Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

The GFS2 Filesystem

Steven Whitehouse
Red Hat, Inc.
swhiteho@redhat.com

Abstract

The GFS2 filesystem is a symmetric cluster filesystem designed to provide a high performance means of sharing a filesystem between nodes. This paper will give an overview of GFS2's make subsystems, features and differences from GFS1 before considering more recent developments in GFS2 such as the new on-disk layout of journaled files, the GFS2 metadata filesystem, and what can be done with it, fast & fuzzy statfs, optimisations of `readdir/getdents64` and optimisations of glocks (cluster locking). Finally, some possible future developments will be outlined.

To get the most from this talk you will need a good background in the basics of Linux filesystem internals and clustering concepts such as quorum and distributed locking.

1 Introduction

The GFS2 filesystem is a 64bit, symmetric cluster filesystem which is derived from the earlier GFS filesystem. It is primarily designed for Storage Area Network (SAN) applications in which each node in a GFS2 cluster has equal access to the storage. In GFS and GFS2 there is no such concept as a metadata server, all nodes run identical software and any node can potentially perform the same functions as any other node in the cluster.

In order to limit access to areas of the storage to maintain filesystem integrity, a lock manager is used. In GFS2 this is a distributed lock manager (DLM) [1] based upon the VAX DLM API. The Red Hat Cluster Suite provides the underlying cluster services (quorum, fencing) upon which the DLM and GFS2 depend.

It is also possible to use GFS2 as a local filesystem with the `lock_nolock` lock manager instead of the DLM. The locking subsystem is modular and is thus easily substituted in case of a future need of a more specialised lock manager.

2 Historical Detail

The original GFS [6] filesystem was developed by Matt O'Keefe's research group in the University of Minnesota. It used SCSI reservations to control access to the storage and ran on SGI's IRIX.

Later versions of GFS [5] were ported to Linux, mainly because the group found there was considerable advantage during development due to the easy availability of the source code. The locking subsystem was developed to give finer grained locking, initially by the use of special firmware in the disk drives (and eventually, also RAID controllers) which was intended to become a SCSI standard called dmep. There was also a network based version of dmep called memexp. Both of these standards worked on the basis of atomically updated areas of memory based upon a "compare and exchange" operation.

Later when it was found that most people preferred the network based locking manager, the Grand Unified Locking Manager, gulm, was created improving the performance over the original memexp based locking. This was the default locking manager for GFS until the DLM (see [1]) was written by Patrick Caulfield and Dave Teigland.

Sistina Software Inc, was set up by Matt O'Keefe and began to exploit GFS commercially in late 1999/early 2000. Ken Preslan was the chief architect of that version of GFS (see [5]) as well as the version which forms Red Hat's current product. Red Hat acquired Sistina Software Inc in late 2003 and integrated the GFS filesystem into its existing product lines.

During the development and subsequent deployment of the GFS filesystem, a number of lessons were learned about where the performance and administrative problems occur. As a result, in early 2005 the GFS2 filesystem was designed and written, initially by Ken Preslan

and more recently by the author, to improve upon the original design of GFS.

The GFS2 filesystem was submitted for inclusion in Linus' kernel and after a lengthy period of code review and modification, was accepted into 2.6.16.

3 The on-disk format

The on-disk format of GFS2 has, intentionally, stayed very much the same as that of GFS. The filesystem is big-endian on disk and most of the major structures have stayed compatible in terms of offsets of the fields common to both versions, which is most of them, in fact.

It is thus possible to perform an in-place upgrade of GFS to GFS2. When a few extra blocks are required for some of the *per node* files (see the metafs filesystem, Subsection 3.5) these can be found by shrinking the areas of the disk originally allocated to journals in GFS. As a result, even a full GFS filesystem can be upgraded to GFS2 without needing the addition of further storage.

