Filesystem Enhancements in AIX 4.1

By Bill Baker

Did you ever wonder why small files consumed so much disk space in the AIX Journaled File System (JFS)? Have you ever tried to decrease the block size in JFS? If so, you will be pleased with AIX Version 4. By supporting fragments, the new JFS enables small files to be allocated more efficiently. For even more space savings, JFS is now able to compress files. In addition, the JFS supports filesystems well in excess of 2 GB.

AIX Version 4 introduces several significant filesystem enhancements in the JFS. These enhancements include support of fragments, capacity to compress files, and new 64-bit integers in the compiler that help to relax the former maximum filesystem size of 2 GB.

JFS Fragments

In AIX Version 3, the JFS had a fixed block size of 4 KB. This design made the JFS much easier to integrate into the pager and avoided the classic problem of free space fragmentation. However, in environments with many small files, such as news servers, large fixed-size blocks can result in wasted space.

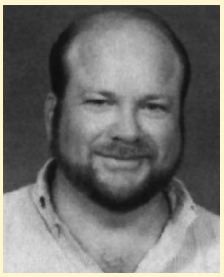
In AIX Version 4, JFS allows disk space to be allocated in smaller blocks known as *fragments*. These partial blocks improve space efficiency for files smaller than the fundamental block size of the filesystem, without losing the performance advantage of large blocks. Fragments have long been implemented in the Berkeley Software Distribution Fast Filesystem[1].

The key to using fragments to reduce space inefficiency of small files lies in partially allocating the last block of a file. On average, the last block of every file is half-wasted. If every file in a filesystem is less than a full block, then 50% of the filesystem space is wasted. However, with fragments, all but the last block have full allocations of contiguous fragments; the last block is only partially allocated.

The partial allocation of the last block of a file or directory occurs when the file has direct geometry—data block addresses in the inode. Since the JFS stores eight direct addresses in the inode, only files less than 32 KB will have their last block fragmented. Files larger than 32 KB will have full block allocations, even in their last block.

For example, on a filesystem with 512-byte fragments, a 4,100-byte file will consume nine fragments. The first logical block of the file will consume eight contiguous fragments. The second logical block, consisting of the four remaining bytes, will consume just one fragment. In AIX Version 3, this second logical block would have consumed another full block.

Partial allocation implies that each block address slot in the inode must be able to represent the number of fragments actually allocated for that slot. Since the maximum full-block-to-fragment ratio is eight 512-byte fragments, each block address slot needs three additional bits to store a partial allocation. These bits are encoded in the upper nibble of the disk block address. The most significant bit is unused. The next three



Bill Baker

[1] McKusick, Marshall Kirk; Joy, William N.; Leffler, Samuel J.; and Fabry, Robert S. "A Fast File System for UNIX," *UNIX System Manager's Manual*. Computer Systems Research Group, University of California at Berkeley, The Regents of the University of California and/or Bell Telephone Laboratories. 1986. SMM 14.

bits record the number of fragments less than a full block.

In the example above, the disk address for the second logical block would encode the number 7 in the upper nibble. Since there are eight fragments per block in this example, the number of fragments actually allocated in that slot is 8-7=1. The rest of the address contains the disk block number for the data.

This organization enables fragmented and non-fragmented filesystems to use an identical disk inode structure. Since these bits are always zero in a non-fragmented filesystem, they will be interpreted as a full block allocation in both fragmented and non-fragmented filesystems.

Another enhancement adopted from the Berkeley Fast Filesystem is the ability to specify the ratio of bytes to the number of inodes in the filesystem. This ratio is described as the Number of Bytes Per Inode (NBPI). In AIX Version 3, the JFS had a fixed NBPI of 4096. If a filesystem had 2,048 4 KB blocks, then it would have 2,048 inodes. More inodes could be added to the filesystem by increasing the size of the filesystem, but the ratio remained constant.

In AIX Version 4, the system administrator can specify different values of NBPI to customize the number of inodes in the filesystem. Raising the value of NBPI to its maximum of 16384 creates filesystems with fewer inodes for a given size. This change reduces the amount of space normally reserved for inodes and benefits filesystems that store a few large files. In contrast, lowering the value of NBPI results in filesystems with more inodes for a given size. This might be useful for a filesystem with many files smaller than 4 KB. When a new filesystem is created, both the fragment size and the NBPI value are set and cannot be changed. The default value of 4096 for both the fragment size and NBPI results in a filesystem that is both disk image-compatible with AIX Version 3 and interchangeable with an AIX Version 3 machine.

