Testing and verification of cluster filesystems

Steven Whitehouse
Red Hat, UK
swhiteho@redhat.com

Abstract

Although software testing can never prove a program is correct, it can catch many errors early and plays an important part in the process of declaring a program "stable" and ready for release. Cluster filesystems (e.g., GFS2), by their very nature require hardware-intensive test environments, and are thus also expensive to test. This tends to limit test coverage compared with their simpler, single-node counterparts (ext2/3/4, xfs, etc).

Since reliability is a key feature of any filesystem, this paper considers a number of techniques which may be used to help simulate a cluster on a single node, reducing the cost of testing and increasing the coverage in the process. Although GFS2 is taken as the example filesystem, the techniques described are generic and apply to any similar filesystem.

1 Introduction

There is a catch-22 situation in filesystem development where users are unsure of trusting their data to a new filesystem until its been in use by other people for a period of time. Software verification and testing is one way to try and break this cycle and to help build confidence.

Testing cluster filesystems is particularly problematic because they, by definition, require clusters to test on, and since these are expensive to run it is often difficult to get enough testing time, particularly on larger configurations.

In addition, it is often those running larger clusters who are the most sensitive to downtime, and thus reliability is often at the top of the list of requirements.

When supporting filesystems in the field, things will occasionally go wrong, and in that case it is imperative to identify the cause of the failure as quickly as possible, and thus to be able to rectify it. Filesystems therefore should be designed with this consideration in mind.

This paper considers the possible methods by which a cluster filesystem (taking GFS2 as the example) can be tested and debugged. We consider some of the more recent developments based around the Linux tracing infrastructure. We include performance testing as well as testing in order to verify functional correctness.

1.1 Proof

In an ideal world, it would be possible to prove the correctness of software. In reality, it is impossible to prove the correctness of anything but the simplest software; the main difficulty is that the combination of all the possible inputs to the system and all the possible states is so large that it is impossible to prove the system correct within a reasonable amount of time. This prevents exhaustive testing, but even with techniques to try and cut down on the total number of states which need to be tested, it is still usually too difficult even for fairly simple systems.

A further complication arises from the possibility that there is an error in the original specification, and in that case a system maybe provably correct, but still fail due to some unforeseen circumstance.

1.2 Source analysis

By carefully annotating the source, some errors can be removed at compile time. The `sparse` tool can detect a number of issues relating to endian conversion, arithmetic with invalid types (e.g., bitwise) and similar issues. It isn't able to detect a huge range of errors, but it does catch a number of basic issues, given a disciplined to annotation of the source code.

`gcc` can also assist in this, of course, as more tests are built in as can careful use of `const` etc. in the source files.

1.3 Not quite exhaustive testing

In this category are methods like lockdep. The idea is to run a "normal" workload on the filesystem under test and then to monitor the operations performed for invalid sequences. The advantage of this method is that it will catch potential deadlocks even when the code paths didn't happen to run (in the test) at the same time, but where it is possible that they might cause a deadlock in the future.

Occasionally this will cause false positives, but those can be removed by suitable annotation or reordering of operations. One example of that is the configfs `mkdir()` issue where the locking is correct, but done in a different way to that expected by lockdep.

1.4 Feature & regression testing

As features are added to GFS2 [1, 2], new tests are added or adapted from other filesystems. In addition, Red Hat's QE department perform regular regression tests over each release of Red Hat Enterprise Linux before release.

Although this prevents previous issues from reappearing and helps to ensure that the basic functionality is working, the total state space is so much larger that it will never be able to catch all the issues.

1.5 Early exposure to users

One of the core philosophies of Open Source is that code should be released early and often. This can be a very successful strategy for projects which have a large audience. However, people who have clusters don't usually have them sitting idle just waiting for a new release of software to test on them.

As a result, the benefit of this form of testing is less than with the more common features, such as single node filesystems. There is still a substantial benefit to be gained from people building GFS2 in a variety of different configurations, but most of the bugs reported have related to build issues.

1.6 Performance testing

Once all the functionality has been tested, the next aim is to be able to narrow down any performance issues, at least to the section of code where they occur. At that point more targeted methods can then be used to identify bottlenecks.

One example of performance testing is Askant [3] (in the contrib section of the gfs2-utils git tree), which uses a combination of static analysis to annotate blktrace output with the details of the on-disk structures being written.

1.7 Testing issues

The main issue raised above is that it is tricky to test a cluster file system without a cluster, which makes testing long-winded and expensive. As a result of that, we plan to try and simulate the effect of being in a cluster as closely as possible, but only using a single node.