3.1 The superblock

GFS2's superblock is offset from the start of the disk by 64k of unused space. The reason for this is entirely historical in that in the dim and distant past, Linux used to read the first few sectors of the disk in the VFS mount code before control had passed to a filesystem. As a result, this data was being cached by the Linux buffer cache without any cluster locking. More recent versions of GFS were able to get around this by invalidating these sectors at mount time, and more recently still, the need for this gap has gone away entirely. It is retained only for backward compatibility reasons.

3.2 Resource groups

Following the superblock are a number of resource groups. These are similar to ext2/3 block groups in that their intent is to divide the disk into areas which helps to group together similar allocations. Additionally in GFS2, the resource groups allow parallel allocation from different nodes simultaneously as the locking granularity is one lock per resource group.

On-disk, each resource group consists of a header block with some summary information followed by a number of blocks containing the allocation bitmaps. There are two bits in the bitmap for each block in the resource group. This is followed by the blocks for which the resource group controls the allocation.

Bit Pattern	Block State
00	Free
01	Allocated non-inode block
10	Unlinked (still allocated) inode
11	Allocated inode

Table 1: GFS2 Resource Group bitmap states

The two bits are nominally *allocated/free* and *data (non-inode)/inode* with the exception that the *free inode* state is used to indicate inodes which are unlinked, but still open.

In GFS2 all metadata blocks start with a common header which includes fields indicating the type of the metadata block for ease of parsing and these are also used extensively in checking for run-time errors.

Each resource group has a set of flags associated with it which are intended to be used in the future as part of a system to allow in-place upgrade of the filesystem. It is possible to mark resource groups such that they will no longer be used for allocations. This is the first part of a plan that will allow migration of the content of a resource group to eventually allow filesystem shrink and similar features.

3.3 Inodes

GFS2's inodes have retained a very similar form to those of GFS in that each one spans an entire filesystem block with the remainder of the block being filled either with data (a "stuffed" inode) or with the first set of pointers in the metadata tree.

GFS2 has also inherited GFS's equal height metadata tree. This was designed to provide constant time access to the different areas of the file. Filesystems such as ext3, for example, have different depths of indirect pointers according to the file offset whereas in GFS2, the tree is constant in depth no matter what the file offset is.

Initially the tree is formed by the pointers which can be fitted into the spare space in the inode block, and is then

grown by adding another layer to the tree whenever the current tree size proves to be insufficient.

Like all the other metadata blocks in GFS2, the indirect pointer blocks also have the common metadata header. This unfortunately also means that the number of pointers they contain is no longer an integer power of two. This, again, was to keep compatibility with GFS and in the future we eventually intend to move to an extent based system rather than change the number of pointers in the indirect blocks.

3.3.1 Attributes

GFS2 supports the standard get/change attributes `ioctl()` used by ext2/3 and many other Linux filesystems. This allows setting or querying the attributes listed in Table 2.

As a result GFS2 is directly supported by the `lsattr(1)` and `chattr(1)` commands. The hashed directory flag, `I`, indicates whether a directory is hashed or not. All directories which have grown beyond a certain size are hashed and section 3.4 gives further details.

3.3.2 Extended Attributes & ACLs

GFS2 supports extended attribute types *user*, *system* and *security*. It is therefore possible to run selinux on a GFS2 filesystem.

GFS2 also supports POSIX ACLs.

3.4 Directories

GFS2's directories are based upon the paper "Extendible Hashing" by Fagin [3]. Using this scheme GFS2 has a fast directory lookup time for individual file names which scales to very large directories. Before ext3 gained hashed directories, it was the single most common reason for using GFS as a single node filesystem.

When a new GFS2 directory is created, it is "stuffed," in other words the directory entries are pushed into the same disk block as the inode. Each entry is similar to an ext3 directory entry in that it consists of a fixed length part followed by a variable length part containing the file name. The fixed length part contains fields to indicate the total length of the entry and the offset to the next entry.

