Home | News | Products | Services | Solutions | About IBM
ShopIBM Support Download

Search ☐ Go

IBM : developerWorks : Linux overview | Open Source : Library - papers

JFS overview
How the Journaled File System cuts system restart times to the quick

Steve Best, IBM
January 2000

JFS provides fast file system restart in the event of a system crash. Using database journaling techniques, JFS can restore a file system to a consistent state in a matter of seconds or minutes, versus hours or days with non-journaled file systems. This white paper gives an overview of the architecture, and describes design features, potential limits, and administrative utilities of the JFS technology available on developerWorks.

Contents:

The Journaled File System (JFS) provides a log-based, byte-level file system that was developed for transaction-oriented, high performance systems. Scalable and robust, its advantage over non-journaled file systems is its quick restart capability: JFS can restore a file system to a consistent state in a matter of seconds or minutes.

While tailored primarily for the high throughput and reliability requirements of servers (from single processor systems to advanced multi-processor and clustered systems), JFS is also applicable to client configurations where performance and reliability are desired.

Architecture and design

The JFS architecture can be explained in the context of its disk layout characteristics.

Logical volumes

The basis for all file system discussion is a logical volume of some sort. This could be a physical disk or some subset of the physical disk space such as an FDISK partition. A logical volume is also known as a disk partition.

Aggregates and filesets

The file system create utility, mkfs, creates an aggregate that is wholly contained within a partition. An aggregate is an array of disk blocks containing a specific format that includes a superblock and an allocation map. The superblock identifies the partition as a JFS aggregate, while the allocation map describes the allocation state of each data block within the aggregate. The format also includes the initial fileset and control structures necessary to describe it. The fileset is the mountable entity.

Files, directories, inodes, and addressing structures

A fileset contains files and directories. Files and directories are represented persistently by inodes; each inode describes the attributes of the file or directory and serves as the starting point for finding the file or directory's data on disk. JFS also uses inodes to represent other file system objects, such as the map that describes the allocation state and location on disk of each inode in the fileset.

Directories map user-specified names to the inodes allocated for files and directories and form the traditional name hierarchy. Files contain user data, and there are no restrictions or formats implied in the data. That is, user data is treated, by JFS, as an uninterpreted byte stream. Extent-based addressing structures rooted in the inode are used for mapping file data to disk. Together, the aggregate superblock and disk allocation map, file descriptor and inode map, inodes, directories, and addressing structures represent JFS control structures or meta-data.

Logs

JFS logs are maintained in each aggregate and used to record information about operations on meta-data. The log has a format that also is set by the file system creation utility. A single log may be used simultaneously by multiple mounted filesets within the aggregate.

Design features

JFS was designed to have journaling fully integrated from the start, rather than adding journaling to an existing file system. A number of features in JFS distinguish it from other file systems.

Journaling

JFS provides improved structural consistency and recoverability and much faster restart times than non-journaled file systems such as HPFS, ext2, and traditional UNIX file systems. These other file systems are subject to corruption in the event of system failure since a logical write file operation often takes multiple media I/Os to accomplish and may not be totally reflected on the media at any given time. These file systems rely on restart-time utilities (that is, fsck), which examine all of the file system's meta-data (such as directories and disk addressing structures) to detect and repair structural integrity problems. This is a time-consuming and error-prone process, which, in the worst case, can lose or misplace data.

In contrast, JFS uses techniques originally developed for databases to log information about operations performed on the file system meta-data as atomic transactions. In the event of a system failure, a file system is restored to a consistent state by replaying the log and applying log records for the appropriate transactions. The recovery time associated with this log-based approach is much faster since the replay utility need only examine the log records produced by recent file system activity rather than examine all file system meta-data.

Several other aspects of log-based recovery are of interest. First, JFS only logs operations on meta-data, so replaying the log only restores the consistency of structural relationships and resource allocation states within the file system. It does not log file data or recover this data to consistent state. Consequently, some file data may be lost or stale after recovery, and customers with a critical need for data consistency should use synchronous I/O.