2 Glocks

Since glocks are the core of GFS2, they are also the initial target of the verification and testing effort. Experience has shown that many different issues in the cluster can result in a deadlock which is first encountered at the glock level.

A glock is a caching mechanism, both for locks and also for the data and metadata associated with them. Each glock can have a number of "holders" associated with it, each of which represents one lock request from the higher layers. System calls relating to GFS2 queue and dequeue holders from the glock to protect the critical section of code.

Each glock corresponds exactly to one DLM lock. The glock state machine provides a local locking mechanism which is designed to reduce the number of remote locking calls made by caching the locks in a particular state until a remote node requires the lock, or until the VM reclaims the glock from the glock LRU list via the shrinker.

The glock state machine is based on a work queue. For performance reasons, we would prefer to use tasklets, however in the current implementation we need to submit I/O from that context which prohibits their use. One of the objectives of the performance tests in this area is to try and work out just how efficient this code is, and whether any improvements can reasonably be made.

The glock debugfs interface allows the visualisation of the internal state of the glocks, the holders and it also includes some summary details of the objects being locked in some cases. Each line of the file either begins `G:` with no indentation (which refers to the glock itself) or it begins with a different letter, indented with a single space, and refers to the structures (`H:` is a holder, `I:` an inode, and `R:` a resource group) associated with the glock immediately above it in the file.

An example is shown in figure 1 which is a series of excerpts (from an approx 18M file) generated by `cat /sys/kernel/debug/gfs2/unity: myfs/glocks >my.lock` during a run of the `postmark` benchmark on a single node GFS2 filesystem. The glocks in the figure have been selected in order to show some of the more interesting features of the glock dumps.

The glock states are either `EX` (exclusive), `DF` (deferred), `SH` (shared) or `UN` (unlocked). These states correspond directly with DLM lock modes except for `UN` which may represent either the DLM null lock state, or that GFS2 doesn't hold a DLM lock. The `s:` field of the glock indicates the current state of the lock and the same field in the holder indicates the requested mode. If the lock is granted, the holder will have the `H` bit set in its flags (`f:` field) otherwise it will have the `W` wait bit set.

The full listing of all the flags for both the holder and the glock are set out in the two tables 3 and 2.

In the current upstream GFS2, once a DLM lock has been taken out it is only ever demoted to the null state (and never unlocked) unless the glock has reached the end of its life and is being freed. This means that holding a reference on a glock that has been promoted to any mode other than `NL` will also result in a reference on the associated DLM lock and thus preserve the content of the lock value block (LVB).

The content of lock value blocks is not currently available via the glock debugfs interface, although we may well add this in the future.

The `n:` field (number) indicates the number associated with each item. For glocks that is the type number followed by the glock number so that, as seen in Figure 1, the first glock is `n:5/75320`—i.e., an iopen glock which relates to inode 75320. In the case of inode and iopen glocks, the glock number is always identical to the inode's disk block number.

Type Number	Lock type	Use
1	Trans	Transaction lock
2	Inode	Inode metadata & data
3	Rgrp	Resource group metadata
4	Meta	The superblock
5	Iopen	Inode last closer detection
6	Flock	`flock(2)` syscall
8	Quota	Quota operations
9	Journal	Journal mutex

Table 1: Glock types

One of the more important glock flags, is the `l` (locked) flag. This is the bit lock which is used to arbitrate access to the glock state when a state change is to be performed. It is set when the state machine is about to send a remote lock request via the DLM, and only cleared when the complete operation has been performed. Sometimes this can mean that more than one lock request will have been sent, with various invalidations occurring between times.

When a remote callback is received from a node which wants to get a lock in a mode which conflicts with that being held on the local node, then one or other of the two flags `D` (demote) or `d` (demote pending) is set. In order to prevent starvation conditions when there is contention on a particular lock, each lock is assigned a minimum hold time. A node which has not yet had the lock for the minimum hold time is allowed to retain that lock until the time interval has expired.

If the time interval has expired, then the `D` (demote) flag will be set and the state required will be recorded. In that case the next time there are no granted locks on the holders queue, the lock will be demoted.

If the time interval has not expired, then the `d` (demote pending) flag is set instead. This also schedules the state machine to clear `d` (demote pending) and set `D` (demote) when the minimum hold time has expired.

The `I` (initial) flag is set when the glock has been assigned a DLM lock. This happens when the glock is first used and the `I` flag will then remain set until the glock is finally freed (which the DLM lock is unlocked).

