Red Hat Enterprise Linux 8

Configuring GFS2 file systems

A guide to the configuration and management of GFS2 file systems

Last Updated: 2020-06-04

Red Hat Enterprise Linux 8 Configuring GFS2 file systems

A guide to the configuration and management of GFS2 file systems

Legal Notice

Abstract

This guide provides information about configuring and managing GFS2 file systems for Red Hat Enterprise Linux 8.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your input on our documentation. Please let us know how we could make it better. To do so:

- For simple comments on specific passages:
  1. Make sure you are viewing the documentation in the *Multi-page HTML* format. In addition, ensure you see the **Feedback** button in the upper right corner of the document.
  2. Use your mouse cursor to highlight the part of text that you want to comment on.
  3. Click the **Add Feedback** pop-up that appears below the highlighted text.
  4. Follow the displayed instructions.
- For submitting more complex feedback, create a Bugzilla ticket:
  1. Go to the Bugzilla website.
  2. As the Component, use **Documentation**.
  3. Fill in the **Description** field with your suggestion for improvement. Include a link to the relevant part(s) of documentation.
  4. Click **Submit Bug**.

CHAPTER 1. PLANNING A GFS2 FILE SYSTEM DEPLOYMENT

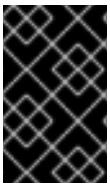
The Red Hat Global File System 2 (GFS2) file system is a 64-bit symmetric cluster file system which provides a shared name space and manages coherency between multiple nodes sharing a common block device. A GFS2 file system is intended to provide a feature set which is as close as possible to a local file system, while at the same time enforcing full cluster coherency between nodes. To achieve this, the nodes employ a cluster-wide locking scheme for file system resources. This locking scheme uses communication protocols such as TCP/IP to exchange locking information.

In a few cases, the Linux file system API does not allow the clustered nature of GFS2 to be totally transparent; for example, programs using POSIX locks in GFS2 should avoid using the **GETLK** function since, in a clustered environment, the process ID may be for a different node in the cluster. In most cases however, the functionality of a GFS2 file system is identical to that of a local file system.

The Red Hat Enterprise Linux (RHEL) Resilient Storage Add-On provides GFS2, and it depends on the RHEL High Availability Add-On to provide the cluster management required by GFS2.

The **gfs2.ko** kernel module implements the GFS2 file system and is loaded on GFS2 cluster nodes.

To get the best performance from GFS2, it is important to take into account the performance considerations which stem from the underlying design. Just like a local file system, GFS2 relies on the page cache in order to improve performance by local caching of frequently used data. In order to maintain coherency across the nodes in the cluster, cache control is provided by the *glock* state machine.



IMPORTANT

Make sure that your deployment of the Red Hat High Availability Add-On meets your needs and can be supported. Consult with an authorized Red Hat representative to verify your configuration prior to deployment.

1.1. KEY GFS2 PARAMETERS TO DETERMINE

Before you install and set up GFS2, note the following key characteristics of your GFS2 file systems:

GFS2 nodes

Determine which nodes in the cluster will mount the GFS2 file systems.

Number of file systems

Determine how many GFS2 file systems to create initially. More file systems can be added later.

File system name

Each GFS2 file system should have a unique name. This name is usually the same as the LVM logical volume name and is used as the DLM lock table name when a GFS2 file system is mounted. For example, this guide uses file system names **mydata1** and **mydata2** in some example procedures.

Journals

Determine the number of journals for your GFS2 file systems. GFS2 requires one journal for each node in the cluster that needs to mount the file system. For example, if you have a 16-node cluster but need to mount only the file system from two nodes, you need only two journals. GFS2 allows you to add journals dynamically at a later point with the **gfs2_jadd** utility as additional servers mount a file system.

Storage devices and partitions

Determine the storage devices and partitions to be used for creating logical volumes (using **lvmlockd**) in the file systems.

Time protocol

Make sure that the clocks on the GFS2 nodes are synchronized. It is recommended that you use the Precision Time Protocol (PTP) or, if necessary for your configuration, the Network Time Protocol (NTP) software provided with your Red Hat Enterprise Linux distribution.
The system clocks in GFS2 nodes must be within a few minutes of each other to prevent unnecessary inode time stamp updating. Unnecessary inode time stamp updating severely impacts cluster performance.

NOTE

You may see performance problems with GFS2 when many create and delete operations are issued from more than one node in the same directory at the same time. If this causes performance problems in your system, you should localize file creation and deletions by a node to directories specific to that node as much as possible.

1.2. GFS2 SUPPORT CONSIDERATIONS

Table 1.1, "GFS2 Support Limits" summarizes the current maximum file system size and number of nodes that GFS2 supports.

Table 1.1. GFS2 Support Limits

Parameter	Maximum
Number of nodes	16 (x86, Power8 on PowerVM) 4 (s390x under z/VM)
File system size	100TB on all supported architectures

GFS2 is based on a 64-bit architecture, which can theoretically accommodate an 8 EB file system. If your system requires larger GFS2 file systems than are currently supported, contact your Red Hat service representative.

NOTE

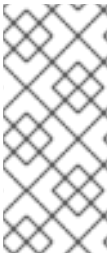
Although a GFS2 file system can be implemented in a standalone system or as part of a cluster configuration, Red Hat does not support the use of GFS2 as a single-node file system. Red Hat does support a number of high-performance single node file systems which are optimized for single node and thus have generally lower overhead than a cluster file system. Red Hat recommends using these file systems in preference to GFS2 in cases where only a single node needs to mount the file system. For information on the file systems that Red Hat Enterprise Linux 8 supports, see Managing file systems.

Red Hat will continue to support single-node GFS2 file systems for mounting snapshots of cluster file systems as might be needed, for example, for backup purposes.

When determining the size of your file system, you should consider your recovery needs. Running the **fsck.gfs2** command on a very large file system can take a long time and consume a large amount of memory. Additionally, in the event of a disk or disk subsystem failure, recovery time is limited by the

speed of your backup media. For information on the amount of memory the **fsck.gfs2** command requires, see Determing required memory for running fsck.gfs2.

While a GFS2 file system may be used outside of LVM, Red Hat supports only GFS2 file systems that are created on a shared LVM logical volume.

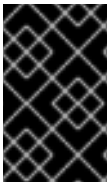


NOTE

When you configure a GFS2 file system as a cluster file system, you must ensure that all nodes in the cluster have access to the shared storage. Asymmetric cluster configurations in which some nodes have access to the shared storage and others do not are not supported. This does not require that all nodes actually mount the GFS2 file system itself.

1.3. GFS2 FORMATTING CONSIDERATIONS

This section provides recommendations for how to format your GFS2 file system to optimize performance.



IMPORTANT

Make sure that your deployment of the Red Hat High Availability Add-On meets your needs and can be supported. Consult with an authorized Red Hat representative to verify your configuration prior to deployment.

File System Size: Smaller Is Better

GFS2 is based on a 64-bit architecture, which can theoretically accommodate an 8 EB file system. However, the current supported maximum size of a GFS2 file system for 64-bit hardware is 100TB.

Note that even though GFS2 large file systems are possible, that does not mean they are recommended. The rule of thumb with GFS2 is that smaller is better: it is better to have 10 1TB file systems than one 10TB file system.

There are several reasons why you should keep your GFS2 file systems small:

- Less time is required to back up each file system.
- Less time is required if you need to check the file system with the **fsck.gfs2** command.
- Less memory is required if you need to check the file system with the **fsck.gfs2** command.

In addition, fewer resource groups to maintain mean better performance.

Of course, if you make your GFS2 file system too small, you might run out of space, and that has its own consequences. You should consider your own use cases before deciding on a size.

Block Size: Default (4K) Blocks Are Preferred

The **mkfs.gfs2** command attempts to estimate an optimal block size based on device topology. In general, 4K blocks are the preferred block size because 4K is the default page size (memory) for Red Hat Enterprise Linux. Unlike some other file systems, GFS2 does most of its operations using 4K kernel buffers. If your block size is 4K, the kernel has to do less work to manipulate the buffers.

It is recommended that you use the default block size, which should yield the highest performance. You may need to use a different block size only if you require efficient storage of many very small files.

Journal Size: Default (128MB) Is Usually Optimal

When you run the **mkfs.gfs2** command to create a GFS2 file system, you may specify the size of the journals. If you do not specify a size, it will default to 128MB, which should be optimal for most applications.

Some system administrators might think that 128MB is excessive and be tempted to reduce the size of the journal to the minimum of 8MB or a more conservative 32MB. While that might work, it can severely impact performance. Like many journaling file systems, every time GFS2 writes metadata, the metadata is committed to the journal before it is put into place. This ensures that if the system crashes or loses power, you will recover all of the metadata when the journal is automatically replayed at mount time. However, it does not take much file system activity to fill an 8MB journal, and when the journal is full, performance slows because GFS2 has to wait for writes to the storage.

It is generally recommended to use the default journal size of 128MB. If your file system is very small (for example, 5GB), having a 128MB journal might be impractical. If you have a larger file system and can afford the space, using 256MB journals might improve performance.