One disadvantage of having variable size allocations is free space fragmentation. The inode organization of JFS requires that a 4 KB block of fragments be allocated contiguously. Even though a filesystem may have plenty of free space, if contiguous fragments cannot be allocated, the system call that attempted to allocate more space will fail with the error "ENOSPC".

The defragmentation utility in AIX Version 4 solves this problem. By attempting to coalesce the free space in the disk allocation map, this utility facilitates future allocations of contiguous fragments. This process enables full block allocations, which can only be made from contiguous fragments. The tool called *defragfs* can be used on an active filesystem. Alternatively, you can use its spy mode to view the fragmentation of the filesystem and any changes that would have been made if you had used the tool in active mode.

File Compression

Compression is built on top of JFS fragments. With JFS, compressed filesystems previously available for DOS are now possible in AIX Version 4. In a compressed filesystem, the data for regular files is dynamically compressed to reduce the amount of space consumed by the file. Directories, though not compressed, are fragmented, just as they would be in a fragmented filesystem.

In AIX Version 4, the system administrator can specify different values of NBPI to customize the number of inodes in the filesystem.

Unlike a fragmented filesystem in which only the last block is partially allocated, a compressed filesystem enables each block to be allocated less than a full block of fragments. A block about to be written to disk is first compressed. If the data can be compressed by at least one fragment, then the exact number of fragments required to hold the compressed data—not a full block—is allocated. This process happens independently for each block in the file.

Although it would be more space efficient to compress the entire file instead of compressing each individual block separately, it is impractical because the JFS must support efficient random I/O. To be accessible at any arbitrary offset, a file would have to be completely decompressed in memory and remain in memory. Although this implementation is good at maximizing the compression rate, the performance and system resource utilization would be extremely inefficient.

JFS uses the algorithm based on Lempel-Ziv[2,3], which results in a high compression-efficiency rate. The efficiency of the algorithm is a function of whether data values are uniformly distributed over the range of possible values (truly random data). Since both text files and executables exhibit a fairly non-uniform distribution, they can compress well and can have a compression rate as high as 50%.

[2] Ziv, J. and Lempel, A. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on Information Theory* (May 1970) p. 337-343.

[3] Brent, R.P. "A Linear Algorithm for Data Compression," *The Australian Computer Journal* 19 (May 1987).

The JFS uses a conservative space-allocation policy to avoid over-allocating disk resources. In standard UNIX, disk space is allocated synchronously. When the `write` system call extends a file, the space is allocated immediately. If there is not enough free space available, the write fails with the error code ENOSPC. In a compressed filesystem, the minimum amount of space required to store a block is unknown until after the data is compressed. Performance reasons make it impractical to compress the data each time a write is performed.

A good solution is to decompress the data immediately after it is read from the disk and to compress it immediately before it is written back to disk. The file block is decompressed while it is in memory, enabling file reads and writes to occur without compressing or decompressing the data.

The JFS uses a conservative space-allocation policy to avoid over-allocating disk resources.

Delaying compression of data until it is written back to disk also implies delaying the allocation of space. This delayed allocation can lead to problems if the filesystem does not have sufficient free space. For example, if an existing file block is compressed, an application modifies the block via the `write` system call. Then JFS prepares to write the data back to disk by compressing the data. If the new data cannot be compressed as tightly as before, and if the filesystem has no free space, then JFS cannot save the data. In this example, since the write is asynchronous to the application program, JFS must simply discard the data.

To prevent this worst-case scenario, JFS uses a conservative approach that fully allocates the block while it is in memory, and then reallocates the block when it is written out. Even if the filesystem is full, or if the data cannot be compressed, the file will have allocated sufficient space so the operation can succeed. This approach guarantees the synchronous allocation requirements for the filesystem.

JFS compression supports different, user-defined compression algorithms. The compression code itself is completely encapsulated in an independent load module. This module is loaded as a kernel extension for internal use by JFS. The module is also explicitly loaded into application programs such as `mkfs` and `backup` for inode backups.

Large Filesystems

In classic UNIX, the starting logical offset for `read` and `write` system calls is stored as an absolute byte address in the system open file table. It is maintained by the system and can be modified with the `lseek` system call. In the typical 32-bit implementation, the file offset is defined as a signed 32-bit integer. This effectively limits the maximum file offset to 2 GB. Since devices are accessed as files, the same limitation applies to devices.

