Linux File Systems
Part I: Journaled File Systems comparison (draft, 0.5)

Alexander Bokovoy `<a.bokovoy@sam-solutions.net>`

20th September 2001

Abstract

While the Extended file system v.2 (ext2fs) was originally released in January 1993, it is still the predominant file system in use by GNU/Linux and GNU HURD. There is a steady growth in the demand for storage architecture that provides quick and effective manipulation of huge amounts of data while the storage itself is kept in a consistent state. First part of this paper will study state of art of the journaled file systems for GNU/Linux operating system in terms of stability, efficiency, extensibility, and interoperability with applications which are mostly used in server appliances. Network-wide file systems will be studied in the second part of the document.

Contents

1 Requirements

There are several journaled file systems available for GNU/Linux operating system. All of them are still in development phase, though their stability and feature set increase quickly. In this section we will try to express common requirements for underlying file systems (FS) so that appliance–related applications will show increase in both feature set and performance when these requirements are satisfied.

For server appliance it is specific that file systems should be separated into two distinct classes: local and network-wide ones. Both sides of appliance's function have different requirements though there are some similarities in them:

1. Both local and network-wide FS should support discretional access model for all their nodes (directories, files, soft and hard links). This is important for consistent access rights checking and for compatibility because this is most used access model in Unix for past 30 years. However, it is unnecessary to support Access Control Lists (ACLs) or other advanced access models at file system level because they are better modeled at kernel level using general security suites like SELinux[1] or RSBAC[2].
2. Both types of FS should support quotas in order to make fine-grained control of disk space usage.
3. File systems should be transparent as much as possible for all user-visible properties like file or directories' names. This includes in first place transparency for node names, allowing existence of international characters[3].
4. Local file systems should not create additional problems for high-level technologies like logic volume management or file systems snapshoting.

Additional requirements for local journaled file systems:

1. Journal relocation. File system should be able to operate when journal logging is done on the different device than data partition. This requirement can be weaken if journal relocation support is planned in short to middle term perspective in the file system's development roadmap.

Network-wide file systems also should be able to operate in so-called "disconnected" mode. However, since network-wide FS usually represent some kind of layer above local FS which allows remote access to data stored in it, only local FS will be reviewed below. This is important closure because there is no much use of journaled FS that allow networking journal placement yet and it reduces amount of data under investigation. Network-wide file systems for GNU/Linux operating system will be analyzed in the second part of this paper.

Journaled file systems allow fast deserialization of meta-information which has been placed during write operations into so-called 'journal' in order to allow playback of meta-information in case of failure. This allows fully ACID[4] operations and ensures data integrity. However, there are several situations when transaction rollback leads to data loss, such situations will be noted in conjunction with particular FS further in this document.

2 Overview

Besides requirements briefly mentioned in the previous section, all journaled FS for GNU/Linux address yet another issues: maximum size of the file system, maximum file size, and block sizes. Their precursors often had fixed storage size for specific fields which created limitations only when hardware was developed enough to overcome these fixed limits. Following table shows limits for 4 journaled file systems available now under GNU/Linux.

[1]See SE Linux home page for more

[2]See RSBAC project home page for more

[3] For network-wide FS this is an important issue because of client-server architecture which allows different settings for client and server. The only way to make this transparency working is to use one of the Unicode encodings (UTF-8, UCS2, etc) as transfer encoding.

[4]ACID is an acronym which stands for Atomicity, Consistency, Isolation, and Durability

	Maximum FS size	Block sizes	Maximum file size
XFS	9000 Pb	512b up to 64Kb	9000 Pb
JFS	4 Pb	512b up to 4096b	512 Tb
JFS	32 Pb	4Kb	4 Pb
ReiserFS	17592 Gb	4Kb up to 64Kb	up to 1024 Pb, cur. 17592 Gb (v.3.6/32-bit)
Ext3FS	4 Tb	1Kb up to 4Kb	2GB

Storage limits' increase also adds overhead in code that looks up in the node set. To reduce this overhead, all reviewed systems use B^+ trees for storing meta-information about FS structure. B^+ tree algorithms perform better on large trees which is in general exactly the same situation we are observing in server appliances with huge storages. However, different file systems use B^+ tree in different places, maintain either single B^+ tree for the whole file system or set of B^+ trees for each single directory.

