Journal File Systems in Linux ^(59113 lectures)
Per **Ricardo Galli Granada**, _**gallir**_ (http://mnm.uib.es/~gallir/)
Creat el 24/01/2002 23:24 modificat el 24/01/2002 23:24

This is a lightly modified version of a couple of articles published in Novática[1] _(Spanish) and Upgrade_[2] _(English, PDF_[3]_) where we explain the implementation of all journaling file systems available for Linux._
PS: the article was written while Ext3 was still under development to include it in the standard kernel.

<div align="center">

Journal File Systems in Linux

Ricardo Galli
gallir@uib.es
Dept. de Matemàtiques i Informàtica
Universitat de les Illes Balears
Versión en castellano[4]
Version en Français[5]
</div>

First of all, there is no a clear winner, XFS is better in some aspects or cases, ReiserFS in others, and both are better than Ext2 in the sense that they are comparable in performance (again, sometimes faster, sometimes slightly slower) but they a journaling file systems, and you already know what are their advantages... And perhaps the most important moral, is that Linux buffer/cache is really impressive and affected, positively, all the figures of my compilations, copies and random reads and writes. So, I would say, buy memory and go journaled ASAP...

Keywords: Linux, operating systems, page cache, buffer cache, journal file systems, ext3, xfs, reiserfs, jfs.

Introduction

The paragraph above was the first one in an article[6] published in the Balearic Islands Linux User Group web which described a second run of benchmarks over the Linux journaled file systems in comparison against traditional Unix file systems.

Although somehow informal, both benchmarks covered FAT32, Ext2, ReiserFS, XFS, and JFS for several cases: Hans Reiser's Mongo benchmark tool, file copying, kernel compilation and a small C program that to simulated the access patterns of database systems.

Both articles were among the first published, and our server was overwhelmed due the _slashdot effect_ derived from their publication in _slashdot.org_. They mainly served to de−mystify the common believe that journaling file systems are significantly slower in comparison with traditional Unix file systems (UFS) and derived, namely ext2, which has been the standard file system in Linux.

Since those days, more benchmarks have been published, but the truth is still the same: there is not a clear winner. Some systems perform better than other in some cases, for example ReiserFS is really good for reading small to medium size files, while XFS behaves better for large files, and JFS is said to facilitate the migration of existing system running in OS/2 Warp and AIX systems.

This article presents the all journal file systems available for Linux: Ext3, ReiserFS, XFS and JFS. We also introduce

the basic concepts of file systems, *buffer−cache*, and *page−cache* implemented in the Linux kernel. The performance of the different file systems is strongly affected by those optmisation techniques. Indeed, not only the performance is affected, but also the implementation and porting of the different file systems. SGI introduced a new module, *pagebuf*, that serves as the interface between their own XFS buffering techniques and the Linux page cache.

The Linux Virtual File System

A *File* is a very important abstraction in the computing programming field. Files serve for storing data permanently, they offer a few simple but powerful primitives to the programmers. Files are normally organised in a tree−like hierarchy where intermediate nodes are directories, which in turns are capable of grouping files and sub−directories.

The file system is the way the operating system organises, manages and maintains the file hierarchy into mass−storage devices, normally hard disks. Every modern operating system supports several different, disparate, file systems. In order to maintain the operating system modular, and to provide applications with a uniform programming interface (API), a higher layer that implements the common functionality of those underlying file systems is implemented in the kernel: the *Virtual File System*.

File systems supported by the Linux VFS fall into three categories:

1. Disk based, including hard disk, floppy disk and CD−ROM, including ext2fs, ReiserFS, XFS, ext3fs, UFS, iso9660, etc.
2. Network based, including NFS, Coda, and SMB.
3. Special file systems, including */proc*, ramfs, and devfs.

The common file model can be viewed as object−oriented, with objects being software constructs (data structures and associated methods/functions) of the following types:

- **Super block**: stores information related to a mounted file system. It is represented by a file system control block stored on disk (for disk based file systems).
- **i−node**: stores information relating to a single file. It corresponds to a file system control block stored on disk. Each i−node holds the meta−information of the file: owner, group, creation time, access−time and a set of pointer to the disk block that store the file date.
- **File**: stores information relating to the interaction of an open file and a process. This object only exists while a process is interacting with a file.
- **Dentry**: links a directory entry (pathname) with its corresponding file. Recently used dentry objects are held in a dentry cache to speed up the translation from a pathname to the inode of the corresponding file.