Size and Number of Resource Groups

When a GFS2 file system is created with the **mkfs.gfs2** command, it divides the storage into uniform slices known as resource groups. It attempts to estimate an optimal resource group size (ranging from 32MB to 2GB). You can override the default with the **-r** option of the **mkfs.gfs2** command.

Your optimal resource group size depends on how you will use the file system. Consider how full it will be and whether or not it will be severely fragmented.

You should experiment with different resource group sizes to see which results in optimal performance. It is a best practice to experiment with a test cluster before deploying GFS2 into full production.

If your file system has too many resource groups, each of which is too small, block allocations can waste too much time searching tens of thousands of resource groups for a free block. The more full your file system, the more resource groups that will be searched, and every one of them requires a cluster-wide lock. This leads to slow performance.

If, however, your file system has too few resource groups, each of which is too big, block allocations might contend more often for the same resource group lock, which also impacts performance. For example, if you have a 10GB file system that is carved up into five resource groups of 2GB, the nodes in your cluster will fight over those five resource groups more often than if the same file system were carved into 320 resource groups of 32MB. The problem is exacerbated if your file system is nearly full because every block allocation might have to look through several resource groups before it finds one with a free block. GFS2 tries to mitigate this problem in two ways:

- First, when a resource group is completely full, it remembers that and tries to avoid checking it for future allocations until a block is freed from it. If you never delete files, contention will be less severe. However, if your application is constantly deleting blocks and allocating new blocks on a file system that is mostly full, contention will be very high and this will severely impact performance.

- Second, when new blocks are added to an existing file (for example, by appending) GFS2 will attempt to group the new blocks together in the same resource group as the file. This is done to increase performance: on a spinning disk, seek operations take less time when they are physically close together.

The worst case scenario is when there is a central directory in which all the nodes create files because all of the nodes will constantly fight to lock the same resource group.

1.4. CLUSTER CONSIDERATIONS

When determining the number of nodes that your system will contain, note that there is a trade-off between high availability and performance. With a larger number of nodes, it becomes increasingly difficult to make workloads scale. For that reason, Red Hat does not support using GFS2 for cluster file system deployments greater than 16 nodes.

Deploying a cluster file system is not a "drop in" replacement for a single node deployment. Red Hat recommends that you allow a period of around 8-12 weeks of testing on new installations in order to test the system and ensure that it is working at the required performance level. During this period, any performance or functional issues can be worked out and any queries should be directed to the Red Hat support team.

Red Hat recommends that customers considering deploying clusters have their configurations reviewed by Red Hat support before deployment to avoid any possible support issues later on.

1.5. HARDWARE CONSIDERATIONS

You should take the following hardware considerations into account when deploying a GFS2 file system.

- Use higher quality storage options
  GFS2 can operate on cheaper shared storage options, such as iSCSI or Fibre Channel over Ethernet (FCoE), but you will get better performance if you buy higher quality storage with larger caching capacity. Red Hat performs most quality, sanity, and performance tests on SAN storage with Fibre Channel interconnect. As a general rule, it is always better to deploy something that has been tested first.

- Test network equipment before deploying
  Higher quality, faster network equipment makes cluster communications and GFS2 run faster with better reliability. However, you do not have to purchase the most expensive hardware. Some of the most expensive network switches have problems passing multicast packets, which are used for passing **fcntl** locks (flocks), whereas cheaper commodity network switches are sometimes faster and more reliable. Red Hat recommends trying equipment before deploying it into full production.

CHAPTER 2. RECOMMENDATIONS FOR GFS2 USAGE

This section provides general recommendations about GFS2 usage.

2.1. CONFIGURING ATIME UPDATES

Each file inode and directory inode has three time stamps associated with it:

- **ctime** – The last time the inode status was changed
- **mtime** – The last time the file (or directory) data was modified
- **atime** – The last time the file (or directory) data was accessed

If **atime** updates are enabled as they are by default on GFS2 and other Linux file systems, then every time a file is read its inode needs to be updated.

Because few applications use the information provided by **atime**, those updates can require a significant amount of unnecessary write traffic and file locking traffic. That traffic can degrade performance; therefore, it may be preferable to turn off or reduce the frequency of **atime** updates.

The following methods of reducing the effects of **atime** updating are available:

- Mount with **relatime** (relative atime), which updates the **atime** if the previous **atime** update is older than the **mtime** or **ctime** update. This is the default mount option for GFS2 file systems.
- Mount with **noatime** or **nodiratime**. Mounting with **noatime** disables **atime** updates for both files and directories on that file system, while mounting with **nodiratime** disables **atime** updates only for directories on that file system, It is generally recommended that you mount GFS2 file systems with the **noatime** or **nodiratime** mount option whenever possible, with the preference for **noatime** where the application allows for this. For more information on the effect of these arguments on GFS2 file system performance, see GFS2 Node Locking.

Use the following command to mount a GFS2 file system with the **noatime** Linux mount option.

```
mount BlockDevice MountPoint -o noatime
```

BlockDevice

Specifies the block device where the GFS2 file system resides.

MountPoint

Specifies the directory where the GFS2 file system should be mounted.

In this example, the GFS2 file system resides on **/dev/vg01/lvol0** and is mounted on directory **/mygfs2** with **atime** updates turned off.

```
# mount /dev/vg01/lvol0 /mygfs2 -o noatime
```

2.2. VFS TUNING OPTIONS: RESEARCH AND EXPERIMENT

Like all Linux file systems, GFS2 sits on top of a layer called the virtual file system (VFS). The VFS provides good defaults for the cache settings for most workloads and should not need changing in most cases. If, however, you have a workload that is not running efficiently (for example, cache is too large or

too small) then you may be able to improve the performance by using the **sysctl**(8) command to adjust the values of the **sysctl** files in the **/proc/sys/vm** directory. Documentation for these files can be found in the kernel source tree **Documentation/sysctl/vm.txt**.

For example, the values for **dirty_background_ratio** and **vfs_cache_pressure** may be adjusted depending on your situation. To fetch the current values, use the following commands:

```
# sysctl -n vm.dirty_background_ratio
# sysctl -n vm.vfs_cache_pressure
```

The following commands adjust the values:

```
# sysctl -w vm.dirty_background_ratio=20
# sysctl -w vm.vfs_cache_pressure=500
```

You can permanently change the values of these parameters by editing the **/etc/sysctl.conf** file.

To find the optimal values for your use cases, research the various VFS options and experiment on a test cluster before deploying into full production.

2.3. SELINUX ON GFS2

Use of Security Enhanced Linux (SELinux) with GFS2 incurs a small performance penalty. To avoid this overhead, you may choose not to use SELinux with GFS2 even on a system with SELinux in enforcing mode. When mounting a GFS2 file system, you can ensure that SELinux will not attempt to read the **seclabel** element on each file system object by using one of the **context** options as described on the **mount**(8) man page; SELinux will assume that all content in the file system is labeled with the **seclabel** element provided in the **context** mount options. This will also speed up processing as it avoids another disk read of the extended attribute block that could contain **seclabel** elements.

For example, on a system with SELinux in enforcing mode, you can use the following **mount** command to mount the GFS2 file system if the file system is going to contain Apache content. This label will apply to the entire file system; it remains in memory and is not written to disk.

```
# mount -t gfs2 -o context=system_u:object_r:httpd_sys_content_t:s0
/dev/mapper/xyz/mnt/gfs2
```

If you are not sure whether the file system will contain Apache content, you can use the labels **public_content_rw_t** or **public_content_t**, or you could define a new label altogether and define a policy around it.

Note that in a Pacemaker cluster you should always use Pacemaker to manage a GFS2 file system. You can specify the mount options when you create a GFS2 file system resource.

2.4. SETTING UP NFS OVER GFS2

Due to the added complexity of the GFS2 locking subsystem and its clustered nature, setting up NFS over GFS2 requires taking many precautions and careful configuration. This section describes the caveats you should take into account when configuring an NFS service over a GFS2 file system.

WARNING

If the GFS2 file system is NFS exported, then you must mount the file system with the **localflocks** option. Because utilizing the **localflocks** option prevents you from safely accessing the GFS2 filesystem from multiple locations, and it is not viable to export GFS2 from multiple nodes simultaneously, it is a support requirement that the GFS2 file system be mounted on only one node at a time when using this configuration. The intended effect of this is to force POSIX locks from each server to be local: non-clustered, independent of each other. This is because a number of problems exist if GFS2 attempts to implement POSIX locks from NFS across the nodes of a cluster. For applications running on NFS clients, localized POSIX locks means that two clients can hold the same lock concurrently if the two clients are mounting from different servers, which could cause data corruption. If all clients mount NFS from one server, then the problem of separate servers granting the same locks independently goes away. If you are not sure whether to mount your file system with the **localflocks** option, you should not use the option. Contact Red Hat support immediately to discuss the appropriate configuration to avoid data loss. Exporting GFS2 via NFS, while technically supported in some circumstances, is not recommended.

For all other (non-NFS) GFS2 applications, do not mount your file system using **localflocks**, so that GFS2 will manage the POSIX locks and flocks between all the nodes in the cluster (on a cluster-wide basis). If you specify **localflocks** and do not use NFS, the other nodes in the cluster will not have knowledge of each other's POSIX locks and flocks, thus making them unsafe in a clustered environment

In addition to the locking considerations, you should take the following into account when configuring an NFS service over a GFS2 file system.