Logging is not particularly effective in the face of media errors. Specifically, an I/O error during the write to disk of the log or meta-data means that a time-consuming and potentially intrusive full integrity check is required after a system crash to restore the file system to a consistent state. This implies that bad block relocation is a key feature of any storage manager or device residing below JFS.

JFS logging semantics are such that, when a file system operation involving meta-data changes -- that is, unlink() -- returns a successful return code, the effects of the operation have been committed to the file system and will be seen even if the system crashes. For example, once a file has been successfully removed, it remains removed and will not reappear if the system crashes and is restarted.

The logging style introduces a synchronous write to the log disk into each inode or vfs operation that modifies meta-data. (For the database mavens, this is a redo-only, physical after-image, write-ahead logging protocol using a no-steal buffer policy.) In terms of performance, this compares well with many non-journaling file systems that reply upon (multiple) careful synchronous meta-data writes for consistency. However, it is a performance disadvantage when compared to other journaling file systems, such as Veritas VxFS and Transarc Episode, which use different logging styles and lazily write log data to disk. In the server environment, where multiple concurrent operations are performed, this performance cost is reduced by group commit, which combines multiple synchronous write operations into a single write operation. JFS logging style has been improved over time and now provides asynchronous logging, which increases performance of the file system.

Extent-based addressing structures
JFS uses extent-based addressing structures, along with aggressive block allocation policies, to produce compact, efficient, and scalable structures for mapping logical offsets within files to physical addresses on disk. An extent is a sequence of contiguous blocks allocated to a file as a unit and is described by a triple, consisting of <logical offset, length, physical>. The addressing structure is a B+tree populated with extent descriptors (the triples above), rooted in the inode and keyed by logical offset within the file.

Variable block size
JFS supports block sizes of 512, 1024, 2048, and 4096 bytes on a per-file system basis, allowing users to optimize space utilization based on their application environment. Smaller block sizes reduce the amount of internal fragmentation within files and directories and are more space efficient. However, small blocks can increase path length since block allocation activities may occur more often than if a large block size were used. The default block size is 4096 bytes since performance, rather than space utilization, is generally the primary consideration for server systems.

Dynamic disk inode allocation
JFS dynamically allocates space for disk inodes as required, freeing the space when it is no longer required. This support avoids the traditional approach of reserving a fixed amount of space for disk inodes at the file system creation time, thus eliminating the need for users to estimate the maximum number of files and directories that a file system will contain. Additionally, this support decouples disk inodes from fixed disk locations.

Directory organization
Two different directory organizations are provided. The first organization is used for small directories and stores the directory contents within the directory's inode. This eliminates the need for separate directory block I/O as well as the need to allocate separate storage. Up to 8 entries may be stored in-line within the inode, excluding the self(.) and parent(..) directory entries, which are stored in separate areas of the inode.

The second organization is used for larger directories and represents each directory as a B+tree keyed on name. It provides faster directory lookup, insertion, and deletion capabilities when compared to traditional unsorted directory organizations.

Sparse and dense files
JFS supports both sparse and dense files, on a per-file system basis.

Sparse files allow data to be written to random locations within a file without instantiating previously unwritten intervening file blocks. The file size reported is the highest byte that has been written to, but the actual allocation of any given block in the file does not occur until a write operation is performed on that block. For example, suppose a new file is created in a file system designated for sparse files. An application writes a block of data to block 100 in the file. JFS will report the size of this file as 100 blocks, although only 1 block of disk space has been allocated to it. If the application next reads block 50 of the file, JFS will return a block of zero-filled bytes. Suppose the application then writes a block of data to block 50 of the file. JFS will still report the size of this file as 100 blocks, and now 2 blocks of disk space have been allocated to it. Sparse files are of interest to applications that require a large logical space but only use a (small) subset of this space.