All modern Unix systems allow file system data to be accessed using two mechanisms (Figure two).

1. Memory mapping with **mmap**: The mmap() system call gives the application direct memory−mapped access to the kernel's page cache data. The purpose of mmap is to map a file data into a VMS address space, so data in the file can be treated as a standard in−memory array or structure. File data is read into the page cache lazily as processes attempt to access the mappings created with mmap() and generate page faults.
2. Direct block I/O system call such as **read** and **write**: The read() system call reads data from block devices into the kernel cache (avoided for CD and DVD reading by means of O_DIRECT *ioctl* parameter), then it copies data from the kernel's cached copy onto the application address space. The write() system call copies data in the opposite direction, from the application address space into the kernel cache and eventually, in a *near future*, writing the data from the cache to disk. These interfaces are implemented using either the *buffer cache* or the *page cache* to store the data in the kernel.

Linux Page−cache and Buffer−cache

In older Linux versions (and general UNIX−like operating systems), memory−mapping requests were handled by the virtual memory management subsystem (VM or MM), while I/O calls were handled independently by the I/O subsystem. For example, in Linux up to version 2.2.x, the VM subsystem and I/O subsystem each have their own data caching mechanisms to improve performance**:** *buffer cach*e and *page cache*.

Figure: Buffer cache and page cache
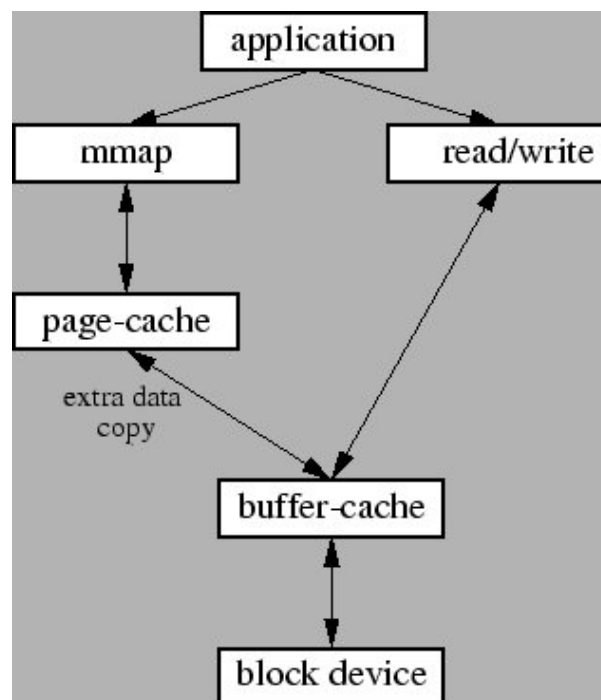
The Buffer Cache

The buffer cache holds individual disk blocks copies. The device and block numbers indexes the cache entries. Each buffer refers to a single arbitrary block on the hard disk, and it consists of a header and an area of memory *equal* to the block size of the associated device.

To minimise management overhead, all buffers are held in one of several linked lists. Each linked list contains buffers in the same state: unused, free, clean, dirty, locked, etc.

Every time a read occurs, the buffer cache sub−system must find if the target block is already in cache. To find it quickly, a hash table is maintained of all the buffers present in the cache. The buffer cache is also used to improve writing performance. Instead of carrying out all writes immediately, the kernel stores data temporally in the buffer cache, waiting to see if it is possible to group several writes together. A buffer that contains data that is waiting to be written to disk is termed *dirty*.

The Page Cache

The page cache, instead, holds full virtual memory pages (4 KB in x86 Linux platform). The pages come from files in the file system, and, in fact, page cache entries are partially indexed by the file i−node number and its offset within the file. A page is almost invariably *larger* than a single disk logical block, and the blocks that make up a single page cache entry may not be contiguous on the disk.

The page cache is largely used to interface the requirements of the virtual memory subsystem, which uses fixed 4 KB size pages, to the VFS subsystem, that uses variable size blocks or other types of techniques, such as extents in XFS and JFS.