- Red Hat supports only Red Hat High Availability Add-On configurations using NFSv3 with locking in an active/passive configuration with the following characteristics. This configuration provides High Availability (HA) for the file system and reduces system downtime since a failed node does not result in the requirement to execute the **fsck** command when failing the NFS server from one node to another.
    - The back-end file system is a GFS2 file system running on a 2 to 16 node cluster.
    - An NFSv3 server is defined as a service exporting the entire GFS2 file system from a single cluster node at a time.
    - The NFS server can fail over from one cluster node to another (active/passive configuration).
    - No access to the GFS2 file system is allowed *except* through the NFS server. This includes both local GFS2 file system access as well as access through Samba or Clustered Samba. Accessing the file system locally via the cluster node from which it is mounted may result in data corruption.
    - There is no NFS quota support on the system.
- The **fsid=** NFS option is mandatory for NFS exports of GFS2.

- If problems arise with your cluster (for example, the cluster becomes inquorate and fencing is not successful), the clustered logical volumes and the GFS2 file system will be frozen and no access is possible until the cluster is quorate. You should consider this possibility when determining whether a simple failover solution such as the one defined in this procedure is the most appropriate for your system.

2.5. SAMBA (SMB OR WINDOWS) FILE SERVING OVER GFS2

You can use Samba (SMB or Windows) file serving from a GFS2 file system with CTDB, which allows active/active configurations.

Simultaneous access to the data in the Samba share from outside of Samba is not supported. There is currently no support for GFS2 cluster leases, which slows Samba file serving. For further information on support policies for Samba, see Support Policies for RHEL Resilient Storage - ctdb General Policies and Support Policies for RHEL Resilient Storage - Exporting gfs2 contents via other protocols on the Red Hat Customer Portal.

2.6. CONFIGURING VIRTUAL MACHINES FOR GFS2

When using a GFS2 file system with a virtual machine, it is important that your VM storage settings on each node be configured properly in order to force the cache off. For example, including these settings for **cache** and **io** in the **libvirt** domain should allow GFS2 to behave as expected.

```
<driver name='qemu' type='raw' cache='none' io='native'/>
```

Alternately, you can configure the **shareable** attribute within the device element. This indicates that the device is expected to be shared between domains (as long as hypervisor and OS support this). If **shareable** is used, **cache='no'** should be used for that device.

2.7. BLOCK ALLOCATION

This section provides a summary of issues related to block allocation in GFS2 file systems. Even though applications that only write data typically do not care how or where a block is allocated, some knowledge of how block allocation works can help you optimize performance.

2.7.1. Leave free space in the file system

When a GFS2 file system is nearly full, the block allocator starts to have a difficult time finding space for new blocks to be allocated. As a result, blocks given out by the allocator tend to be squeezed into the end of a resource group or in tiny slices where file fragmentation is much more likely. This file fragmentation can cause performance problems. In addition, when a GFS2 file system is nearly full, the GFS2 block allocator spends more time searching through multiple resource groups, and that adds lock contention that would not necessarily be there on a file system that has ample free space. This also can cause performance problems.

For these reasons, it is recommended that you not run a file system that is more than 85 percent full, although this figure may vary depending on workload.

2.7.2. Have each node allocate its own files, if possible

When developing applications for use with GFS2 file systems, it is recommended that you have each node allocate it own files, if possible. Due to the way the distributed lock manager (DLM) works, there will be more lock contention if all files are allocated by one node and other nodes need to add blocks to

those files.

With DLM, the first node to lock a resource (like a file) becomes the “lock master” for that lock. Other nodes may lock that resource, but they have to ask permission from the lock master first. Each node knows which locks for which it is the lock master, and each node knows which node it has lent a lock to. Locking a lock on the master node is much faster than locking one on another node that has to stop and ask permission from the lock’s master.

As in many file systems, the GFS2 allocator tries to keep blocks in the same file close to one another to reduce the movement of disk heads and boost performance. A node that allocates blocks to a file will likely need to use and lock the same resource groups for the new blocks (unless all the blocks in that resource group are in use). The file system will run faster if the lock master for the resource group containing the file allocates its data blocks (it is faster to have the node that first opened the file do all the writing of new blocks).

2.7.3. Preallocate, if possible

If files are preallocated, block allocations can be avoided altogether and the file system can run more efficiently. GFS2 includes the **fallocate(1)** system call, which you can use to preallocate blocks of data.

CHAPTER 3. GFS2 FILE SYSTEMS

This section provides information on the commands and options you use to create, mount, and grow GFS2 file systems.

3.1. GFS2 FILE SYSTEM CREATION

You create a GFS2 file system with the **mkfs.gfs2** command. A file system is created on an activated LVM volume.

3.1.1. The GFS2 mkfs command

The following information is required to run the **mkfs.gfs2** command to create a clustered GFS2 file system:

- Lock protocol/module name, which is **lock_dlm** for a cluster
- Cluster name
- Number of journals (one journal required for each node that may be mounting the file system)

NOTE

Once you have created a GFS2 file system with the **mkfs.gfs2** command, you cannot decrease the size of the file system. You can, however, increase the size of an existing file system with the **gfs2_grow** command.

The format for creating a clustered GFS2 file system is as follows. Note that Red Hat does not support the use of GFS2 as a single-node file system.

```
mkfs.gfs2 -p lock_dlm -t ClusterName:FSName -j NumberJournals BlockDevice
```

If you prefer, you can create a GFS2 file system by using the **mkfs** command with the **-t** parameter specifying a file system of type **gfs2**, followed by the GFS2 file system options.

```
mkfs -t gfs2 -p lock_dlm -t ClusterName:FSName -j NumberJournals BlockDevice
```

WARNING

Improperly specifying the *ClusterName:FSName* parameter may cause file system or lock space corruption.

ClusterName

The name of the cluster for which the GFS2 file system is being created.

FSName

The file system name, which can be 1 to 16 characters long. The name must be unique for all **lock_dlm** file systems over the cluster.

NumberJournals

Specifies the number of journals to be created by the **mkfs.gfs2** command. One journal is required for each node that mounts the file system. For GFS2 file systems, more journals can be added later without growing the file system.

BlockDevice

Specifies a logical or other block device

Table 3.1, "Command Options: **mkfs.gfs2**" describes the **mkfs.gfs2** command options (flags and parameters).

Table 3.1. Command Options: mkfs.gfs2

Flag	Parameter	Description
-c	**Megabytes**	Sets the initial size of each journal's quota change file to **Megabytes**.
-D		Enables debugging output.
-h		Help. Displays available options.
-J	**Megabytes**	Specifies the size of the journal in megabytes. Default journal size is 128 megabytes. The minimum size is 8 megabytes. Larger journals improve performance, although they use more memory than smaller journals.
-j	**Number**	Specifies the number of journals to be created by the **mkfs.gfs2** command. One journal is required for each node that mounts the file system. If this option is not specified, one journal will be created. For GFS2 file systems, you can add additional journals at a later time without growing the file system.
-O		Prevents the **mkfs.gfs2** command from asking for confirmation before writing the file system.

Flag	Parameter	Description
-p	**LockProtoName**	* Specifies the name of the locking protocol to use. Recognized locking protocols include: * **lock_dlm** – The standard locking module, required for a clustered file system. * **lock_nolock** – Used when GFS2 is acting as a local file system (one node only).
-q		Quiet. Do not display anything.
-r	**Megabytes**	Specifies the size of the resource groups in megabytes. The minimum resource group size is 32 megabytes. The maximum resource group size is 2048 megabytes. A large resource group size may increase performance on very large file systems. If this is not specified, **mkfs.gfs2** chooses the resource group size based on the size of the file system: average size file systems will have 256 megabyte resource groups, and bigger file systems will have bigger RGs for better performance.

Flag	Parameter	Description
-t	**LockTableName**	* A unique identifier that specifies the lock table field when you use the **lock_dlm** protocol; the **lock_nolock** protocol does not use this parameter. * This parameter has two parts separated by a colon (no spaces) as follows: **ClusterName:FSName**. * **ClusterName** is the name of the cluster for which the GFS2 file system is being created; only members of this cluster are permitted to use this file system. * **FSName**, the file system name, can be 1 to 16 characters in length, and the name must be unique among all file systems in the cluster.
-V		Displays command version information.

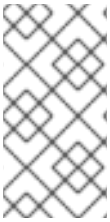
3.1.2. Creating a GFS2 file system

The following example creates two GFS2 file systems. For both of these file systems, lock_dlm` is the locking protocol that the file system uses, since this is a clustered file system. Both file systems can be used in the cluster named **alpha**.

For the first file system, file system name is **mydata1**. it contains eight journals and is created on **/dev/vg01/lvol0**. For the second file system, the file system name is **mydata2**. It contains eight journals and is created on **/dev/vg01/lvol1**.

```
# mkfs.gfs2 -p lock_dlm -t alpha:mydata1 -j 8 /dev/vg01/lvol0
# mkfs.gfs2 -p lock_dlm -t alpha:mydata2 -j 8 /dev/vg01/lvol1
```

3.2. MOUNTING A GFS2 FILE SYSTEM



NOTE

You should always use Pacemaker to manage the GFS2 file system in a production environment rather than manually mounting the file system with a **mount** command, as this may cause issues at system shutdown as described in Unmounting a GFS2 file system.