There are minor variations on the binary trees theme along journaled file systems. Some of them like JFS store both extended information and bitmaps and use n-ary trees for some components. Others, like ReiserFS, use B^* trees with a single tree for the whole file system where each subdirectory has its own subtree inside this big B^* tree. ReiserFS also lacks extended information though support for it is planned.

Extended information is used to enhance properties of the original FS and add new features. However, there is no standardized approach to store this kind of information which leads to possible meta-information loss during data transition between two different file systems. It is important problem for server appliances because this greatly complicates upgrades with possible replacement of underlying file system. But since most of extended information is used to store additional node attributes like ACLs, and there are better ways to store this information already exist, it is possible not to use extended information at all, bypassing these issues without any troubles. It also allows to use journaled file systems which don't provide extended attributes but more stable than their "extended" analogs.

There are also symbiotic FS which function on both local and network levels. They are will be reviewed in the second part of this paper and here we only name few of them:

– Global File System (GFS) which supports serverless operation, network-oriented journalling and load balancing for the cluster of storages connected by Fibre Channel or shared SCSI devices. GFS also has full support for Logic Volume Manager enabling automatic online resizing of storage space. For more information see GFS project home page.

– Cluster File System a.k.a. "Lustre project" uses an object-based storage protocol and allows various cluster file system functions to be partitioned across multiple systems in different ways. Lustre extension called OSD provides an extended object-based SCSI command set which could be used for low-cost SAN replacement. In comparison with iSCSI solution, this is truly object environment but in reality it is far from working product yet.

– InterMezzo is a new distributed file system that supports disconnected operations and automatic recovery from network outages. It was inspired by Coda File System, and main target of InterMezzo is to provide support for flexible replication of directories[5].

3 Journaled file systems for GNU/Linux

In this section we'll review following journaled file systems: XFS, ReiserFS, JFS, and Ext3fs. For each file system its details will be shown, discussing their application to ApplianceWare product line and possible problems with higher level applications. Each file system will be tested using common testing framework [TBD].

3.1 Why file system X isn't in the list?

Obviously, there are more journalling file systems available for GNU/Linux. Unfortunately, in most cases it is so-called "press availability" when development project is started but its results are not usable in production yet. This is exciting how many attempts exist to provide journalling support for GNU/Linux

[5]It is interesting that Cluster FS, InterMezzo, and Coda FS all are under influence of one person — Peter J. Braam from Carnegie Mellon University and Mountain Data Vista. All of them share similar ideas but target slightly different markets.

file systems but our goal is to find those which could be used in production in short term. All four file systems we are reviewing in this paper are close to our requirements for stability and supported feature set. If any other experimental journalling file system for GNU/Linux will reach same degree of attitude as these four FS, we'll test it and add results into this document. For now, XFS, ReiserFS, JFS, and Ext3fs are much more suited for production than any other journalling file systems for GNU/Linux.

3.2 SGI's XFS

XFS was originally developed by Silicon Graphics Inc. (now SGI) for it's own IRIX operating system in early 1990's. It was designed to scale to meet the most demanding storage capacity and I/O needs now and for many years to come (at least, it was supposed to be so in '90s). XFS achieves this through extensive use of B^+ trees in place of traditional linear FS structures and by ensuring that all data structures are appropriately sized. Original IRIX version of XFS includes a fully integrated volume manager, XLV, but Linux port does not support it and plan to make XLV operational on Linux is long-term based.

3.2.1 File system details

Currently, XFS supports 2Tb FS size on 32-bit architectures with Linux 2.4. When Linux kernel will support 64-bit operations on block devices' layer, maximum FS size will be 9000 Pb. The maximum accessible file offset is 16Tb using 4Kb page size and 64Tb using 16Kb page size. However, working 64-bit block devices' layer will increase these numbers up to 9000 Pb. This means that maximum file size of XFS is limited only by disk space and storage partitioning. XFS can easily store file with size up to maximum FS size.