Integration of page and buffer cache

The above two mechanisms operated semi−independently of each other. The operating system has to take special care to synchronise the two caches in order to prevent applications from receiving invalid data. Furthermore, if the system become short on memory, it has to take hard decisions on whether reclaim memory from the page cache or buffer cache.

The page cache tends to be easier to deal with, since it more directly represents the concepts used in higher levels of the kernel code. The buffer cache also has the limitation that cached data must always be mapped into kernel virtual space, which puts an additional artificial limit on the amount of data that can be cached since modern hardware can easily have more RAM than kernel virtual address space.

Thus, over time, **parts of the kernel have shifted over from using the buffer cache to using the page cache**. The individual blocks of a page cache entry are still managed through the buffer cache. But accessing the buffer cache directly can create confusion between the two levels of caching.

This lack of integration led to inefficient overall system performance and a lack of flexibility. To achieve good performance it is important for the virtual memory and I/O subsystems to be highly integrated.

The approach taken by Linux to reduce the inefficiencies of double copies is **to store the file data only in the page cache** (Figure shared).

Figure: Data is shared by page cache and buffer cache
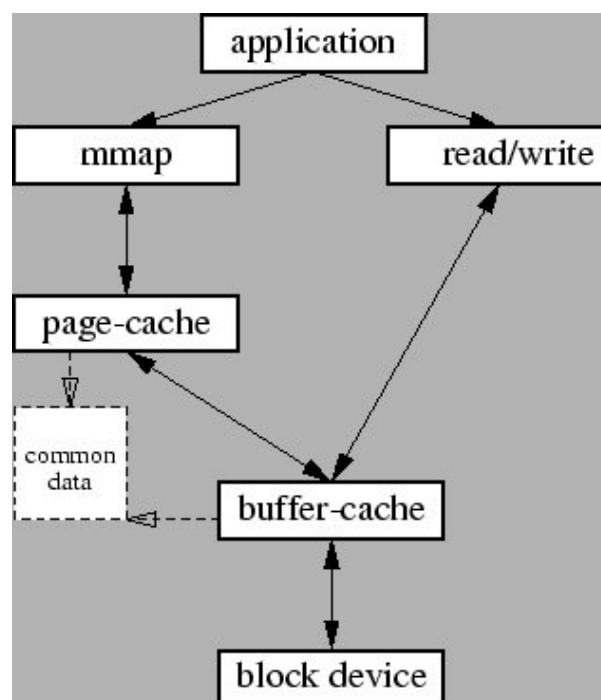
Temporary mappings of page cache pages to support read() and write() are hardly needed since Linux maps permanently all of physical memory into the kernel virtual address space. One interesting twist that Linux adds is that the device block numbers where a page is stored on disk are cached with the page in the form of a list of *buffer_head* structures. When a modified page is to be written back to disk, the I/O requests can be sent to the device driver right away, without needing to read any indirect blocks to determine where the data must be written.

Page−cache Unification

Following the ``Unified I/O and Memory Caching Subsystem for NetBSD'', Linus Torvalds wanted to change the page−buffer cache behaviour for Linux. In May 4th, 2001, in a message to the Linux−kernel developer's list, he wrote the following:

> *I do want to re−write block_read/write to use the page cache, but not because it would impact anything in this discussion. I want to do it early in 2.5.x, because:*
>
> *− it will speed up accesses*

> *– it will re−use existing code better and conceptualize things more cleanly (ie it would turn a disk into a _really_ simple filesystem with just one big file ;).*
>
> *– it will make MM handling much better for things like fsck − the memory pressure is designed to work on page cache things.*
>
> *– it will be one less thing that uses the buffer cache as a ``cache'' (I want people to think of, and use, the buffer cache as an _IO_ entity, not a cache).*
>
> *It will not make the ``cache at bootup'' thing change at all (because even in the page cache, there is no commonality between a virtual mapping of a _file_ (or metadata) and a virtual mapping of a _disk_).*

Although these desirable changes were not expected until 2.5.x, finally Linus decided to integrate a bunch of Andrea Arcangeli patches, plus changes on his own, and released 2.4.10, **which finally unified the page and buffer cache** (Figure unified). Thus, an important improvement in I/O operations is expected, altogether with a better VM tuning, especially for memory shortage situation.