The most important holder flags are `H` (holder) and `W` (wait), as mentioned earlier, since they are set on granted

```
G:  s:SH n:5/75320 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:EX n:3/258028 f:yI t:EX d:EX/0 a:3 r:4
 H: s:EX f:tH e:0 p:4466 [postmark] gfs2_inplace_reserve_i+0x177/0x780 [gfs2]
 R: n:258028 f:05 b:22256/22256 i:16800
G:  s:EX n:2/219916 f:yfI t:EX d:EX/0 a:0 r:3
 I: n:75661/219916 t:8 f:0x10 d:0x00000000 s:7522/7522
G:  s:SH n:5/127205 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:EX n:2/50382 f:yfI t:EX d:EX/0 a:0 r:2
G:  s:SH n:5/302519 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:SH n:5/313874 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:SH n:5/271916 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:SH n:5/312732 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
```

Figure 1: Example glock dump from debugsfs (edited for size)

Table 2: Glock flags

Flag	Name	Meaning
l	Locked	The glock is in the process of changing state
D	Demote	A demote request (local or remote)
d	Pending demote	A deferred (remote) demote request
p	Demote in progress	The glock is in the process of responding to a demote request
y	Dirty	Data needs flushing to disk before releasing this glock
f	Log flush	The log needs to be committed before releasing this glock
i	Invalidate in progress	In the process of invalidating pages under this glock
r	Reply pending	Reply received from remote node is awaiting processing
I	Initial	Set when DLM lock is associated with this glock
F	Frozen	Replies from remote nodes ignored - recovery is in progress

Table 3: Glock holder flags

Flag	Name	Meaning
t	Try	A “try” lock
T	Try 1CB	A “try” lock which sends a callback
e	No expire	Ignore subsequent lock cancel requests
A	Any	Any compatible lock mode is acceptable
p	Priority	Enqueue holder at the head of the queue
a	Async	Don’t wait for glock result (will poll for result later)
E	Exact	Must have exact lock mode
c	No cache	When unlocked, demote DLM lock immediately
H	Holder	Indicates that requested lock is granted
W	Wait	Set while waiting for request to complete
F	First	Set when holder is the first to be granted for this lock

lock requests and queued lock requests respectively. The ordering of the holders in the list is important; if there are any granted holders, they will always be at the head of the queue, followed by any queued holders.

If there are no granted holders, then the first holder in the list will be the one which triggers the next state change. Since demote requests are always considered higher priority than requests from the filesystem, that might not always directly result in a change to the state requested.

The glock subsystem supports two kinds of "try" locks. These are useful both because they allow the taking of locks out of the normal order (with suitable back-off and retry) and because they can be used to help avoid resources in use by other nodes. The normal `t` (try) lock is basically just that; the so called `T` (try 1CB) is identical, except that the DLM will send a single callback to current incompatible lock holders.

One uses of the `T` (try 1CB) is with the iopen locks which are used to arbitrate among the nodes when an inode's `n_link` count is zero, as to which of the nodes will be responsible for deallocating the inode. The iopen glock is normally held in the shared state, but when the `n_link` count becomes zero and `->delete_inode()` is called, it will request an exclusive lock with `T` (try 1CB) set. It will continue to deallocate the inode if the lock is granted. If the lock is not granted it will result in the node(s) which was/were preventing the grant of the lock marking their glock(s) with the `D` (demote) flag which is checked at `->drop_inode()` time in order to ensure that the deallocation is not forgotten.

This means that inodes which have zero link count, but are still open, will be deallocated by the node on which the final `close()` occurs. Also, at the same time as the inode's link count is decremented to zero, the inode is marked as being in the special state of having zero link count, but still in use in the resource group bitmap. This functions like ext3's orphan list in that it allows any subsequent reader of the bitmap to know that there is potentially space which might be reclaimed, and to have a go at reclaiming it.

2.1 Callback injection

In recent kernels, there is a sysfs interface which allows the injection of demote requests from user space. Since GFS2 has no communication with other nodes, except for the callbacks issued in response to DLM lock requests, this provides a way to simulate, on a single node the same callbacks as if it was part of a cluster.

In combination with the GFS2 trace points and blktrace, it becomes possible to check that the correct locks are being held when the block requests I/O are issued and completed. Also, the overhead of doing this is small enough that it can be left running during normal system operation without slowing it significantly (given a not too unreasonable rate of simulated callbacks).