Once enough entries have been added that it's no longer possible to fit them all in the directory block itself, the directory is turned into a hashed directory. In this case, the hash table takes the place of the directory entries in the directory block and the entries are moved into a directory "leaf" block.

In the first instance, the hash table size is chosen to be half the size of the inode disk block. This allows it to coexist with the inode in that block. Each entry in the hash table is a pointer to a leaf block which contains a number of directory entries. Initially, all the pointers in the hash table point to the same leaf block. When that leaf block fills up, half the pointers are changed to point to a new block and the existing directory entries moved to the new leaf block, or left in the existing one according to their respective hash values.

Eventually, all the pointers will point to different blocks, assuming that the hash function (in this case a CRC-32) has resulted in a reasonably even distribution of directory entries. At this point the directory hash table is removed from the inode block and written into what would be the data blocks of a regular file. This allows the doubling in size of the hash table which then occurs each time all the pointers are exhausted.

Eventually when the directory hash table hash reached a maximum size, further entries are added by chaining leaf blocks to the existing directory leaf blocks.

As a result, for all but the largest directories, a single hash lookup results in reading the directory block which contains the required entry.

Things are a bit more complicated when it comes to the `readdir` function, as this requires that the entries in each hash chain are sorted according to their hash value (which is also used as the file position for `lseek`) in order to avoid the problem of seeing entries twice, or missing them entirely in case a directory is expanded during a set of repeated calls to `readdir`. This is discussed further in the section on future developments.

3.5 The metadata filesystem

There are a number of special files created by `mkfs.gfs2` which are used to store additional metadata related to the filesystem. These are accessible by

Attribute	Symbol	Get or Set
Append Only	a	Get and set on regular inodes
Immutable	i	Get and set on regular inodes
Journaling	j	Set on regular files, get on all inodes
No atime	A	Get and set on all inodes
Sync Updates	S	Get and set on regular files
Hashed dir	I	Get on directories only

Table 2: GFS2 Attributes

mounting the `gfs2meta` filesystem specifying a suitable gfs2 filesystem. Normally users would not do this operation directly since it is done by the GFS2 tools as and when required.

Under the root directory of the metadata filesystem (called the master directory in order that it is not confused with the real root directory) are a number of files and directories. The most important of these is the resource index (rindex) whose fixed-size entries list the disk locations of the resource groups.

3.5.1 Journals

Below the master directory there is a subdirectory which contains all the journals belonging to the different nodes of a GFS2 filesystem. The maximum number of nodes which can mount the filesystem simultaneously is set by the number of journals in this subdirectory. New journals can be created simply by adding a suitably initialised file to this directory. This is done (along with the other adjustments required) by the `gfs2_jadd` tool.

3.5.2 Quota file

The quota file contains the system wide summary of all the quota information. This information is synced periodically and also based on how close each user is to their actual quota allocation. This means that although it is possible for a user to exceed their allocated quota (by a maximum of two times) this is in practise extremely unlikely to occur. The time period over which syncs of quota take place are adjustable via sysfs.

3.5.3 statfs

The statfs files (there is a master one, and one in each `per_node` subdirectory) contain the information required to give a fast (although not 100% accurate) result for the `statfs` system call. For large filesystems mounted on a number of nodes, the conventional approach to `statfs` (i.e., iterating through all the resource groups) requires a lot of CPU time and can trigger a lot of I/O making it rather inefficient. To avoid this, GFS2 by default uses these files to keep an approximation of the true figure which is periodically synced back up to the master file.

There is a sysfs interface to allow adjustment of the sync period or alternatively turn off the fast & fuzzy `statfs` and go back to the original 100% correct, but slower implementation.

3.5.4 inum

These files are used to allocate the `no_formal_ino` part of GFS2's `struct gfs2_inum` structure. This is effectively a version number which is mostly used by NFS, although it is also present in the directory entry structure as well. The aim is to give each inode an additional number to make it unique over time. The master inum file is used to allocate ranges to each node, which are then replenished when they've been used up.