Before you can mount a GFS2 file system, the file system must exist, the volume where the file system exists must be activated, and the supporting clustering and locking systems must be started. After those requirements have been met, you can mount the GFS2 file system as you would any Linux file system.

To manipulate file ACLs, you must mount the file system with the **-o acl** mount option. If a file system is mounted without the **-o acl** mount option, users are allowed to view ACLs (with **getfacl**), but are not allowed to set them (with **setfacl**).

3.2.1. Mounting a GFS2 file system with no options specified

In this example, the GFS2 file system on **/dev/vg01/lvol0** is mounted on the **/mygfs2** directory.

```
# mount /dev/vg01/lvol0 /mygfs2
```

3.2.2. Mounting a GFS2 file system that specifies mount options

The following is the format for the command to mount a GFS2 file system that specifies mount options.

```
mount BlockDevice MountPoint -o option
```

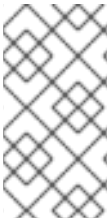
BlockDevice

Specifies the block device where the GFS2 file system resides.

MountPoint

Specifies the directory where the GFS2 file system should be mounted.

The **-o option** argument consists of GFS2-specific options (see Table 3.2, "GFS2-Specific Mount Options") or acceptable standard Linux **mount -o** options, or a combination of both. Multiple **option** parameters are separated by a comma and no spaces.



NOTE

The **mount** command is a Linux system command. In addition to using GFS2-specific options described in this section, you can use other, standard, **mount** command options (for example, **-r**). For information about other Linux **mount** command options, see the Linux **mount** man page.

Table 3.2, "GFS2-Specific Mount Options" describes the available GFS2-specific **-o option** values that can be passed to GFS2 at mount time.

NOTE

This table includes descriptions of options that are used with local file systems only. Note, however, that Red Hat does not support the use of GFS2 as a single-node file system. Red Hat will continue to support single-node GFS2 file systems for mounting snapshots of cluster file systems (for example, for backup purposes).

Table 3.2. GFS2-Specific Mount Options

Option	Description

Option	Description
acl	Allows manipulating file ACLs. If a file system is mounted without the **acl** mount option, users are allowed to view ACLs (with **getfacl**), but are not allowed to set them (with **setfacl**).
data=[ordered|writeback]	When **data=ordered** is set, the user data modified by a transaction is flushed to the disk before the transaction is committed to disk. This should prevent the user from seeing uninitialized blocks in a file after a crash. When **data=writeback** mode is set, the user data is written to the disk at any time after it is dirtied; this does not provide the same consistency guarantee as **ordered** mode, but it should be slightly faster for some workloads. The default value is **ordered** mode.
* **ignore_local_fs** * **Caution:** This option should*not* be used when GFS2 file systems are shared.	Forces GFS2 to treat the file system as a multi-host file system. By default, using **lock_nolock** automatically turns on the **localflocks** flag.
* **localflocks** * **Caution:** This option should not be used when GFS2 file systems are shared.	Tells GFS2 to let the VFS (virtual file system) layer do all flock and fcntl. The **localflocks** flag is automatically turned on by **lock_nolock**.
lockproto=LockModuleName	Allows the user to specify which locking protocol to use with the file system. If **LockModuleName** is not specified, the locking protocol name is read from the file system superblock.
locktable=LockTableName	Allows the user to specify which locking table to use with the file system.
quota=[off/account/on]	Turns quotas on or off for a file system. Setting the quotas to be in the **account** state causes the per UID/GID usage statistics to be correctly maintained by the file system; limit and warn values are ignored. The default value is **off**.
errors=panic|withdraw	When **errors=panic** is specified, file system errors will cause a kernel panic. When **errors=withdraw** is specified, which is the default behavior, file system errors will cause the system to withdraw from the file system and make it inaccessible until the next reboot; in some cases the system may remain running.

Option	Description
discard/nodiscard	Causes GFS2 to generate "discard" I/O requests for blocks that have been freed. These can be used by suitable hardware to implement thin provisioning and similar schemes.
barrier/nobarrier	Causes GFS2 to send I/O barriers when flushing the journal. The default value is **on**. This option is automatically turned **off** if the underlying device does not support I/O barriers. Use of I/O barriers with GFS2 is highly recommended at all times unless the block device is designed so that it cannot lose its write cache content (for example, if it is on a UPS or it does not have a write cache).
quota_quantum=*secs*	Sets the number of seconds for which a change in the quota information may sit on one node before being written to the quota file. This is the preferred way to set this parameter. The value is an integer number of seconds greater than zero. The default is 60 seconds. Shorter settings result in faster updates of the lazy quota information and less likelihood of someone exceeding their quota. Longer settings make file system operations involving quotas faster and more efficient.
statfs_quantum=*secs*	Setting **statfs_quantum** to 0 is the preferred way to set the slow version of **statfs**. The default value is 30 secs which sets the maximum time period before **statfs** changes will be synced to the master**statfs** file. This can be adjusted to allow for faster, less accurate **statfs** values or slower more accurate values. When this option is set to 0, **statfs** will always report the true values.
statfs_percent=*value*	Provides a bound on the maximum percentage change in the **statfs** information on a local basis before it is synced back to the master **statfs** file, even if the time period has not expired. If the setting of **statfs_quantum** is 0, then this setting is ignored.

3.2.3. Unmounting a GFS2 file system

GFS2 file systems that have been mounted manually rather than automatically through Pacemaker will not be known to the system when file systems are unmounted at system shutdown. As a result, the GFS2 resource agent will not unmount the GFS2 file system. After the GFS2 resource agent is shut down, the standard shutdown process kills off all remaining user processes, including the cluster infrastructure, and tries to unmount the file system. This unmount will fail without the cluster infrastructure and the system will hang.

To prevent the system from hanging when the GFS2 file systems are unmounted, you should do one of the following:

- Always use Pacemaker to manage the GFS2 file system.
- If a GFS2 file system has been mounted manually with the **mount** command, be sure to unmount the file system manually with the **umount** command before rebooting or shutting down the system.

If your file system hangs while it is being unmounted during system shutdown under these circumstances, perform a hardware reboot. It is unlikely that any data will be lost since the file system is synced earlier in the shutdown process.

The GFS2 file system can be unmounted the same way as any Linux file system, by using the **umount** command.

NOTE

The **umount** command is a Linux system command. Information about this command can be found in the Linux **umount** command man pages.

Usage

```
umount MountPoint
```

MountPoint

Specifies the directory where the GFS2 file system is currently mounted.

3.3. BACKING UP A GFS2 FILE SYSTEM

It is important to make regular backups of your GFS2 file system in case of emergency, regardless of the size of your file system. Many system administrators feel safe because they are protected by RAID, multipath, mirroring, snapshots, and other forms of redundancy, but there is no such thing as safe enough.

It can be a problem to create a backup since the process of backing up a node or set of nodes usually involves reading the entire file system in sequence. If this is done from a single node, that node will retain all the information in cache until other nodes in the cluster start requesting locks. Running this type of backup program while the cluster is in operation will negatively impact performance.

Dropping the caches once the backup is complete reduces the time required by other nodes to regain ownership of their cluster locks/caches. This is still not ideal, however, because the other nodes will have stopped caching the data that they were caching before the backup process began. You can drop caches using the following command after the backup is complete:

```
echo -n 3 > /proc/sys/vm/drop_caches
```

It is faster if each node in the cluster backs up its own files so that the task is split between the nodes. You might be able to accomplish this with a script that uses the **rsync** command on node-specific directories.

Red Hat recommends making a GFS2 backup by creating a hardware snapshot on the SAN, presenting the snapshot to another system, and backing it up there. The backup system should mount the snapshot with **-o lockproto=lock_nolock** since it will not be in a cluster.

3.4. SUSPENDING ACTIVITY ON A GFS2 FILE SYSTEM

You can suspend write activity to a file system by using the **dmsetup suspend** command. Suspending write activity allows hardware-based device snapshots to be used to capture the file system in a consistent state. The **dmsetup resume** command ends the suspension.

The format for the command to suspend activity on a GFS2 file system is as follows.

```
dmsetup suspend MountPoint
```

This example suspends writes to file system **/mygfs2**.

```
# dmsetup suspend /mygfs2
```

The format for the command to end suspension of activity on a GFS2 file system is as follows.

```
dmsetup resume MountPoint
```

This example ends suspension of writes to file system **/mygfs2**.

```
# dmsetup resume /mygfs2
```

3.5. GROWING A GFS2 FILE SYSTEM

The **gfs2_grow** command is used to expand a GFS2 file system after the device where the file system resides has been expanded. Running the **gfs2_grow** command on an existing GFS2 file system fills all spare space between the current end of the file system and the end of the device with a newly initialized GFS2 file system extension. All nodes in the cluster can then use the extra storage space that has been added.

NOTE

You cannot decrease the size of a GFS2 file system.

The **gfs2_grow** command must be run on a mounted file system. The following procedure increases the size of the GFS2 file system in a cluster that is mounted on the logical volume **shared_vg/shared_lv1** with a mount point of **/mnt/gfs2**.