XFS uses extents technique for disk space allocation. It means that large numbers of contiguous blocks (extents) are allocated to the file and tracked as a unit. XFS support per-file configurable extents size, however, real extent size is a multiple of file system block size which is currently fixed at system page size (4Kb on IA-32). It is important to understand that this is both advantage and disadvantage: while large files will benefit from such design, small ones will lose. It is especially important for files with average sizes less than file system block size (e.g. 4Kb) because in this case disk space is allocated suboptimal and reading of n small files will require no less than n I/O requests (ReiserFS, on contrary, owns ability to pack small files into file system blocks, reducing number of I/O operations to get them).

XFS also provides solution for this problem by allowing to store small enough file directly in the inode to conserve space. By default, inode size is 256 bytes but it can be enlarged up to 64Kb during file system creation. Thus, this isn't flexible but acceptable for applications where nature of stored files is known in advance though it isn't our case.

Hence, XFS by design is well suited for storing large files. There are several situations when such large files actually are sparse thus additional optimisations for occupied storage space are desirable. While most conventional file systems don't have such optimisations (though outdated UFS had albeit inefficient in access costs sparse file support), XFS does. It supports 64-bit sparse address space for each file. The methods that XFS uses to allocate and manage extents make this efficient since XFS stores the block offset within the file as part of the extent descriptor. This leads to possibly discontinuous file extents.

From our point of view, it may cause a problem for FS snapshotting since it requires intensive work with XFS internals, even duplicating part of XFS internal logic in the snapshot application.

XFS supports both large directories and large number of files, either in one directory or in the whole file system. Internally, XFS uses an on-disk B^+ tree structure for its directories. But instead of using directory names as keys, it first converts them into 32-bit hash values and then uses these hashed integers as keys in the B^+ tree describing directory. It makes efficient lookup, create, and remove operations in directories with millions of entries. However, listing operation is inefficient due both the size of the resulting output and B^+ tree organization which isn't well suited for such tasks.

In XFS, the number of files in a file system is limited only by the amount of space available to store them. XFS dynamically allocates inodes as needed, eliminating a problem with inode limitations being found, for example, in Ext2fs.

3.2.2 Journalling details

Logging disk writes to journal is possible in two ways: synchronous and asynchronous. During synchronous logging file system driver tries to write journal log before declaring a transaction committed and unlocking resources. During asynchronous logging file system driver ensures that the write ahead logging protocol is followed in that modified data cannot be flushed to disk until after data is committed to the on-disk log. The modified resources themselves are unlocked in memory until the transaction commit is written to the on-disk log. Thus, correct order (log before, write data after) is preserved and even during power outages file meta information could be restored easily.

XFS writes meta information only to the log. Actually, this meta information contains binary representation of inodes, directory blocks, free extent tree blocks, inode allocation tree blocks, file extent map blocks, allocation group header blocks, and the super block. Recovering the data structures from the log requires nothing but replaying the block and inode images in the log out to their real locations in the file system. The log recovery does not know that it is recovering a B^+ tree. It only knows that it is restoring the latest images of some file system blocks.

It makes XFS journalling technique very flexible to the actual file system structure and also allows to change its components as far as committing module understands these changes.

Most time XFS is journalling meta information in asynchronous way but it could be reverted back to synchronous one by utilising '*wsync*' mount option. This is most useful when XFS partition is exported via NFS which requires that all transactions be synchronous.

Synchronous mode for NFS–exported partitions will obviously cause performance drop but percentage of it isn't measured in any known tests of journaled file systems for GNU/Linux.

Exporting XFS partitions to NFS also requires logging on a separate device in order to make this process more reliable. Fortunately, XFS does support log relocation in the Linux port where log can be put on any other device like dedicated disk or non-volatile memory device. It is controlled by '*logdev*' mount option.

3.2.3 Performance details

This section will contain results of test benchmark when it will be done. [TBD].

3.2.4 Conclusion

XFS looks promising. It satisfies all requirements described above and even goes further implementing several additional features like seamless NFS integration and non-buffered data access for streamlined operations (live audio or video and so on). [Add here performance conclusion].