4 Locking

Whereas most filesystems define an on-disk format which has to be largely invariant and are then free to change their internal implementation as needs arise, GFS2 also has to specify its locking with the same degree of care as for the on-disk format to ensure future compatibility.

Lock type	Use
Non-disk	mount/umount/recovery
Meta	The superblock
Inode	Inode metadata & data
Iopen	Inode last closer detection
Rgrp	Resource group metadata
Trans	Transaction lock
Flock	`flock(2)` syscall
Quota	Quota operations
Journal	Journal mutex

Table 3: GFS2 lock types

GFS2 internally divides its cluster locks (known as glocks) into several types, and within each type a 64 bit lock number identifies individual locks. A lock name is the concatenation of the glock type and glock number and this is converted into an ASCII string to be passed to the DLM. The DLM refers to these locks as resources. Each resource is associated with a lock value block (LVB) which is a small area of memory which may be used to hold a few bytes of data relevant to that resource. Lock requests are sent to the DLM by GFS2 for each resource which GFS2 wants to acquire a lock upon.

All holders of DLM locks may potentially receive callbacks from other intending holders of locks should the DLM receive a request for a lock on a particular resource with a conflicting mode. This is used to trigger an action such as writing back dirty data and/or invalidating pages in the page cache when an inode's lock is being requested by another node.

GFS2 uses three lock modes internally, *exclusive*, *shared* and *deferred*. The *deferred* lock mode is effectively another shared lock mode which is incompatible with the normal *shared* lock mode. It is used to ensure that direct I/O is cluster coherent by forcing any cached pages for an inode to be disposed of on all nodes in the cluster before direct I/O commences. These are mapped to the DLMs lock modes (only three of the six modes are used) as shown in table 4.

The DLM's `DLM_LOCK_NL` (*Null*) lock mode is used as a reference count on the resource to maintain the value of the LVB for that resource. Locks for which GFS2 doesn't maintain a reference count in this way (or are unlocked) may have the content of their LVBs set to zero upon the next use of that particular lock.

5 NFS

The GFS2 interface to NFS has been carefully designed to allow failover from one GFS2/NFS server to another, even if those GFS2/NFS servers have CPUs of a different endianness. In order to allow this, the filehandles must be constructed using the `fsid=` method. GFS2 will automatically convert endianness during the decoding of the filehandles.

6 Application writers' notes

In order to ensure the best possible performance of an application on GFS2, there are some basic principles which need to be followed. The advice given in this section can be considered a FAQ for application writers and system administrators of GFS2 filesystems.

There are two simple rules to follow:

- Make maximum use of caching
- Watch out for lock contention

When GFS2 performs an operation on an inode, it first has to gain the necessary locks, and since this potentially requires a journal flush and/or page cache invalidate on a remote node, this can be an expensive operation. As a result for best performance in a cluster scenario it is vitally important to ensure that applications do not contend for locks for the same set of files wherever possible.

GFS2 uses one lock per inode, so that directories may become points of contention in case of large numbers of inserts and deletes occurring in the same directory from multiple nodes. This can rapidly degrade performance.

The single most common question asked relating to GFS2 performance is how to run an smtp/imap email server in an efficient manner. Ideally the spool directory is broken up into a number of subdirectories each of which can be cached separately resulting in fewer locks being bounced from node to node and less data being flushed when it does happen. It is also useful if the locality of the nodes to a particular set of directories can be enhanced using other methods (e.g. DNS) in the case of an email server which serves multiple virtual hosts.

GFS2 Lock Mode	DLM Lock Mode
LM_ST_EXCLUSIVE	DLM_LOCK_EX (exclusive)
LM_ST_SHARED	DLM_LOCK_PR (protected read)
LM_ST_DEFERRED	DLM_LOCK_CW (concurrent write)

Table 4: GFS2/DLM Lock modes

6.1 fcntl(2) caveat

When using the fcntl(2) command F_GETLK note that although the PID of the process will be returned in the l_pid field of the struct flock, the process blocking the lock may not be on the local node. There is currently no way to find out which node the lock blocking process is actually running on, unless the application defines its own method.