Figure: Unified to page cache
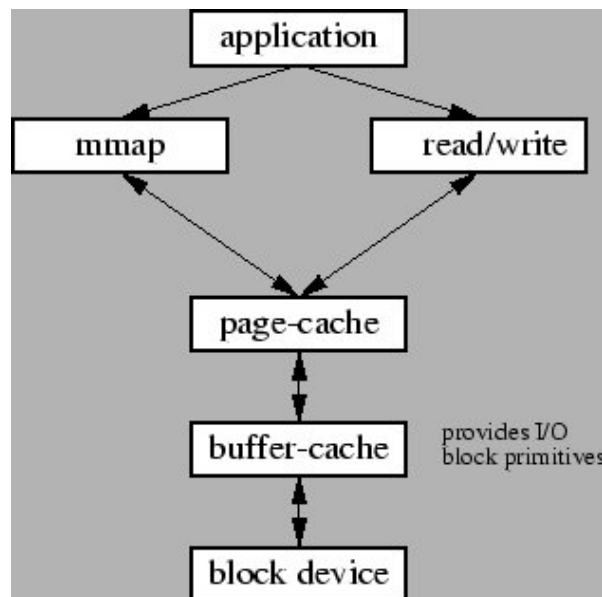
Journaling File Systems

The standard file system for Linux was *ext2f*s. Ext2 was designed by Wayne Davidson with collaboration from Stephen Tweedie and Theodore Ts'o. It is an improvement of the previous *ext* file system designed by Rémy Card. The ext2fs is an i−node based file−system, the i−node maintains the metadata of the file and the pointers to the actual data blocks.

To speed up the performance of I/O operations, data is temporally allocated in RAM memory by means of the buffer cache and page cache subsystem. The problem appears if there is a system crash of electric outage before the modified data in the cache (dirty buffers) have been written to disk. This would cause an inconsistency of the whole file system, for example a new file that wasn't created in the disk or files that were removed but their i−nodes and data blocks still remain in the disk.

The **fsck** (*file system check*) was the common recovering tool to resolve the inconsistencies. But fsck has to scan the whole disk partition and check the interdependencies among i−nodes, data blocks and directory contents. With the enlargement of disk's capacity, restoring the consistency of the file system has become a very time consuming task, which creates serious problems of availability for large servers. This is the main reason for the file systems to inherit database transaction and recover technologies, and thus the appearance of *Journaling File Systems* or *Journal File*

Systems.

A journaling file system is a fault–resilient file system in which data integrity is ensured because updates to files' metadata are written to a serial log on disk before the original disk blocks are updated. In the event of a system failure, a full journaling file system ensures that the file system consistency is restored. The most common approach is a method of *journaling* or *logging* the metadata of files. With logging, whenever something is changed in the metadata of a file, this new attribute information is logged into a reserved area of the file system. The file system will write the actual data to the disk only after the write of the metadata to the log is complete. When a system crash occurs, the system recovery code will analyse the metadata log and try to clean up only those inconsistent files by replaying the log file**.**

The earliest journaling file systems, created in the mid–1980s, included Veritas (VxFS), Tolerant, and IBM's JFS. With increasing demands being placed on file systems to support terabytes of data, thousands upon thousands of files per directory and 64–bit capability, the interest in journaling file system for Linux has growth over the last years.

Linux has three new contenders in the journaling file systems in the past few months: ReiserFS[7] from Namesys, XFS[8] from SGI, JFS[9] from IBM and Ext3[10] developed by Stephen Tweedie, co–creator of Ext2.

While ReiserFS is a complete new file system written from scratch, XFS, JFS and Ext3 are derived from commercial products or existing file systems. XFS is based, and partially shares the same code, on the system developed by SGI for its workstations and servers. JFS was designed and developed by IBM for its OS/2 Warp, which is itself derived from the AIX file system.

ReiserFS is the only one included in the standard Linux kernel tree, the others are planned to be included in version 2.5, although XFS and JFS are fully functional, officially released as kernel patches, and in production quality.