3.3 IBM's JFS

IBM's Journaled File System (JFS) represents one of the numerous IBM's contributions for Open Source community. JFS was originally developed for IBM AIX operating system and is currently used in IBM enterprise servers. Though AIX version of JFS has proved its stability, Linux port is still in the development stage. Don't be fooled by 1.0 version number, Linux JFS is beyond capabilities and stability of its AIX version. Moreover, Linux JFS source code has little in common with AIX version. This "port" is actually rewrite of OS/2 port instead. The latter was also not so stable as original AIX version.

Other fact worth to note here is that both XFS and JFS were designed with different hardware requirements in mind than most popular GNU/Linux platform (IA32). It means that some algorithms might be less optimized than in IA32–grown file systems.

3.3.1 File system details

JFS uses slightly different terminology for its components. It distinguishes meta information about disk partition itself and file-specific meta information. Both are contained in the aggregate which is JFS' abstracted term for partition. Aggregate is a container for disk blocks and allocation map as well as file set and control structures necessary to describe it. This terminology was born with the need to support DCE DFS (Distributed Computing Environment Distributed File System). In JFS notation file set is a mountable entitity. This differs from usual meaning of mountable partition because aggregate might contain several file sets and each of them might be mounted separately while all of them physically reside on the same disk partition. However, JFS 1.0 for GNU/Linux supports only one file set per aggregate.

JFS is a full 64-bit file system. The minimum file system size supported by JFS is 16 Mb, maximum depends on the file system block size and the maximum number of blocks supported by the file system meta–data structures. For 512B blocks maximum file system size is 512 Tb, for 4 Kb blocks — 4 Pb.

File size is limited only by virtual file system framework which currently allows 64–bit file offsets. It means that maximum file size on JFS file set can be up to 4 Pb.

JFS manages allocation groups (AGs) in each aggregate. Allocation groups divide the space in an aggregate into chunks, and allow JFS resource allocation policies to use well known methods for achieving great JFS I/O performance. First, the allocation policies try to cluster disk blocks and disk inodes for related data to achieve good locality for the disk. Files are often read and written sequentially, and the files within a directory are often accessed together. Second, the allocation policies try to distribute unrelated data throughout the aggregate in order to accommodate locality. Allocation groups within an aggregate are identified by a zero-based AG index, the AG number.

A "file" is allocated in sequences of extents. An extent is a contiguous variable-length sequence of aggregate blocks allocated as a unit. An extent can range in size from 1 to $2^{24} - 1$ aggregate blocks. An extent may span multiple Allocation Groups. These extents are indexed in a B^+ tree for better performance in inserting new extents, locating particular extents, etc.

Two values are needed to define an extent, its length and its address. The length is measured in units of aggregate block size. JFS uses a 24–bit value to represent the length of an extent, so an extent can range in size from 1 to $2^{24} - 1$ aggregate blocks.

With a 512–byte aggregate block size (the smallest allowable), the maximum extent is $512 * (2^{24} - 1)$ bytes long (slightly under 8G). With a 4096–byte aggregate block size (the largest allowable), the maximum extent is $4096 * (2^{24} - 1)$ bytes long (slightly under 64G). These limits only apply to a single extent; they have no limiting effects on overall file size. The address is the address of the first block of the extent. The address is also in units of the aggregate blocks: it is the block offset from the beginning of the aggregate.

In general, the allocation policy for JFS tries to maximize contiguous allocation by allocating a minimum number of extents, with each extent as large and contiguous as possible. This allows for large I/O transfer, resulting in improved performance. However, in special cases this is not always possible. For example, copy–on–write clones of a segment will cause a contiguous extent to be partitioned into a sequence of smaller contiguous extents. Another case is restriction of extent size. For example, the extent size is restricted for compressed files since JFS must read the entire extent into memory and decompress it. JFS has a limited amount of memory available, so it must ensure that it will have enough room for the decompressed extent.

A defragmentation utility is provided to reduce external fragmentation, which occurs from dynamic allocation/deallocation of variable–size extents. This allocation and deallocation can result in disconnected variable size free extents all over the aggregate. The defragmentation utility will coalesce multiple small free extents into single larger extents.

