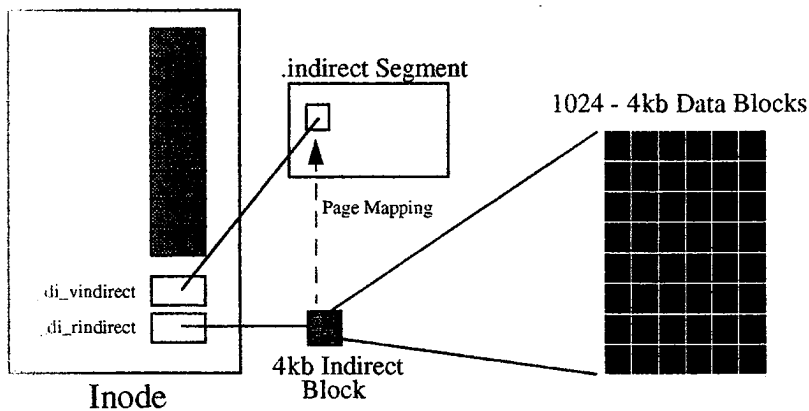


Figure 6.17 Single-indirect data block accessing scheme.

resort to another storage scheme. Figure 6.17 shows how the JFS implements single indirect access. Each inode contains a field named `_di_rindirect`. This field holds the logical disk address of the file's indirect block. An indirect block is a 4-kilobyte data block that has been converted, by the JFS, to hold up to 1024 4-byte logical disk addresses for data blocks. This allows a file to access up to 4 megabytes of space.

To speed up access to files that use indirect addressing, the JFS works with the VMM to map the indirect block into a page of a VMM segment called `.indirect`. There is a single `.indirect` virtual file created for each mounted file system. Each page of the `.indirect` segment can hold an indirect block from disk. All files larger than 32 kb within the same file system share the use of the `.indirect` file.

Each inode that has an indirect block uses a field named `_di_vindirect` to indicate the page number of the memory mapped indirect block within the `.indirect` segment. For files larger than 4 Mb, the JFS uses a third scheme, known as the double indirect block. In this case, the inode's `_di_rindirect` field contains the logical disk address of a double indirect block. A double indirect block is a 4-kilobyte data block that has been converted to hold up to 512 8-byte values, each of which consists of two 4-byte pointers. One of the 4-byte pointers from each 8-byte pair holds the logical disk address of an indirect block. The other 4-byte pointer from each 8-byte pair holds the page number of where its corresponding indirect block has been mapped in the `.indirect` segment. This scheme allows the double indirect block to access up to 512 indirect blocks, each of which can access up to 1024 4-kb data blocks. This provides a maximum access size of 2 gigabytes ($512 * 1024 * 4096$). Since 2 gigabytes is the largest supported file system in AIX 3.2, this scheme works perfectly. Figure 6.18 shows how a very large file (>4 Mb) might be accessed by the JFS.

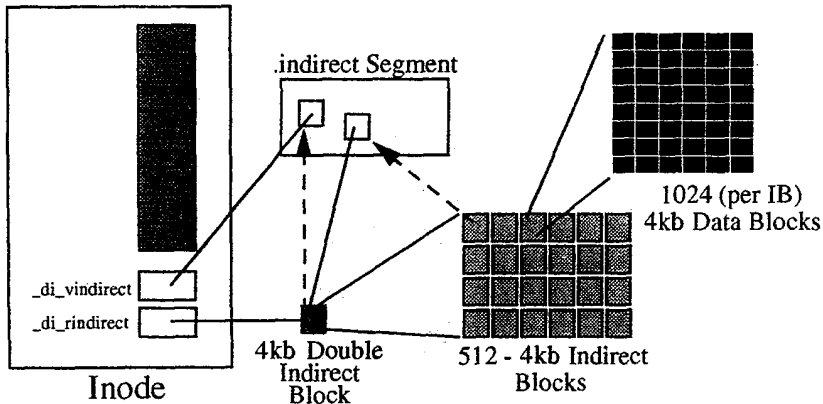


Figure 6.18 Double-indirect data block accessing scheme.