The various fcntl(2) operations are provided via the userspace gfs2_controld which relies upon openais for its communications layer rather than using the DLM. This system keeps on each node a complete copy of the fcntl(2) lock state, with new lock requests being passed around the cluster using a token passing protocol which is part of openais. This protocol ensures that each node will see the lock requests in the same order as every other node.

It is faster (for whole file locking) for applications to use flock(2) locks which do use the DLM. In addition it is possible to disable the cluster fcntl(2) locks and make them local to each node, even in a cluster configuration for higher performance. This is useful if you know that the application will only need to lock against processes local to the node.

6.2 Using the DLM from an application

The DLM is available through a userland interface in order that applications can take advantage of its cluster locking facility. Applications can open and use lockspaces which are independent of those used by GFS2.

7 Future Development

7.1 readdir

Currently we have already completed some work relating to speeding up readdir and also considered the way in which readdir is used in combination with other syscalls, such as stat.

There has also been some discussion (and more recently in a thread on lkml [2]) relating to the readdir interface to userspace (currently via the getdents64 syscall) and the other two interfaces to NFS via the struct export_operations. At the time of writing, there are no firm proposals to change any of these, but there are a number of issues with the current interface which might be solved with a suitable new interface. Such things include:

- Eliminating the sorting in GFS2's readdir for the NFS getname operation where ordering is irrelevant.
- Boosting performance by returning more entries at once.
- Optionally returning stat information at the same time as the directory entry (or at least indicating the intent to call stat soon).
- Reducing the problem of lseek in directories with insert and delete of entries (does it result in seeing entries twice or not at all?).

7.2 inotify & dnotify

GFS2 does not support inotify nor do we have any plans to support this feature. We would like to support dnotify if we are able to design a scheme which is both scalable and cluster coherent.

7.3 Performance

There are a number of ongoing investigations into various aspects of GFS2's performance with a view to gaining greater insight into where there is scope for further improvement. Currently we are focusing upon increasing the speed of file creations via open(2).

8 Resources

GFS2 is included in the Fedora Core 6 kernel (and above). To use GFS2 in Fedora Core 6, install the `gfs2-utils` and `cman` packages. The `cman` package is not required to use GFS2 as a local filesystem.

There are two GFS2 git trees available at kernel.org. Generally the one to look at is the `-nmw` (next merge window) tree [4] as that contains all the latest developments. This tree is also included in Andrew Morton's `-mm` tree. The `-fixes` git tree is used to send occasional fixes to Linus between merge windows and may not always be up-to-date.

The user tools are available from Red Hat's CVS at: `http://sources.redhat.com/cgi-bin/cvsweb.cgi/cluster/?cvsroot=cluster`

References

[1] "DLM—Kernel Distributed Lock Manager," Patrick Caulfield, *Minneapolis Cluster Summit 2004*, `http://sources.redhat.com/cluster/events/summit2004/presentations.html#mozTocId443696`

[2] Linux Kernel Mailing List. Thread "If not readdir() then what?" started by Ulrich Drepper on Sat, 7 Apr 2007.

[3] "Extendible Hashing," Fagin, et al., *ACM Transactions on Database Systems*, Sept., 1979.

[4] The GFS2 git tree: `git://git.kernel.org/pub/scm/linux/git/steve/gfs2-2.6-nmw.git` (next merge window)

[5] "64-bit, Shared Disk Filesystem for Linux," Kenneth W. Preslan, et al., *Proceedings of the Seventh NASA Goddard Conference on Mass Storage*, San Diego, CA, March, 1999.

[6] "The Global File System," S. Soltis, T. Ruwart, and M. O'Keefe, *Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, September, 1996.