JFS on–disk inode is 512 bytes. A JFS on–disk inode contains four basic sets of information. The first set describes the POSIX attributes of the JFS object. The second set describes additional attributes for JFS object; these attributes include information necessary for the VFS support, information specific to the OS environment, and the header for the B^+ tree. The third set contains either the extent allocation descriptors of the root of the B^+ tree or in–line data. The fourth set contains extended attributes, more in–line data, or additional extent allocation descriptors.

JFS allocates inodes dynamically by allocating inode extents which are simply contiguous chunk of inodes on the disk. It is important that inode numbers are simple counters and there is no direct relation between them and the disk address of inode.

FS 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.

A normal file is represented by an inode containing the root of a B^+ tree which describes the extents containing user data. The B^+ tree is indexed by the offset of the extents.

JFS uses original technique for storing directories. First, directory is a journaled media–data file in JFS. A directory is composed of directory entries which indicate the objects contained in the directory. A directory does not contain specific entries for self(".") and parent (".."). Instead these are represented in the inode itself. Self is the directory's own inode number.

The directory inode contains the root of its B^+ tree in a similar manner to a normal file. However this B^+ tree is keyed by name. The leaf nodes of a directory B+ tree contain the directory entry and are keyed from the complete name of the entry. The directory B+ tree uses suffix compression for the last internal nodes of the B+ tree. The rest of the internal nodes will use the same compressed suffix. Suffix compression truncates the name to just enough characters to distinguish the current entry from the previous entry.

GNU/Linux port of JFS does not currenly provide facility for Access Control Lists and Extended Attributes though these are defined in the file system layout white paper. It also lacks quota support.

3.3.2 Journalling details

As any other novel journaled file systems for GNU/Linux, 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 rely upon (multiple) careful synchronous meta–data writes for consistency. However, JFS logging style by design slower than XFS one though they are comparable when XFS is used with synchronous writes. 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.

JFS logging system consists of two main components: the log file itself and the transaction manager. Latter provides the core functionality that JFS uses to do logging. JFS logs following file system operations that change meta–data on disk:

- File creation
- Linking
- Making directory
- Making node (device)
- Removing file
- Removing directory
- Symbolic link
- Set ACL[6]
- Writing file
- Truncating regular file

JFS log file is about 40% of the aggregate size and is rounded up to a megabyte boundary. The maximum size that the log file can be is 32 Mb. The log file size is then converted into aggregate blocks.

Currently, JFS cannot relocate log file and puts it inside aggregate during file system creation phase (mkfs.jfs call).

3.3.3 Performance details

This section will contain results of test benchmark when it will be done. [TBD].

3.3.4 Conclusion

Unfortunately, current release of Linux JFS doesn't satisfy requirements described in the 'Requirements' section. JFS lacks quota support and journal relocation which are planned for long term without any precise schedule.

3.4 ReiserFS

ReiserFS is probably the oldest Linux–oriented journaling file system. It was originated in early 1990's when Hans Reiser went to Russia in 1992 and worked with russian programmers on the file system called 'treefs' which was later renamed to ReiserFS. ReiserFS introduces a new approach in the file system development using technologies well-known in the database world. After that all contemporary journaled file systems use more or less same algorithms. But ReiserFS goes further and tries to change approach in which user data is partitioned between files in storage. As Hans Reiser wrote in Linux-kernel mailing list: "The goal of reiserfs is to one by one eliminate the reasons why these [application–specific] new namespaces keep getting invented rather than using the filesystem namespace". ReiserFS is also the first journaled file system that made its way to official Linux kernel.

3.4.1 File system details

ReiserFS introduces balanced trees to the file system internal structure. This technique, almost obvious nowadays, was revolutionary in early 1990's. Though it was successfully used in database development, its application to file systems remained unexplored. Now ReiserFS uses B^* trees instead of B^+ trees which are in use by other FS.

These trees are used for layout optimization on four levels:

1. The mapping of logical block numbers to physical locations on disk;
2. The assigning of nodes to logical block numbers;

[6]Does not work in GNU/Linux port version 1.0.1 yet

3. The ordering of objects within the tree;
4. The balancing of the objects across nodes they are packed into.

The last level shows yet another feature of ReiserFS: it doesn't make file boundaries always block aligned. Instead, it tries to pack as much data into block as possible. This leads to better read performance when many small files are being requested because it requires less I/O operations.