Ext3 is an extension to ext2. It adds two independent modules, a transaction and a logging module. Ext3 is close to is final version, RedHat 7.2 already includes it as option and will be the official file system of Red Hat distributions.

B–Trees

The basic tool for improving performance compared to traditional UNIX file systems is to avoid the use of linked lists or bitmaps –for free blocks, directory entries and data block addressing– that have inherent scalability problem (typical complexity for search is O(n)) and are not adequate for new, vary large capacity disks. All the new systems use *Balanced Trees* (*B–Trees*) or variation of them (*B+Trees*).

Balanced tree is an well studied structure, they are more robust in their performance but at the same time management and balancing algorithms become more complex. The B+Tree structure has been used on databases indexing structures for a long time. This structure provided databases with a scalable and fast manner to access their records. The + sign means that the B–Tree is a modified version of the original that:

- Place all keys at the leaves.
- Leaves nodes can be linked together.
- Internal nodes and leaves can be of different sizes.
- Never needs to change parent if a key in a leaf is deleted.
- Makes sequential operations easier and cheaper.

ReiserFS

ReiserFS is based on fast balanced trees (B+Tree) to organise file system objects. File systems objects are the structures used to maintain file information: access time, file permissions, etc. In other words, the information contained within an i–node, directories and the files' data. ReiserFS calls those objects, *stat data* items, *directory* items and *direct/indirect* items, respectively. ReiserFS only provides metadata journaling. In case of a non–planned reboot, data in blocks that were being used at the time of the crash could have been corrupted; thus ReiserFS does not guarantee the file contents themselves are uncorrupted.

Unformatted nodes are logical blocks with no given format, used to store file data, and the *direct item*s consist of file data itself. Also, those items are of variable size and stored within the leaf nodes of the tree, sometimes with others in

case there is enough space within the node. File information is stored close to file data, since the file system always tries to put stat data items and the direct/indirect items of the same file together. Opposed to direct items, the file data pointed by indirect items is not stored within the tree. This special management of direct items is due to small file support: *tail packing*.

*Tail packin*g is a special ReiserFS feature. Tails are files that are smaller than a logical block, or the trailing portions of files that do not fill up a complete block. To save disk space, ReiserFS uses tail packing to hold tails into as small a space as possible. Generally, this allows a ReiserFS to hold around 5% more than an equivalent Ext2 file system. The direct items are intended to keep small file data and even the tails of the files. Therefore, several tails could be kept within the same leaf node.

ReiserFS has an excellent small–file performance because it is able to incorporate these tails into its B–Tree so that they are really close to the *stat data*. Since tails do not fill up a complete block, they can waste disk space.

The problem is that using this technique of keeping the file's tails together would increase external fragmentation, since the file data is now further from the file tail. Moreover, the task of packing tails is time–consuming and leads to performance penalties. This is a consequence of the memory shifts needed when someone appends data to a file. Namesys realised this problem and allows system administrator to disable the tail packing by specifying the *notail* option at the time the file system is mounted or event remounted.

ReiserFS uses fixed size block (4KB) oriented allocation that affects negatively to the performance of I/O operations of large files. The other weakness of ReiserFS is that the sparse file performance is significantly worse compared to ext2, although Namesys is working on optimising this case.

Figure: Block based allocation
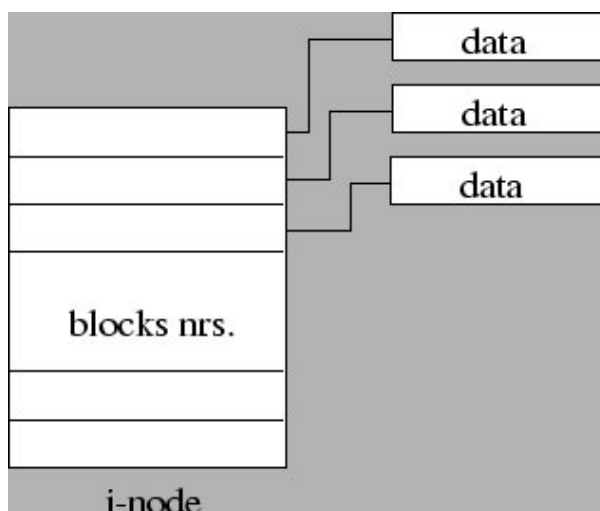
XFS

On May 1 2001, SGI made available Release 1.0 of its journaling XFS file system for Linux. XFS, is recognised by its support for large disk farm and very high I/O throughput (tested up to 7GB/sec). XFS was developed for the IRIX 5.3 SGI Unix operating system, its first version was introduced in December 1994. The target of the file system was to support vary large files and high throughput for real time video recording and playing.