To allow filesystems larger than 2 GB, both the kernel and application programs must have the ability to address the storage beyond the 2 GB limit. For the kernel, this is not a problem since the kernel uses the block-oriented strategy interface of the device driver to make I/O requests. For the filesystem commands, such as `fsck`, only the byte-oriented `lseek` interface is available.

Several alternatives for extending the range of the file offset are possible. By changing the file offset to an unsigned integer, the effective range can be increased to 4 GB. Some implementations have added a special `ioctl` command, which causes the file offset to be interpreted as a block offset instead of a byte offset. Perhaps the most natural solution is to increase the size of the offset.

In AIX Version 4, the file table offset is re-implemented as a 64-bit integer. A new abstract data type, `offset_t`, derived from the "`long long`" base type, is defined in `<sys/types.h>`. A new system call, `llseek`, is created. Its interface is identical to `lseek`, except that the second parameter and the return value are the `offset_t` type.

In addition to the system open file table itself, the device driver interface must be enabled for device access beyond 2 GB. The `uio` structure is passed to the device driver's read and write entry points. This structure defines the I/O request, including the logical starting offset. The offset field in the structure is changed to be the `offset_t` type.

Although the read and write entry points of a device driver are of interest in understanding the implementation of the `read` and `write` system calls, they are not used by the filesystem when making I/O requests. Instead, the filesystem uses the strategy interface of the device driver. Only device drivers that can support "block" devices need to provide a strategy entry point.

The filesystem constructs a buffer header that describes the I/O to the device driver. Encoded in the `b_blkno` field is the logical 512-byte block number for the request. Since the strategy interface is not byte-oriented, no changes were necessary to allow filesystems larger than 2 GB. The signed 32-bit block number allows block offsets

up to 2^{40}. The maximum device size for AIX Version 4 is effectively 1 terabyte.

The maximum filesystem size for the JFS is now limited by the internal data structures and algorithms of the JFS, not the size of the file table offset. The factors that determine the maximum size include the width of the disk addresses in inode and indirect blocks, as well as the limit on the maximum number of inodes in a filesystem. The new theoretical maximum filesystem size is 256 GB.

The filesystem is not the only beneficiary of the larger offset. Application programs that directly access devices, such as raw logical volumes, can now access data beyond the 2 GB boundary. Applications need only be recoded to use `offset_t` and `llseek`.

Conclusions

These enhancements improve the competitive position of the AIX filesystem. Fragments brings the JFS into line with the capabilities of the Berkeley Fast Filesystem. Compression raises the bar for all UNIX filesystems. Large filesystems ease the system management aspects of multi-disk configurations and make the first step towards files greater than 2 GB.

Bill Baker, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: web@austin.ibm.com. Mr. Baker is a senior programmer in the AIX File System Development department of IBM's RISC System/6000 Division. He previously was a software engineer at Texas Instruments, where he led the team that developed a UNIX implementation for TI's 680x0-based multiprocessor system. He has a BS in Computer Science from the University of Oklahoma.

COMMON Prepares for Blowout Fall Conference in San Antonio

More than 4,000 midrange computer professionals are expected to attend the COMMON Fall 1994 Conference (October 16-20) and the COMMON Fall Expo (October 15-17) at the Marriott Rivercenter, the Convention Center in San Antonio, Texas.

COMMON, the world's largest IBM midrange user group, is composed of over 4,800 installations and users of the IBM AS/400®, System/36, RISC System/6000, and personal computers. Members also include users interested in connectivity or networking solutions within the Enterprise Systems Architecture product family.

Nearly 100 sessions and labs will be devoted to RISC and AIX solutions at COMMON '94. In addition, the Fall Expo will feature 90,000 square feet filled with the latest midrange products and services offered by IBM and more than 120 of the industry's leading vendors.

The conference keynote address will be delivered by IBM CEO and Chairman Lou Gerstner on Monday, October 17. Gerstner is expected to discuss IBM's plans to help customers face the future. Many IBM Austin executives and developers are also expected to attend.

Conference registration is limited to COMMON members; both company and individual memberships are available. Companies that register more than 10 attendees will receive a multiple employee discount. For more information about COMMON, including the 1994 and 1995 conference schedules, contact the following:

COMMON Headquarters
401 North Michigan Avenue ◆ Chicago, IL 60611-4267
phone: (800) 777-6734 ◆ (312) 644-6610
fax: (312) 644-0297