1. Perform a backup of the data on the file system.

2. If you do not know the logical volume that is used by the file system to be expanded, you can determine this by running the **df *mountpoint*** command. This will display the device name in the following format:
   /dev/mapper/*vg-lv*

   For example, the device name **/dev/mapper/shared_vg-shared_lv1** indicates that the logical volume is **shared_vg/shared_lv1**.

3. On one node of the cluster, expand the underlying cluster volume with the **lvextend** command, using the **--lockopt skiplv** option to override normal logical volume locking.

   ```
   # lvextend --lockopt skiplv -L+1G shared_vg/shared_lv1
   WARNING: skipping LV lock in lvmlockd.
   Size of logical volume shared_vg/shared_lv1 changed from 5.00 GiB (1280 extents) to 6.00
   ```

```
  GiB (1536 extents).
  WARNING: extending LV with a shared lock, other hosts may require LV refresh.
  Logical volume shared_vg/shared_lv1 successfully resized.
```

4. If you are running RHEL 8.0, on every additional node of the cluster refresh the logical volume to update the active logical volume on that node. This step is not necessary on systems running RHEL 8.1 and later as the step is automated when the logical volume is extended.

```
# lvchange --refresh shared_vg/shared_lv1
```

5. One one node of the cluster, increase the size of the GFS2 file system. Do not extend the file system if the logical volume was not refreshed on all of the nodes, otherwise the file system data may become unavailable throughout the cluster.

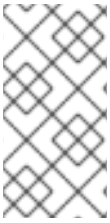
```
# gfs2_grow /mnt/gfs2
FS: Mount point:             /mnt/gfs2
FS: Device:                  /dev/mapper/shared_vg-shared_lv1
FS: Size:                    1310719 (0x13ffff)
DEV: Length:                 1572864 (0x180000)
The file system will grow by 1024MB.
gfs2_grow complete.
```

6. Run the **df** command on all nodes to check that the new space is now available in the file system. Note that it may take up to 30 seconds for the the **df** command on all nodes to show the same file system size

```
# df -h /mnt/gfs2
Filesystem                      Size  Used Avail Use% Mounted on
/dev/mapper/shared_vg-shared_lv1  6.0G  4.5G  1.6G  75% /mnt/gfs2
```

3.6. ADDING JOURNALS TO A GFS2 FILE SYSTEM

GFS2 requires one journal for each node in a cluster that needs to mount the file system. If you add additional nodes to the cluster, you can add journals to a GFS2 file system with the **gfs2_jadd** command. You can add journals to a GFS2 file system dynamically at any point without expanding the underlying logical volume. The **gfs2_jadd** command must be run on a mounted file system, but it needs to be run on only one node in the cluster. All the other nodes sense that the expansion has occurred.



NOTE

If a GFS2 file system is full, the **gfs2_jadd** command will fail, even if the logical volume containing the file system has been extended and is larger than the file system. This is because in a GFS2 file system, journals are plain files rather than embedded metadata, so simply extending the underlying logical volume will not provide space for the journals.

Before adding journals to a GFS2 file system, you can find out how many journals the GFS2 file system currently contains with the **gfs2_edit -p jindex** command, as in the following example:

```
# gfs2_edit -p jindex /dev/sasdrives/scratch|grep journal
  3/3 [fc7745eb] 4/25 (0x4/0x19): File    journal0
  4/4 [8b70757d] 5/32859 (0x5/0x805b): File    journal1
  5/5 [127924c7] 6/65701 (0x6/0x100a5): File    journal2
```

The format for the basic command to add journals to a GFS2 file system is as follows.

```
gfs2_jadd -j Number MountPoint
```

Number

Specifies the number of new journals to be added.

MountPoint

Specifies the directory where the GFS2 file system is mounted.

In this example, one journal is added to the file system on the **/mygfs2** directory.

```
gfs2_jadd -j 1 /mygfs2
```

CHAPTER 4. GFS2 QUOTA MANAGEMENT

File system quotas are used to limit the amount of file system space a user or group can use. A user or group does not have a quota limit until one is set. When a GFS2 file system is mounted with the **quota=on** or **quota=account** option, GFS2 keeps track of the space used by each user and group even when there are no limits in place. GFS2 updates quota information in a transactional way so system crashes do not require quota usages to be reconstructed.

To prevent a performance slowdown, a GFS2 node synchronizes updates to the quota file only periodically. The fuzzy quota accounting can allow users or groups to slightly exceed the set limit. To minimize this, GFS2 dynamically reduces the synchronization period as a hard quota limit is approached.

NOTE

GFS2 supports the standard Linux quota facilities. In order to use this you will need to install the **quota** RPM. This is the preferred way to administer quotas on GFS2 and should be used for all new deployments of GFS2 using quotas. This section documents GFS2 quota management using these facilities.

For more information on disk quotas, see the **man** pages of the following commands:

- **quotacheck**
- **edquota**
- **repquota**
- **quota**

4.1. CONFIGURING GFS2 DISK QUOTAS

To implement disk quotas, use the following steps:

1. Set up quotas in enforcement or accounting mode.
2. Initialize the quota database file with current block usage information.
3. Assign quota policies. (In accounting mode, these policies are not enforced.)

Each of these steps is discussed in detail in the following sections.

4.1.1. Setting up quotas in enforcement or accounting mode

In GFS2 file systems, quotas are disabled by default. To enable quotas for a file system, mount the file system with the **quota=on** option specified.

To mount a file system with quotas enabled, specify **quota=on** for the **options** argument when creating the GFS2 file system resource in a cluster. For example, the following command specifies that the GFS2 **Filesystem** resource being created will be mounted with quotas enabled.

```
# pcs resource create gfs2mount Filesystem options="quota=on" device=BLOCKDEVICE
directory=MOUNTPOINT fstype=gfs2 clone
```

It is possible to keep track of disk usage and maintain quota accounting for every user and group without enforcing the limit and warn values. To do this, mount the file system with the **quota=account** option specified.

To mount a file system with quotas disabled, specify **quota=off** for the **options** argument when creating the GFS2 file system resource in a cluster.

4.1.2. Creating the quota database files

After each quota-enabled file system is mounted, the system is capable of working with disk quotas. However, the file system itself is not yet ready to support quotas. The next step is to run the **quotacheck** command.

The **quotacheck** command examines quota-enabled file systems and builds a table of the current disk usage per file system. The table is then used to update the operating system's copy of disk usage. In addition, the file system's disk quota files are updated.

To create the quota files on the file system, use the **-u** and the **-g** options of the **quotacheck** command; both of these options must be specified for user and group quotas to be initialized. For example, if quotas are enabled for the /**home** file system, create the files in the /**home** directory:

```
quotacheck -ug /home
```

4.1.3. Assigning quotas per user

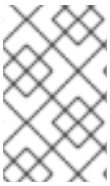
The last step is assigning the disk quotas with the **edquota** command. Note that if you have mounted your file system in accounting mode (with the **quota=account** option specified), the quotas are not enforced.

To configure the quota for a user, as root in a shell prompt, execute the command:

```
# edquota username
```

Perform this step for each user who needs a quota. For example, if a quota is enabled for the /**home** partition (/**dev**/**VolGroup00**/**LogVol02** in the example below) and the command **edquota testuser** is executed, the following is shown in the editor configured as the default for the system:

```
Disk quotas for user testuser (uid 501):
Filesystem                   blocks    soft    hard    inodes   soft   hard
/dev/VolGroup00/LogVol02  440436       0       0
```



NOTE

The text editor defined by the **EDITOR** environment variable is used by **edquota**. To change the editor, set the **EDITOR** environment variable in your ~/**.bash_profile** file to the full path of the editor of your choice.

The first column is the name of the file system that has a quota enabled for it. The second column shows how many blocks the user is currently using. The next two columns are used to set soft and hard block limits for the user on the file system.

The soft block limit defines the maximum amount of disk space that can be used.

The hard block limit is the absolute maximum amount of disk space that a user or group can use. Once this limit is reached, no further disk space can be used.

The GFS2 file system does not maintain quotas for inodes, so these columns do not apply to GFS2 file systems and will be blank.

If any of the values are set to 0, that limit is not set. In the text editor, change the limits. For example:

```
Disk quotas for user testuser (uid 501):
Filesystem              blocks    soft    hard    inodes  soft  hard
/dev/VolGroup00/LogVol02  440436  500000  550000
```

To verify that the quota for the user has been set, use the following command:

```
# quota testuser
```

You can also set quotas from the command line with the **setquota** command. For information on the **setquota** command, see the **setquota**(8) man page.

4.1.4. Assigning quotas per group

Quotas can also be assigned on a per-group basis. Note that if you have mounted your file system in accounting mode (with the **account=on** option specified), the quotas are not enforced.

To set a group quota for the **devel** group (the group must exist prior to setting the group quota), use the following command:

```
# edquota -g devel
```

This command displays the existing quota for the group in the text editor:

```
Disk quotas for group devel (gid 505):
Filesystem              blocks   soft    hard    inodes  soft  hard
/dev/VolGroup00/LogVol02  440400      0       0
```

The GFS2 file system does not maintain quotas for inodes, so these columns do not apply to GFS2 file systems and will be blank. Modify the limits, then save the file.