To increase the scalability of the file system XFS uses of B+Trees extensively. They are used for tracking free extents, index directories and to keep track of dynamically allocated i–nodes scattered throughout the file system. In addition, XFS uses an asynchronous write ahead logging scheme for protecting metadata updates and allowing fast file system recovery.

XFS uses an extent based space allocation, and it has features like delayed allocation, space pre–allocation and space coalescing on deletion, and goes to great lengths in attempting to layout files using the largest extents possible. To

make the management of large amounts of contiguous space in a file efficient, XFS uses very large extent descriptors in the file extent map. Each descriptor can describe up to two million file system blocks. Describing large numbers of blocks with a single extent descriptor eliminates the CPU overhead of scanning entries in the extent map to determine whether blocks in the file are contiguous, it can simply read the length of the extent rather than looking at each entry to see if it is contiguous with the previous entry.

Figure: Extent based allocation
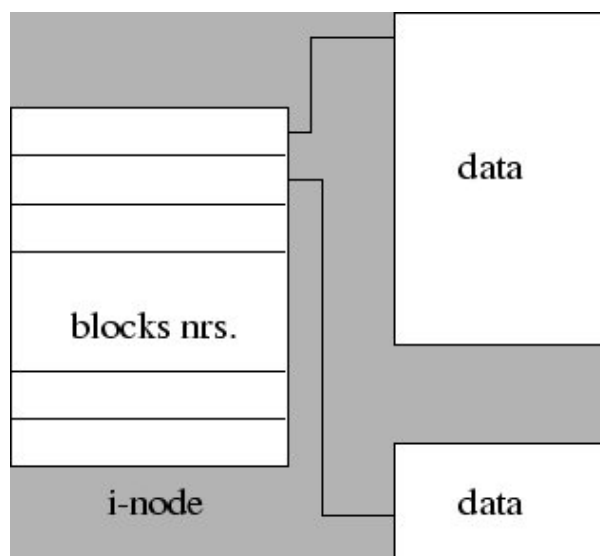
XFS allows variable sized blocks, from 512 bytes to 64 kilobytes on a per file system basis. Changing the file system block size can vary fragmentation. File systems with large numbers of small files typically use smaller block sizes in order to avoid wasting space via internal fragmentation. File systems with large files tend to make the opposite choice and use large block sizes in order to reduce external fragmentation of the file system and their files' extents.

XFS is complex chunk of code on IRIX and very IRIX−centric, so in porting to Linux this interface was redesigned and rewritten from scratch. The result is the Linux *pagebuf* module, it provides the interface between XFS and the virtual memory subsystem and also between XFS and the Linux block device layer.

XFS support ACL's (integrated with the Samba server) and transactional quotes. On Linux it supports quotes per−group instead of IRIX per−project quota, as this is the way quota are implemented in Linux file systems (and Linux has no equivalent concept to "projects"). More esoteric features of XFS on IRIX that provide file system services customized for specific demanding applications (e.g. real−time video serving) have not been ported to Linux yet.

The normal mode of operation for XFS is to use an *asynchronously written log*. It still ensures that the write ahead logging protocol is followed in that modified data cannot be flushed to disk until the data is committed to the on−disk log. XFS gains two things by writing the log asynchronously:

1. Multiple updates can be batched into a single log write. This increases the efficiency of the log writes with respect to the underlying disk array.
2. The performance of metadata updates is normally made independent of the speed of the underlying drives. This independence is limited by the amount of buffering dedicated to the log, but it is far better than the synchronous updates of older file systems.

XFS also has a fairly extensive set of userspace tools for dumping, restoring, repairing, growing, snapshotting, tools for using ACLs and disk quotas, etc.