ReiserFS uses three different kinds of nodes in its trees: internal nodes, formatted nodes, and unformatted nodes. The contents of internal and formatted nodes are sorted in the order of their keys, while unformatted nodes contain no keys. Internal nodes consist of pointers to sub-trees and exist solely to allow which formatted node contains the item corresponding to a key.

Formatted nodes consist of items. Items have four types: direct items, indirect items, directory items, and stat data items. All items contain a key which is unique to the item. This key is used to sort, and find, the item.

Direct items contain the tails of files, and tails are the last part of the file (the last file size modulo FS block size of a file).

Indirect items consist of pointers to unformatted nodes. All but the tail of the file is contained in its unformatted nodes.

Directory items contain the key of the first directory entry in the item followed by a number of directory entries.

Depending on the configuration of reiserfs, stat data may be stored as a separate item, or it may be embedded in a directory entry. We are still benchmarking to determine which way is best.

A file consists of a set of indirect items followed by a set of up to two direct items, with the existence of two direct items representing the case when a tail is split across two nodes. If a tail is larger than the maximum size of a file that can fit into a formatted node but is smaller than the unformatted node size (4k), then it is stored in an unformatted node, and a pointer to it plus a count of the space used is stored in an indirect item.

Directories consist of a set of directory items. Directory items consist of a set of directory entries. Directory entries contain the filename and the key of the file which is named. There is never more than one item of the same item type from the same object stored in a single node (there is no reason one would want to use two separate items rather than combining).

With all this information one should be able to proper balance it in the tree in order to make access to it in an efficient way. ReiserFS uses folowing ordering priorities

1. minimize number of nodes used
2. minimize number of nodes affected by the balancing operation
3. minimize the number of uncached nodes affected by the balancing operation
4. if shifting to another formatted node is necessary, maximize the bytes shifted

Most important feature of internal structuring of ReiserFS is that it tries to eliminate need to cache in RAM a map of object id to location for every object because it tends to be greed for RAM consuming in case of large number of small files. Internal nodes are meant to be "a compression" of object location information made effective by the existence of an ordering function, and this compression is both essential for small files.

With all this details theoretically ReiserFS is more suited for storing and accessing large number of small objects. That's why it is also positioned as a common replacement for an application specific namespace clustering when huge ammounts of small data blocks are organized into logical domains at higher level than FS and emulated using regular file/directory notion with all overhead neccessary to re–implement effective access techniques in every particulary application. The issue of namespace clustering is rightfully addressed by ReiserFS.

3.4.2 Journalling details

ReiserFS uses for journalling a system of write ordering that tracks all shifting of items in the tree, and ensures that no node that has has an item shifted from it was written before the node that has received the item was written. This approach has been slightly modified during development but in general it tries to keep locality of data as well as meta–data. Following paragraph from the ReiserFS site describes this algorithm:

> If an item is shifted from a node, change the block that its buffer will be written to. Change it to the nearest free block to the old blocks left neighbor, and rather than freeing it, place the old block number on a "preserve list". (Saying nearest is slightly simplistic, in that the blocknr assignment function moves from the left neighbor in the direction of increasing block numbers.) When a "moment of consistency" is achieved, free all of the blocks on the preserve list. A moment of consistency occurs when there are no nodes in memory into which objects have been shifted (this could be made more precise but then it would be more complex). If disk space runs out, force a moment of consistency to occur. This is sufficient to ensure that the file system is recoverable. Note that during the large file benchmarks the preserve list was freed several times in the middle of the benchmark. The percentage of buffers preserved is small in practice except during deletes, and one can arrange for moments of consistency to occur as frequently as one wants to.

If the journal block count is smaller than n translations, application may lose speed but there is no general function for determining this number, ReiserFS team suggests that journal block should be no less than 8–16 transactions. Typical transaction size depends on the application, how often fsync is called, and how many meta–data blocks are dirty in a 30 second period. The more small files ($< 16Kb$) are in use, the larger transactions will be.