To verify that the group quota has been set, use the following command:

```
$ quota -g devel
```

4.2. MANAGING GFS2 DISK QUOTAS

If quotas are implemented, they need some maintenance, mostly in the form of watching to see if the quotas are exceeded and making sure the quotas are accurate.

If users repeatedly exceed their quotas or consistently reach their soft limits, a system administrator has a few choices to make depending on what type of users they are and how much disk space impacts their work. The administrator can either help the user determine how to use less disk space or increase the user's disk quota.

You can create a disk usage report by running the **repquota** utility. For example, the command **repquota /home** produces this output:

```
* Report for user quotas on device /dev/mapper/VolGroup00-LogVol02
Block grace time: 7days; Inode grace time: 7days
  Block limits   File limits
User  used soft hard grace used soft hard grace
----------------------------------------------------------------------
root      --      36       0       0              4     0     0
kristin   --     540       0       0            125     0     0
testuser  --  440400  500000  550000          37418     0     0
```

To view the disk usage report for all (option **-a**) quota-enabled file systems, use the command:

```
# repquota -a
```

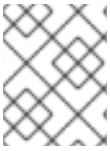
The **--** displayed after each user is a quick way to determine whether the block limits have been exceeded. If the block soft limit is exceeded, a **+** appears in place of the first **-** in the output. The second **-** indicates the inode limit, but GFS2 file systems do not support inode limits so that character will remain as **-**. GFS2 file systems do not support a grace period, so the **grace** column will remain blank.

Note that the **repquota** command is not supported over NFS, irrespective of the underlying file system.

4.3. KEEPING GFS2 DISK QUOTAS ACCURATE WITH THE QUOTACHECK COMMAND

If you enable quotas on your file system after a period of time when you have been running with quotas disabled, you should run the **quotacheck** command to create, check, and repair quota files. Additionally, you may want to run the **quotacheck** command if you think your quota files may not be accurate, as may occur when a file system is not unmounted cleanly after a system crash.

For more information about the **quotacheck** command, see the **quotacheck** man page.



NOTE

Run **quotacheck** when the file system is relatively idle on all nodes because disk activity may affect the computed quota values.

4.4. SYNCHRONIZING QUOTAS WITH THE QUOTASYNC COMMAND

GFS2 stores all quota information in its own internal file on disk. A GFS2 node does not update this quota file for every file system write; rather, by default it updates the quota file once every 60 seconds. This is necessary to avoid contention among nodes writing to the quota file, which would cause a slowdown in performance.

As a user or group approaches their quota limit, GFS2 dynamically reduces the time between its quota-file updates to prevent the limit from being exceeded. The normal time period between quota synchronizations is a tunable parameter, **quota_quantum**. You can change this from its default value of 60 seconds using the **quota_quantum=** mount option. Table 25.2. GFS2-Specific Mount Options. The **quota_quantum** parameter must be set on each node and each time the file system is mounted. Changes to the **quota_quantum** parameter are not persistent across unmounts. You can update the **quota_quantum** value with the **mount -o remount**.

You can use the **quotasync** command to synchronize the quota information from a node to the on-disk quota file between the automatic updates performed by GFS2. Usage **Synchronizing Quota Information**

```
# `quotasync [-ug -a|mountpoint..a`].
```

u

Sync the user quota files.

g

Sync the group quota files

a

Sync all file systems that are currently quota-enabled and support sync. When -a is absent, a file system mountpoint should be specified.

mountpoint

Specifies the GFS2 file system to which the actions apply.

You can tune the time between synchronizations by specifying a **quota-quantum** mount option.

```
# mount -o quota_quantum=secs,remount BlockDevice MountPoint
```

MountPoint

Specifies the GFS2 file system to which the actions apply.

secs

Specifies the new time period between regular quota-file synchronizations by GFS2. Smaller values may increase contention and slow down performance.

The following example synchronizes all the cached dirty quotas from the node it is run on to the on-disk quota file for the file system **/mnt/mygfs2**.

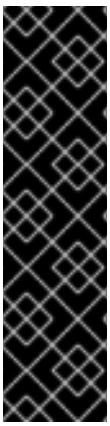
```
# quotasync -ug /mnt/mygfs2
```

This following example changes the default time period between regular quota-file updates to one hour (3600 seconds) for file system **/mnt/mygfs2** when remounting that file system on logical volume **/dev/volgroup/logical_volume**.

```
# mount -o quota_quantum=3600,remount /dev/volgroup/logical_volume /mnt/mygfs2
```

CHAPTER 5. GFS2 FILE SYSTEM REPAIR

When nodes fail with the file system mounted, file system journaling allows fast recovery. However, if a storage device loses power or is physically disconnected, file system corruption may occur. (Journaling cannot be used to recover from storage subsystem failures.) When that type of corruption occurs, you can recover the GFS2 file system by using the **fsck.gfs2** command.



IMPORTANT

The **fsck.gfs2** command must be run only on a file system that is unmounted from all nodes. When the file system is being managed as a Pacemaker cluster resource, you can disable the file system resource, which unmounts the file system. After running the **fsck.gfs2** command, you enable the file system resource again. The *timeout* value specified with the **--wait** option of the **pcs resource disable** indicates a value in seconds.

```
# pcs resource disable --wait=timeoutvalue resource_id
[fsck.gfs2]
# pcs resource enable resource_id
```

To ensure that **fsck.gfs2** command does not run on a GFS2 file system at boot time, you can set the **run_fsck** parameter of the **options** argument when creating the GFS2 file system resource in a cluster. Specifying **"run_fsck=no"** will indicate that you should not run the **fsck** command.

5.1. DETERMING REQUIRED MEMORY FOR RUNNING FSCK.GFS2

Running the **fsck.gfs2** command may require system memory above and beyond the memory used for the operating system and kernel. Larger file systems in particular may require additional memory to run this command.

The following table shows approximate values of memory that may be required to run **fsck.gfs2** file systems on GFS2 file systems that are 1TB, 10TB, and 100TB in size with a block size of 4K.

GFS2 file system size	Approximate memory required to run **fsck.gfs2**
1 TB	0.16 GB
10 TB	1.6 GB
100 TB	16 GB

Note that a smaller block size for the file system would require a larger amount of memory. For example, GFS2 file systems with a block size of 1K would require four times the amount of memory indicated in this table.

5.2. REPAIRING A GFS2 FILESYSTEM

The following shows the format of the **fsck.gfs2** command to repair a GFS2 filesystem.

```
fsck.gfs2 -y BlockDevice
```

-y

The **-y** flag causes all questions to be answered with **yes**. With the **-y** flag specified, the **fsck.gfs2** command does not prompt you for an answer before making changes.

BlockDevice

Specifies the block device where the GFS2 file system resides.

In this example, the GFS2 file system residing on block device **/dev/testvg/testlv** is repaired. All queries to repair are automatically answered with **yes**.

```
# fsck.gfs2 -y /dev/testvg/testlv
Initializing fsck
Validating Resource Group index.
Level 1 RG check.
(level 1 passed)
Clearing journals (this may take a while)...
Journals cleared.
Starting pass1
Pass1 complete
Starting pass1b
Pass1b complete
Starting pass1c
Pass1c complete
Starting pass2
Pass2 complete
Starting pass3
Pass3 complete
Starting pass4
Pass4 complete
Starting pass5
Pass5 complete
Writing changes to disk
fsck.gfs2 complete
```

CHAPTER 6. IMPROVING GFS2 PERFORMANCE

This section provides advice for improving GFS2 peformance.

For general recommendations for deploying and upgrading Red Hat Enterprise Linux clusters using the High Availability Add-On and Red Hat Global File System 2 (GFS2) see the article "Red Hat Enterprise Linux Cluster, High Availability, and GFS Deployment Best Practices" on the Red Hat Customer Portal at https://access.redhat.com/kb/docs/DOC-40821.

6.1. GFS2 FILE SYSTEM DEFRAGMENTATION

While there is no defragmentation tool for GFS2 on Red Hat Enterprise Linux, you can defragment individual files by identifying them with the **filefrag** tool, copying them to temporary files, and renaming the temporary files to replace the originals.

6.2. GFS2 NODE LOCKING

In order to get the best performance from a GFS2 file system, it is important to understand some of the basic theory of its operation. A single node file system is implemented alongside a cache, the purpose of which is to eliminate latency of disk accesses when using frequently requested data. In Linux the page cache (and historically the buffer cache) provide this caching function.

With GFS2, each node has its own page cache which may contain some portion of the on-disk data. GFS2 uses a locking mechanism called *glocks* (pronounced gee-locks) to maintain the integrity of the cache between nodes. The glock subsystem provides a cache management function which is implemented using the *distributed lock manager* (DLM) as the underlying communication layer.

The glocks provide protection for the cache on a per-inode basis, so there is one lock per inode which is used for controlling the caching layer. If that glock is granted in shared mode (DLM lock mode: PR) then the data under that glock may be cached upon one or more nodes at the same time, so that all the nodes may have local access to the data.

If the glock is granted in exclusive mode (DLM lock mode: EX) then only a single node may cache the data under that glock. This mode is used by all operations which modify the data (such as the **write** system call).