I just received an email from the maintainer of the XFS FAQ, Seth Mos, who clarifies a couple of XFS issues for Linux. Thanks.
As the XFS FAQ maintainer i would like to make a small note.

XFS only supports blocksize where blocksize == pagesize == 4k.

There currently is no code to do the block < page size but it is on the todo list with low priority. It will be done probably this year since disks formatted under IRIX frequently use 512 Byte blocks. So this support is needed for extra compatibility. Disks formatted under IRIX with 4K blocks and v2 directories will be readable under both IRIX and Linux without problems.

The block > page size is way down the list as that requires a lot of effort and changes in the core linux kernel as well.

Cheers

JFS

IBM introduced its UNIX file system as the Journaled File System (JFS) with the initial release of AIX Version 3.1. It has now introduced a second file system that is to run on AIX systems called Enhanced Journaled File System (JFS2), which is available in AIX Version 5.0 and later versions. The JFS open source code on originated from that currently shipping with the OS/2 Warp Server for e−business.

JFS is tailored primarily for the high throughput and reliability requirements of servers. JFS uses extent−based addressing structures, along with clustered 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, rooted in the i−node and keyed by logical offset within the file.

JFS logs are maintained in each file system and used to record information about operations on metadata. The log has a format that also is set by the file system creation utility.

JFS logging semantics are such that, when a file system operation involving meta−data changes returns a successful return code, the effects of the operation have been already committed to the file system and will be seen even if the system crashes inmediately after the operation.

The old logging style introduced a synchronous write to the log disk into each i−node or VFS operation that modifies meta−data. In terms of performance, it is a performance disadvantage when compared to other journaling file systems, such as Veritas VxFS and XFS, which use different logging styles and lazily write log data to disk. When 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 and it currently provides asynchronous logging, which increases performance of the file system.

JFS supports block sizes of 512, 1024, 2048, and 4096 bytes on a per−file system basis. Smaller block sizes reduce the amount of internal fragmentation. 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.

JFS dynamically allocates space for disk i−nodes as required, freeing the space when it is no longer needed. Two different directory organizations are provided.

1. The first organization is used for small directories and stores the directory contents within the directory's i−node. This eliminates the need for separate directory block I/O as well as the need to allocate separate storage.
2. 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.

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 others unwritten 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.

Ext3

Ext3 is compatible with Ext2, actually it is an ext2fs with a journal file. Ext3 is the *half* of a journaling file system mentioned, it is a layer atop the traditional ext2 file system that does keep a journal file of disk activity so that recovery from an improper shutdown is much quicker than that of ext2 alone. But, because it is tied to ext2, it suffers some of the limitations of the older ext2 system and therefore does not exploit all the potential of the pure journaling file systems, for example, it is still block based and uses sequential search of file names in directories.

Its major advantages are:

- Ext3 journal and maintain order consistency in both, the data and the metadata. Differently to the above journal file systems, consistency is assured for the content of the file as well. The level of journaling can be controlled with *mount* options[11].
- Ext3 partitions do not have a file structure different from ext2, so porting or backing out to the old system, by choice or in the event the journal file were to become corrupted, is straightforward.

Ext3 reserves one of the special ext2 i–nodes for storing the journal log, but the journal can be on any i–node in any file system or it can be on any arbitrary sub–range, set of contiguous blocks on any block device. It is possible to have multiple file systems sharing the same journal.

The journal file job is to record the new contents of file system metadata blocks while it is in the process of committing transactions. The only other requirement is that the system must assure that can commit the transactions atomically.

Three type of data blocks are written to the journal:

1. Metadata,
2. Descriptor blocks, and
3. Header blocks.

A journal metadata block contains the entire single block of file system metadata as updated by a transaction. Whenever a small change is done to the file systems, an entire journal block has to be written. However, it is relatively cheap because journal I/O operations can be batched into large clusters and the blocks can be written directly from the page cache system by exploiting the *buffer_head* structure.

Descriptor blocks describe other metadata journal blocks so the recovery mechanism can copy the metadata back to the main file system. They are written before any change to the journal metadata is done.

Finally, the header blocks describe the head and tail of the journal plus a sequence number to guarantee write ordering during recovery.