Author's Note: One might be tempted to think that the address scheme described above is the reason that AIX 3.2 has a file size limit of 2 Gb. Actually, the reason for the 2-Gb file size limit is more historical than technical. The field called `di_size` in the `dinode` structure defined in `/usr/include/jfs/ino.h` is a typedef of `off_t`. The `off_t` is defined in `/usr/include/sys/types.h` as a signed long, which means that one of the 32 bits is used to show the sign of the number, leaving 31 bits to hold the number itself; 31 bits can access up to 2 Gb. Your next questions might be "Why is the `off_t` a signed long? When would I have a negative file size?" The `off_t` is also used to define the read/write offset within a file (see the manual page for the `lseek()` subroutine). Since it is possible to seek backward through a file, negative values had to be supported. At the time this strategy was created, no one could implement 2-Gb file sizes because the hardware didn't support it. (Remember, it wasn't that long ago that we were all paying big bucks for 10 Mb hard drives!) Therefore, a maximum file size of 2 Gb seemed more than adequate. Most open system vendors have implemented or are considering schemes that extend the maximum file size beyond 2 Gb. AIX Version 4.2 will support 64-Gb maximum file and file system sizes.

The actual layout of the `.indirect` segment is not as simple as the example given above. The `.indirect` segment is described through comments found in the `/usr/include/jfs/ino.h` header file.

6.6 How the JFS Log Works

As mentioned previously, the goal of the journaled file system is to provide a more robust file system by logging changes made to its own structures and lists. This includes changes made to the file system super block, the inodes, directories, indirect blocks, and free lists of inodes and data blocks.

Author's Note: The term "free list" is used figuratively here. Recall that the JFS uses bit maps to track free and allocated inodes and data blocks.

Compor

A JFS e

Whenever a JFS is mounted, the AIX kernel verifies its consistency by examining the log records of that file system. The log records show transactions that were completed (committed), as well as, in the case of a file system crash, incomplete transactions. The JFS reconstructs the committed transactions, bringing the file system structures up to date. Incomplete transactions are discarded, since they would leave the file system structures in a "half-baked" state.

JFS logging is similar to the way many data base programs log transactions. It's important to remember, however, that the JFS does not log the user data bound for the file system data blocks. In other words, when a system crash occurs or someone shuts off the power to the computer without properly bringing down the operating system, data in memory which have not yet been written to disk are lost. The JFS log will make it possible to recover the file system control data, but the user data are not recoverable.

Components of the JFS

The JFS uses a physical disk partition (usually 4 megabytes in size) as a log device. Each volume group (see Chap. 2 for a discussion of volume groups and the logical volume manager) must have a JFS log device. The rootvg's log device is /dev/hd8. A JFS's log device can be specified at the time the file system is created.

The JFS also maintains a log segment (a 256-megabyte segment) in virtual memory for each log device. Pages of this segment are written to the disk log device at regular intervals. The JFS also maintains an inode manager and a lock manager to ensure that in-core inodes are locked while they are updated. (In-core inodes are detailed in Chap. 7.)

A JFS example

To illustrate how the JFS logs changes made to the file system structures, an example of the `mkdir` command is used. The following list includes just a few of the events that take place in the JFS when a new directory is created by a user:

✍ An inode is allocated for the new directory. The `.inodemap` segment is updated by setting the bit that represents the inode.

An entry is made for the new directory in the parent directory's data block. The new directory's inode number is associated with the new directory's name. This operation may involve adding a new directory block to the directory.

A data block is allocated to the new directory. The `.diskmap` segment is updated.

The `"."` and `".."` file names are added to the new directory.

The link count is incremented for the parent directory's inode (due to the `".."` file name in the new directory).