If another node requests a glock which cannot be granted immediately, then the DLM sends a message to the node or nodes which currently hold the glocks blocking the new request to ask them to drop their locks. Dropping glocks can be (by the standards of most file system operations) a long process. Dropping a shared glock requires only that the cache be invalidated, which is relatively quick and proportional to the amount of cached data.

Dropping an exclusive glock requires a log flush, and writing back any changed data to disk, followed by the invalidation as per the shared glock.

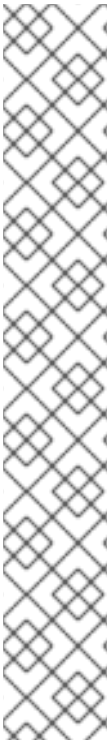
The difference between a single node file system and GFS2, then, is that a single node file system has a single cache and GFS2 has a separate cache on each node. In both cases, latency to access cached data is of a similar order of magnitude, but the latency to access uncached data is much greater in GFS2 if another node has previously cached that same data.

Operations such as **read** (buffered), **stat,** and **readdir** only require a shared glock. Operations such as **write** (buffered), **mkdir**, **rmdir**, and **unlink** require an exclusive glock. Direct I/O read/write operations require a deferred glock if no allocation is taking place, or an exclusive glock if the write requires an allocation (that is, extending the file, or hole filling).

There are two main performance considerations which follow from this. First, read-only operations parallelize extremely well across a cluster, since they can run independently on every node. Second, operations requiring an exclusive glock can reduce performance, if there are multiple nodes contending for access to the same inode(s). Consideration of the working set on each node is thus an important factor in GFS2 file system performance such as when, for example, you perform a file system backup, as described in Backing up a GFS2 file system .

A further consequence of this is that we recommend the use of the **noatime** or **nodiratime** mount option with GFS2 whenever possible, with the preference for **noatime** where the application allows for this. This prevents reads from requiring exclusive locks to update the **atime** timestamp.

For users who are concerned about the working set or caching efficiency, GFS2 provides tools that allow you to monitor the performance of a GFS2 file system: Performance Co-Pilot and GFS2 tracepoints.



NOTE

Due to the way in which GFS2's caching is implemented the best performance is obtained when either of the following takes place:

- An inode is used in a read-only fashion across all nodes.
- An inode is written or modified from a single node only.

Note that inserting and removing entries from a directory during file creation and deletion counts as writing to the directory inode.

It is possible to break this rule provided that it is broken relatively infrequently. Ignoring this rule too often will result in a severe performance penalty.

If you **mmap**() a file on GFS2 with a read/write mapping, but only read from it, this only counts as a read.

If you do not set the **noatime mount** parameter, then reads will also result in writes to update the file timestamps. We recommend that all GFS2 users should mount with **noatime** unless they have a specific requirement for **atime**.

6.3. ISSUES WITH POSIX LOCKING

When using Posix locking, you should take the following into account:

- Use of Flocks will yield faster processing than use of Posix locks.
- Programs using Posix locks in GFS2 should avoid using the **GETLK** function since, in a clustered environment, the process ID may be for a different node in the cluster.

6.4. PERFORMANCE TUNING WITH GFS2

It is usually possible to alter the way in which a troublesome application stores its data in order to gain a considerable performance advantage.

A typical example of a troublesome application is an email server. These are often laid out with a spool directory containing files for each user (**mbox**), or with a directory for each user containing a file for each message (**maildir**). When requests arrive over IMAP, the ideal arrangement is to give each user an

affinity to a particular node. That way their requests to view and delete email messages will tend to be served from the cache on that one node. Obviously if that node fails, then the session can be restarted on a different node.

When mail arrives by means of SMTP, then again the individual nodes can be set up so as to pass a certain user's mail to a particular node by default. If the default node is not up, then the message can be saved directly into the user's mail spool by the receiving node. Again this design is intended to keep particular sets of files cached on just one node in the normal case, but to allow direct access in the case of node failure.

This setup allows the best use of GFS2's page cache and also makes failures transparent to the application, whether **imap** or **smtp**.

Backup is often another tricky area. Again, if it is possible it is greatly preferable to back up the working set of each node directly from the node which is caching that particular set of inodes. If you have a backup script which runs at a regular point in time, and that seems to coincide with a spike in the response time of an application running on GFS2, then there is a good chance that the cluster may not be making the most efficient use of the page cache.

Obviously, if you are in the position of being able to stop the application in order to perform a backup, then this will not be a problem. On the other hand, if a backup is run from just one node, then after it has completed a large portion of the file system will be cached on that node, with a performance penalty for subsequent accesses from other nodes. This can be mitigated to a certain extent by dropping the VFS page cache on the backup node after the backup has completed with following command:

```
echo -n 3 >/proc/sys/vm/drop_caches
```

However this is not as good a solution as taking care to ensure the working set on each node is either shared, mostly read-only across the cluster, or accessed largely from a single node.

6.5. TROUBLESHOOTING GFS2 PERFORMANCE WITH THE GFS2 LOCK DUMP

If your cluster performance is suffering because of inefficient use of GFS2 caching, you may see large and increasing I/O wait times. You can make use of GFS2's lock dump information to determine the cause of the problem.

This section provides an overview of the GFS2 lock dump.

The GFS2 lock dump information can be gathered from the **debugfs** file which can be found at the following path name, assuming that **debugfs** is mounted on **/sys/kernel/debug/**:

```
/sys/kernel/debug/gfs2/fsname/glocks
```

The content of the file is a series of lines. Each line starting with G: represents one glock, and the following lines, indented by a single space, represent an item of information relating to the glock immediately before them in the file.

The best way to use the **debugfs** file is to use the **cat** command to take a copy of the complete content of the file (it might take a long time if you have a large amount of RAM and a lot of cached inodes) while the application is experiencing problems, and then looking through the resulting data at a later date.

NOTE

It can be useful to make two copies of the **debugfs** file, one a few seconds or even a minute or two after the other. By comparing the holder information in the two traces relating to the same glock number, you can tell whether the workload is making progress (it is just slow) or whether it has become stuck (which is always a bug and should be reported to Red Hat support immediately).

Lines in the **debugfs** file starting with H: (holders) represent lock requests either granted or waiting to be granted. The flags field on the holders line f: shows which: The 'W' flag refers to a waiting request, the 'H' flag refers to a granted request. The glocks which have large numbers of waiting requests are likely to be those which are experiencing particular contention.

Table 6.1, “Glock flags” shows the meanings of the different glock flags and Table 6.2, “Glock holder flags” shows the meanings of the different glock holder flags.

Table 6.1. Glock flags

Flag	Name	Meaning
b	Blocking	Valid when the locked flag is set, and indicates that the operation that has been requested from the DLM may block. This flag is cleared for demotion operations and for "try" locks. The purpose of this flag is to allow gathering of stats of the DLM response time independent from the time taken by other nodes to demote locks.
d	Pending demote	A deferred (remote) demote request
D	Demote	A demote request (local or remote)
f	Log flush	The log needs to be committed before releasing this glock
F	Frozen	Replies from remote nodes ignored - recovery is in progress. This flag is not related to file system freeze, which uses a different mechanism, but is used only in recovery.
i	Invalidate in progress	In the process of invalidating pages under this glock
I	Initial	Set when DLM lock is associated with this glock

Flag	Name	Meaning
l	Locked	The glock is in the process of changing state
L	LRU	Set when the glock is on the LRU list
o	Object	Set when the glock is associated with an object (that is, an inode for type 2 glocks, and a resource group for type 3 glocks)
p	Demote in progress	The glock is in the process of responding to a demote request
q	Queued	Set when a holder is queued to a glock, and cleared when the glock is held, but there are no remaining holders. Used as part of the algorithm the calculates the minimum hold time for a glock.
r	Reply pending	Reply received from remote node is awaiting processing
y	Dirty	Data needs flushing to disk before releasing this glock

Table 6.2. Glock holder flags

Flag	Name	Meaning
a	Async	Do not wait for glock result (will poll for result later)
A	Any	Any compatible lock mode is acceptable
c	No cache	When unlocked, demote DLM lock immediately
e	No expire	Ignore subsequent lock cancel requests
E	exact	Must have exact lock mode
F	First	Set when holder is the first to be granted for this lock

Flag	Name	Meaning
H	Holder	Indicates that requested lock is granted
p	Priority	Enqueue holder at the head of the queue
t	Try	A "try" lock
T	Try 1CB	A "try" lock that sends a callback
W	Wait	Set while waiting for request to complete

Having identified a glock which is causing a problem, the next step is to find out which inode it relates to. The glock number (n: on the G: line) indicates this. It is of the form *type*/*number* and if *type* is 2, then the glock is an inode glock and the *number* is an inode number. To track down the inode, you can then run **find -inum *number*** where *number* is the inode number converted from the hex format in the glocks file into decimal.

WARNING

If you run the **find** command on a file system when it is experiencing lock contention, you are likely to make the problem worse. It is a good idea to stop the application before running the **find** command when you are looking for contended inodes.

Table 6.3, "Glock types" shows the meanings of the different glock types.