Performance and Conclusions

Different benchmarks (see Resources below) have shown that XFS and ReiserFS have a very good performance compared to the well–tested and optimised Ext2. Ext3 showed that is slower but getting closer to Ext2 performance. We expect that the performance will improve considerably over the following months. On the other hand, JFS get the worst results in all benchmarks, not only in performance but it had also some stability problems in the Linux port.

XFS, ReiserFS and Ext3 have demonstrated they are excellent and reliable file systems. There is an important area where XFS has higher performance: I/O operation on large files, specially compared to its closer competitor, ReiserFS. This is understandable and subjected to change over the time, ReiserFS uses the 2.4 generic read and write Linux, while XFS has ported sophisticated IRIX I/O operations to Linux, the most important is the extent based allocation and direct I/O operations. Furthermore, the current version of ReiserFS does a complete tree traversal for every 4 KB block it writes, and then inserts one pointer at a time, which introduces an important overhead of balancing the tree while it copies data around.

For operation on small files, normally between 100 and 10.000 bytes, ReiserFS has shown that has the best results, if the affected files are not in the cache yet (as occurs during booting). In case you are reading files that are already

cached in RAM, the difference is almost negligible for Ext2, Ext3, ReiserFS and XFS.

Among all journal file systems, ReiserFS is the only one that is included in the standard Linus tree since 2.4.1 and SuSE supports it for more than two years now. However, Ext3 is going the be the standard file system for Red Hat and XFS is being used in large servers, specially in the Hollywood industry, due mainly to the influence of SGI in that market. IBM has to put a lot of efforts on JFS in they want to see it in the mainstream, although is a valid alternative for migrating AIX and OS/2 installation to Linux.

Resources

- [Ext3 architecture](#)[12]
- [Introduction to Linux Journal File Systems](#)[13]
- [XFS Home Page](#)[14]
- [JFS Home Page](#)[9]
- [ReiserFS Home Page](#)[15]
- [Storage Foundry](#)[16]
- [OS News](#)[17]

Benchmarks

- [Ext2, Ext3, ReiserFS, XFS and JFS benchmarks](#)[18]
- [Ext2, ReiserFS and XFS Benchmarks](#)[6]
- [Pruebas con XFS, ReiserFS, Ext2FS, y FAT32](#)[19]
- [Namesys benchmarks](#)[20]
- [Ext2, XFS, ReiserFS and JFS Mongo Benchmarks](#)[21]

About this document ...

This document was generated using the **_LaTeX_2HTML**[22] translator Version 2K.1beta (1.48)

Ricardo Galli 2001−10−22

Lista de enlaces de este artículo:

1. http://www.ati.es/novatica/
2. http://www.upgrade−cepis.org/issues/2001/6/upgrade−vII−6.html
3. http://www.upgrade−cepis.org/issues/2001/6/up2−6Galli.pdf
4. http://bulma.net/body.phtml?nIdNoticia=1153
5. http://bulma.net/body.phtml?nIdNoticia=1167
6. http://bulma.net/body.phtml?nIdNoticia=642
7. http://www.namesys.com
8. http://bulma.net/http//oss.sgi.com/projects/xfs/
9. http://oss.software.ibm.com/developerworks/opensource/jfs/
10. http://e2fsprogs.sourceforge.net/ext2.html
11. http://bulma.net/body.phtml?nIdNoticia=985
12. ftp://ftp.kernel.org/pub/linux/kernel/people/sct/ext3/
13. http://www.linuxgazette.com/issue55/florido.html
14. http://oss.sgi.com/projects/xfs/
15. http://www.namesys.com/
16. http://sourceforge.net/foundry/storage/
17. http://www.osnews.com/story.php?news_id=69
18. http://www.mandrakeforum.org/article.php?sid=1212
19. http://bulma.net/body.phtml?nIdNoticia=626
20. http://www.namesys.com/benchmarks/benchmark−results.html
21. http://bulma.net/body.phtml?nIdNoticia=648
22. http://www−texdev.mpce.mq.edu.au/l2h/docs/manual/

E–mail del autor: gallir _ARROBA_ uib.es
Podrás encontrar este artículo e información adicional en: http://bulma.net/body.phtml?nIdNoticia=1154