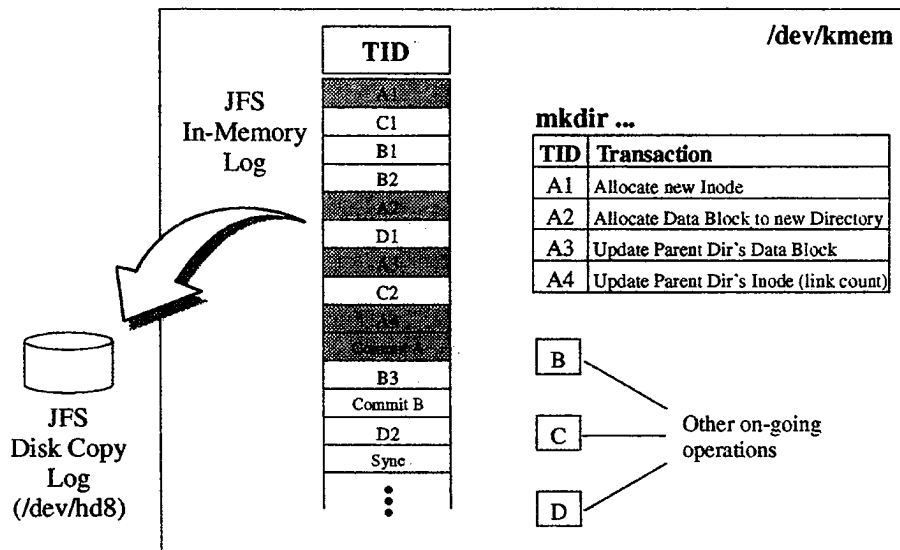


Figure 6.19 A JFS logging example.

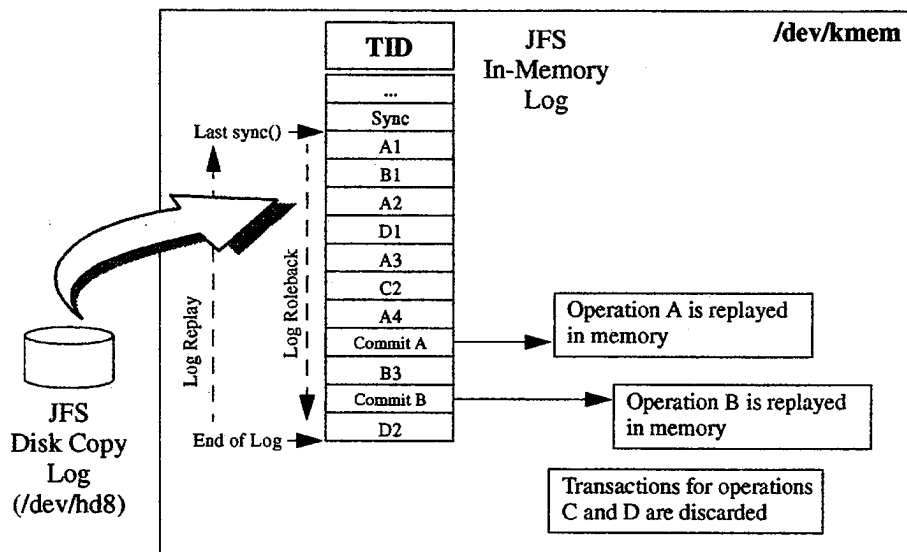


Figure 6.20 The JFS log redo.

The link count is incremented in the new directory's inode (due to the "." file name it now contains).

Author's Note: That's quite a bit of activity, isn't it? On most UNIX-based systems, if the system crashed while in the middle of performing these tasks, the file system would end up in a corrupted state. The JFS attempts to assure that these activities are performed atomically.

It's important to understand that the events listed above happen in memory and are then written to disk some time later. Generally, user data and file system control data are written to disk whenever a sync occurs. The JFS is able to confirm that this has happened by writing a sync record to the log.

Author's Note: Many UNIX-based systems use a sync daemon, which writes all modified file pages and file system data to disk every 30 seconds. The AIX 3.2 syncd process performs a sync once every 60 seconds, by default. The sync time can be changed. The syncd daemon is launched from the /sbin/rc.boot shell script at system start-up. A system administrator can change the parameter to the syncd program from within this script.

As the new directory is being created, the JFS logs each transaction (each event from the list above) in the log segment, as the activity takes place in memory. Once all transactions have been completed in memory and recorded to the log segment, the JFS writes a commit record to the log segment. As mentioned earlier, the log segment pages are written to the disk-based log device at regular intervals. Figure 6.19 illustrates the example so far.

When a file system is mounted, the JFS checks the log entries for the file system. If the log does not end with a sync record, the JFS concludes that the file system was not unmounted cleanly. The JFS performs a log redo, searching back through the log for the previous sync record. The JFS does not care about any transaction above this sync record since the sync caused all changes to be written to disk.

The JFS then performs all transactions for which commit records exist. Any transactions that do not have a commit record are discarded. This should bring the file system structures to a complete and known state. Figure 6.20 illustrates the log redo procedure.

Author's Note: While the JFS is not foolproof, I can attest to the improved file system reliability. I have tried, on occasion, to corrupt a JFS file system by doing things like powering down in the middle of copying a large file. While I have experienced a loss of user data, I have not corrupted a file system. I am not advocating, however, that you try this on your system!