It is generally safe to inject callbacks for inode glocks (type 2) however, for reasons which should be clear from the discussion in the previous section, injecting callbacks for iopen glocks (type 5) is likely to result in a corrupted filesystem.

Callback injection is intended for testing in carefully designed test environments, and is not something that we would encourage general use of.

3 Trace points

Linux has an event tracing subsystem which provides a high-performance method of tracking certain operations which are switchable at run time, can be filtered and provided to user space either as a series of human readable ASCII messages or via a binary output format.

The goal in placing trace points in GFS2 was to find a minimum possible number of trace points which can elicit the maximum possible amount of information about the correctness and performance of GFS2 whilst at the same time ensuring that they are generic enough that they will not need to be altered during future developments.

With that in mind, the trace points in GFS2 have been put into three major categories reflecting the main functions of the filesystem.

3.1 glock subsystem

For some time there has been a debugfs file available which gives a static view of the glock state as described above. It has been extremely useful in debugging deadlock issues, however although it shows what state the

cluster is currently in, its main failing is that it doesn't always show how it got into that state.

The trace points have also been designed with a view to being able to confirm the correctness of the cache control by combining them with the blktrace output and with knowledge of the on-disk layout. It is then possible to check that any given I/O has been issued and completed under the correct lock, and that no races are present.

The on-disk layout could be parsed in a static manner before the test starts, and then updated can be gained by using the block map trace points.

On the performance side, there are specific questions which we want to be able to answer relating to latencies of performing the various cache control operations. With the current set of trace points it is possible to measure the latencies of granting a new or cached glock, and receiving a callback and flushing the cache.

Information gathered from this will be then used to help target our efforts in improving the scalability of the filesystem.

In the initial implementation of the glock tracing patch, it was an extension to `blktrace` rather than the event tracer that it is currently. There was only one trace point at that time, which was `gfs2_glock_state_change` right in the heart of the glock layer.

Due to the way in which the glock layer is designed, it is only valid to read data or metadata relating to an inode when the glock is in the shared, deferred or exclusive states. It is only valid to write data or metadata when the glock is in the deferred or exclusive states.

Given knowledge of which disk blocks belong to which inode, a task made much easier by the `FIEMAP ioctl()` it should be possible to take the combined information from `blktrace` and `gfs2_glock_state_change` and check that this is indeed the case.

Since the scope of the tracing in GFS2 has been expanded, it is also possible to use the bmap trace points to elicit the same information. Also, due to some further recent patches, GFS2 also tags all of its metadata I/O with the metadata flag so that any violations should be easier to spot.

The `gfs2_glock_put` trace point was introduced so that tracing applications would be able to reduce the amount of state that they needed to remember, and also to allow dynamic tracking of the numbers of allocated glocks.

In order to keep track of demote requests the `gfs2_demote_rq` provides enough information to know when a request has been received and its source (local or remote). The plan is to use this to help measure performance.

Also of interest in the performance area is the time taken from a lock request being submitted to it being granted. This can be derived from the trace points `gfs2_glock_queue` and `gfs2_promote`.

3.2 Block map subsystem

Block mapping is a task central to any filesystem. GFS2 uses a traditional bitmap based system with two bits per block. The main aim of the trace points in this subsystem is to allow monitoring of the time taken to allocate and map blocks.

Additionally, with knowledge of the blocks being mapped into inodes, it is possible to keep track of the filesystem layout dynamically and extend tools to take advantage of this.

The `gfs2_bmap` trace point is called twice for each bmap operation. Once at the start to display the bmap request, and once at the end to display the result.

To keep track of allocated blocks, `gfs2_block_alloc` is called not only on allocations, but also on freeing of blocks.

3.3 Journal/log subsystem

The trace points in this subsystem track blocks being added to and removed from the journal (`gfs2_pin`), as well as the time taken to commit the transactions to the log (`gfs2_log_flush`). This can be very useful when trying to balance the memory usage of a large log against filesystem performance.

The `gfs2_log_blocks` keeps track of the reserved blocks in the log, which can help show if the log is too small for the workload, for example.

References

[1] "The GFS2 Filesystem," Steven Whitehouse, *Proceedings of the Linux Symposium*, Ottawa, Canada, June, 2007.

[2] "Global File System," Steven Whitehouse *et al.*, *Wikipedia*, `http://en.wikipedia.org/wiki/Global_File_System`

[3] "Askant: A Linux file system performance analysis tool," Andrew Price, *Final year project*, Swansea University, Summer 2008.

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*