If the journal fills faster than dirty buffers get flushed to disk, it must flush them before allowing the journal to wrap, which slows things down. If there is a need for high speed meta data updates, the journal be big enough to prevent wrapping before dirty meta blocks get to disk.

If the batch max is smaller than the transaction max, FS will waste space at the end of the journal because journalling code sets the next transaction to start at 0 if the next transaction has a any chance of wrapping.

The large the batch max age, the better the speed, and the more meta data changes will be lost after a crash.

Currently ReiserFS has no means to relocate journal but this feature is still in the development and wouldn't find its way into official kernel before 2.5.1.

3.4.3 Performance details

This section will contain results of test benchmark when it will be done. [TBD].

3.4.4 Conclusion

ReiserFS seems to be most stable journaled file system for GNU/Linux. It some features we are looking for like journal relocation which is under development right now. [Add perfomance test results here].

3.5 Ext3 file system

Ext3 file system is different from all FS reviewed above in that instead of designing completely new journaled file system from scratch, its author decided to extend existing Ext2 file system by journal logging.

3.5.1 File system details

The layout of the journaled ext3 file system is fully compatible with existing ext2fs. Traditional UNIX file systems (to which ext2fs belongs) store data on disk by associating each file with unique numbered inode on the disk. Ext2fs provides a number of reserved inodes which Ext3 uses to store the file system journal.

Thus, Ext3 inherits all the properties Ext2 has.

Ext3 is still a work–in–progress. The design of the initial implementation is both stable and simple. However, the development is slow and only version 0.0.8 has been released recently[7].

3.5.2 Journalling details

Ext3 journalling subsystem also logs file system meta–data blocks but puts them into special journal file instead of internal FS structure. It was needed in order to keep compatibility with Ext2 on–disk structure. Journalling subsystem has three different types of data blocks in the journal: meta–data, descriptor, and header blocks.

A journal meta–data block contains the entire contents of a single block of file system meta–data as updated by a transaction. This implies writting of an entire journal block out to log on change but is acceptable for two reasons:

- Journal writes are quite fast since most writes to the journal are sequential, and the journal I/O operations could be put in a batch of large clusters which can be handled efficiently by the disk driver;
- By writting out the entire contents of the changed meta–data buffer from the file system cache to the journal less CPU work is used in the journalling code.

Descriptor blocks are journal blocks which describe other journal meta–data blocks. Whenever meta–data blocks are in the journal queue, descriptor blocks are used to record which disk blocks the meta–data normally lives at, so that the recovery mechanism can copy the meta–data back nito the main file system. A descriptor block is written out before each set of meta–data blocks in the journal, and contains the number of meta–data blocks to be written plus their disk block numbers.

Both descriptor and meta–data blocks are written sequentially to the journal, starting again from the start of the journal whenever end of it is reached.

Finally, the journal file contains a number of header blocks at fixed locations. These record the current head and tail of the journal as well as a sequence number. At recovery time, the header blocks are scanned to find the block with the highest sequence number and to run through all journal blocks from the tail to the head, as recorded in that header block.

There is no complete suspend of file system updates during transaction commit. Instead, Ext3 creates a new compound transaction in which to record updates which arrive while the old transaction is being committed. This may lead to cases when the new transaction wants to write to the buffer which creates circular dependency between transactions.

The solutions is to make a new copy of the buffer in such cases. Fortunately, reclamation of the old transaction's log space is handled automatically due to the fact that the buffer must necessarily be committed into the next transaction's journal records.

Ext3 also attempts to provide file system–independent journalling level into Linux kernel. This code might be per-used by other file systems be them local or network–wide. But currently there are no other users of this functionality.

3.5.3 Performance details

3.5.4 Conclusion

Ext3 maintains compatibility with Ext2 and adds journalling into it. Most exciting feature of Ext3 is the ability to easily switch ext2/ext3 drivers for the same partition given that partition itself was properly unmounted or file system checking utility was used before mount. There are special programs that allow to add basic journal info into existing ext2 partition to be able to use it with ext3 driver. [Add performance test results here]

4 Conclusion

[Add final conclusion here once performance tests will be done]

[7]It is first release of Ext3 for Linux 2.4 kernel series