For dense files, disk resources are allocated to cover the file size. In the above example, the first write (a block of data to block 100 in the file) would cause 100 blocks of disk space to be allocated to the file. A read operation on any block that has been implicitly written to will return a block of zero-filled bytes, just as in the case of the sparse file.

Internal JFS (potential) limits

JFS is a full 64-bit file system. All of the appropriate file system structure fields are 64-bits in size. This allows JFS to support both large files and partitions.

File system size

The minimum file system size supported by JFS is 16 Mbytes. The maximum file system size is a function of the file system block size and the maximum number of blocks supported by the file system meta-data structures. JFS will support a maximum file size of 512 terabytes (with block size 512 bytes) to 4 petabytes (with block size 4 Kbytes).

File size

The maximum file size is the largest file size that virtual file system framework supports. For example, if the frame work only supports 32-bits, then this limits the file size.

Removable media

JFS will not support diskettes as an underlying file system device.

Standard administrative utilities

JFS provides standard administration utilities for creating and maintaining file system.

Create a file system

This utility provides the JFS-specific portion of the mkfs command, initializing a JFS file system on a specified drive. This utility operates at a low level and assumes any creation/ initialization of the volume on which the file system is to reside is handled outside of this utility at a higher level.

Check/recover a file system

This utility provides the JFS-specific portion of the fsck command. It checks the file system for consistency and repairs problems discovered. It replays the log and applies committed changes to the file system meta-data. If the file system is declared clean as a result of the log replay, no further action is taken. If the file system is not deemed clean, indicating that the log was not replayed completely and correctly for some reason or that the file system could not be restored to a consistent state simply by replaying the log, then a full pass of the file system is performed.

In performing a full integrity check, the check/repair utility's primary goal is to achieve a reliable file system state to prevent future file system corruption or failures, with a secondary goal of preserving data in the face of corruption. This means the utility may throw away data in the interest of achieving file system consistency. Specifically, data is discarded when the utility does not have the information needed to restore a structurally inconsistent file or directory to a consistent state without making assumptions. In the case of an inconsistent file or directory, the entire file or directory is discarded with no attempt to save any portion. Any file or sub-directories orphaned by the deletion of the corrupted directory are placed in the lost+found directory located at the root of the file system.

An important consideration for a file system check/repair utility is the amount of virtual memory it requires. Traditionally, the amount of virtual memory required by these utilities is dependent on file system size, since the bulk of the required virtual memory is used to track the allocation state of the individual blocks in the file system. As file systems grow larger, the number of blocks increases and so does the amount of virtual memory needed to track these blocks.

The design of the JFS check/repair utility differs in that its virtual memory requirements are dictated by the number of files and directories (rather than the number of blocks) within the file system. The virtual memory requirements for the JFS check/repair utility are on the order of 32 bytes per file or directory, or approximately 32 Mbytes for a file system that contains 1 million files and directories, regardless of the file system size. Like all other file systems, the JFS utility needs to track block allocation states but avoids using virtual memory to do so by using a small reserved work area located within the actual file system.

Summary

JFS is a key technology for Internet file servers since it provides fast file system restart times in the event of a system crash. Using database journaling techniques, JFS can restore a file system to a consistent state in a matter of seconds or minutes. In non-journaled file systems, file recovery can take hours or days. Most file server customers cannot tolerate the downtime associated with non-journaled file systems. Only by a technology shift to journaling could these file systems avoid the time-consuming process of examining all of a file system's meta-data to verify/restore the file system to a consistent state.

Resources

- JFS open source, on developerWorks
- IBM makes JFS technology available for Linux , dW feature story

About the author

Steve Best works in the Software Solutions & Strategy Division of IBM in Austin, Texas as a member of the File System development department. Steve has worked on operating system development in the areas of the file system, internationalization, and security. Steve is currently working on the port of JFS to Linux. He can be reached at sbest@us.ibm.com.

What do you think of this article?

○ Killer! ○ Good stuff ○ So-so; not bad ○ Needs work ○ Lame!

Comments?

Submit feedback