Table 6.3. Glock types

Type number	Lock type	Use
1	Trans	Transaction lock
2	Inode	Inode metadata and data
3	Rgrp	Resource group metadata
4	Meta	The superblock
5	Iopen	Inode last closer detection
6	Flock	**flock**(2) syscall

Type number	Lock type	Use
8	Quota	Quota operations
9	Journal	Journal mutex

If the glock that was identified was of a different type, then it is most likely to be of type 3: (resource group). If you see significant numbers of processes waiting for other types of glock under normal loads, report this to Red Hat support.

If you do see a number of waiting requests queued on a resource group lock there may be a number of reasons for this. One is that there are a large number of nodes compared to the number of resource groups in the file system. Another is that the file system may be very nearly full (requiring, on average, longer searches for free blocks). The situation in both cases can be improved by adding more storage and using the **gfs2_grow** command to expand the file system.

6.6. ENABLING DATA JOURNALING

Ordinarily, GFS2 writes only metadata to its journal. File contents are subsequently written to disk by the kernel's periodic sync that flushes file system buffers. An **fsync()** call on a file causes the file's data to be written to disk immediately. The call returns when the disk reports that all data is safely written.

Data journaling can result in a reduced **fsync()** time for very small files because the file data is written to the journal in addition to the metadata. This advantage rapidly reduces as the file size increases. Writing to medium and larger files will be much slower with data journaling turned on.

Applications that rely on **fsync()** to sync file data may see improved performance by using data journaling. Data journaling can be enabled automatically for any GFS2 files created in a flagged directory (and all its subdirectories). Existing files with zero length can also have data journaling turned on or off.

Enabling data journaling on a directory sets the directory to "inherit jdata", which indicates that all files and directories subsequently created in that directory are journaled. You can enable and disable data journaling on a file with the **chattr** command.

The following commands enable data journaling on the **/mnt/gfs2/gfs2_dir/newfile** file and then check whether the flag has been set properly.

```
# chattr +j /mnt/gfs2/gfs2_dir/newfile
# lsattr /mnt/gfs2/gfs2_dir
---------j--- /mnt/gfs2/gfs2_dir/newfile
```

The following commands disable data journaling on the **/mnt/gfs2/gfs2_dir/newfile** file and then check whether the flag has been set properly.

```
# chattr -j /mnt/gfs2/gfs2_dir/newfile
# lsattr /mnt/gfs2/gfs2_dir
---------- /mnt/gfs2/gfs2_dir/newfile
```

You can also use the **chattr** command to set the **j** flag on a directory. When you set this flag for a directory, all files and directories subsequently created in that directory are journaled. The following set of commands sets the **j** flag on the **gfs2_dir** directory, then checks whether the flag has been set

properly. After this, the commands create a new file called **newfile** in the **/mnt/gfs2/gfs2_dir** directory and then check whether the **j** flag has been set for the file. Since the **j** flag is set for the directory, then **newfile** should also have journaling enabled.

```
# chattr -j /mnt/gfs2/gfs2_dir
# lsattr /mnt/gfs2
---------j--- /mnt/gfs2/gfs2_dir
# touch /mnt/gfs2/gfs2_dir/newfile
# lsattr /mnt/gfs2/gfs2_dir
---------j--- /mnt/gfs2/gfs2_dir/newfile
```

CHAPTER 7. DIAGNOSING AND CORRECTING PROBLEMS WITH GFS2 FILE SYSTEMS

This section provides information about some common GFS2 issues and how to address them.

7.1. GFS2 FILESYSTEM UNAVAILABLE TO A NODE (THE GFS2 WITHDRAW FUNCTION)

The GFS2 *withdraw* function is a data integrity feature of the GFS2 file system that prevents potential file system damage due to faulty hardware or kernel software. If the GFS2 kernel module detects an inconsistency while using a GFS2 file system on any given cluster node, it withdraws from the file system, leaving it unavailable to that node until it is unmounted and remounted (or the machine detecting the problem is rebooted). All other mounted GFS2 file systems remain fully functional on that node. (The GFS2 withdraw function is less severe than a kernel panic, which causes the node to be fenced.)

The main categories of inconsistency that can cause a GFS2 withdraw are as follows:

- Inode consistency error
- Resource group consistency error
- Journal consistency error
- Magic number metadata consistency error
- Metadata type consistency error

An example of an inconsistency that would cause a GFS2 withdraw is an incorrect block count for a file's inode. When GFS2 deletes a file, it systematically removes all the data and metadata blocks referenced by that file. When done, it checks the inode's block count. If the block count is not 1 (meaning all that is left is the disk inode itself), that indicates a file system inconsistency, since the inode's block count did not match the actual blocks used for the file.

In many cases, the problem may have been caused by faulty hardware (faulty memory, motherboard, HBA, disk drives, cables, and so forth). It may also have been caused by a kernel bug (another kernel module accidentally overwriting GFS2's memory), or actual file system damage (caused by a GFS2 bug).

In most cases, the best way to recover from a withdrawn GFS2 file system is to reboot or fence the node. The withdrawn GFS2 file system will give you an opportunity to relocate services to another node in the cluster. After services are relocated you can reboot the node or force a fence with this command.

```
# pcs stonith fence node
```

WARNING

Do not try to unmount and remount the file system manually with the **umount** and **mount** commands. You must use the **pcs** command, otherwise Pacemaker will detect the file system service has disappeared and fence the node.

The consistency problem that caused the withdraw may make stopping the file system service impossible as it may cause the system to hang.

If the problem persists after a remount, you should stop the file system service to unmount the file system from all nodes in the cluster, then perform a file system check with the fsck.gfs2 command before restarting the service with the following procedure.

1. Reboot the affected node.
2. Disable the non-clone file system service in Pacemaker to unmount the file system from every node in the cluster.

```
# pcs resource disable --wait=100 mydata_fs
```

3. From one node of the cluster, run the **fsck.gfs2** command on the file system device to check for and repair any file system damage.

```
# fsck.gfs2 -y /dev/vg_mydata/mydata > /tmp/fsck.out
```

4. Remount the GFS2 file system from all nodes by re-enabling the file system service:

```
# pcs resource enable --wait=100 mydata_fs
```

You can override the GFS2 withdraw function by mounting the file system with the **-o errors=panic** option specified in the file system service.

```
# pcs resource update mydata_fs “options=noatime,errors=panic”
```

When this option is specified, any errors that would normally cause the system to withdraw force a kernel panic instead. This stops the node’s communications, which causes the node to be fenced. This is especially useful for clusters that are left unattended for long periods of time without monitoring or intervention.

Internally, the GFS2 withdraw function works by disconnecting the locking protocol to ensure that all further file system operations result in I/O errors. As a result, when the withdraw occurs, it is normal to see a number of I/O errors from the device mapper device reported in the system logs.

7.2. GFS2 FILE SYSTEM HANGS AND REQUIRES REBOOT OF ONE NODE

If your GFS2 file system hangs and does not return commands run against it, but rebooting one specific node returns the system to normal, this may be indicative of a locking problem or bug. Should this occur, gather GFS2 data during one of these occurences and open a support ticket with Red Hat Support, as described in Gathering GFS2 data for troubleshooting.

7.3. GFS2 FILE SYSTEM HANGS AND REQUIRES REBOOT OF ALL NODES

If your GFS2 file system hangs and does not return commands run against it, requiring that you reboot all nodes in the cluster before using it, check for the following issues.

- You may have had a failed fence. GFS2 file systems will freeze to ensure data integrity in the event of a failed fence. Check the messages logs to see if there are any failed fences at the time of the hang. Ensure that fencing is configured correctly.

- The GFS2 file system may have withdrawn. Check through the messages logs for the word **withdraw** and check for any messages and call traces from GFS2 indicating that the file system has been withdrawn. A withdraw is indicative of file system corruption, a storage failure, or a bug. At the earliest time when it is convenient to unmount the file system, you should perform the following procedure:

  a. Reboot the node on which the withdraw occurred.

  ```
  # /sbin/reboot
  ```

  b. Stop the file system resource to unmount the GFS2 file system on all nodes.

  ```
  # pcs resource disable --wait=100 mydata_fs
  ```

  c. Capture the metadata with the **gfs2_edit savemeta…** command. You should ensure that there is sufficient space for the file, which in some cases may be large. In this example, the metadata is saved to a file in the **/root** directory.

  ```
  # gfs2_edit savemeta /dev/vg_mydata/mydata /root/gfs2metadata.gz
  ```

  d. Update the **gfs2-utils** package.

  ```
  # sudo yum update gfs2-utils
  ```

  e. On one node, run the **fsck.gfs2** command on the file system to ensure file system integrity and repair any damage.

  ```
  # fsck.gfs2 -y /dev/vg_mydata/mydata > /tmp/fsck.out
  ```

  f. After the **fsck.gfs2** command has completed, re-enable the file system resource to return it to service:

  ```
  # pcs resource enable --wait=100 mydata_fs
  ```

  g. Open a support ticket with Red Hat Support. Inform them you experienced a GFS2 withdraw and provide logs and the debugging information generated by the **sosreports** and **gfs2_edit savemeta** commands.
  In some instances of a GFS2 withdraw, commands can hang that are trying to access the file system or its block device. In these cases a hard reboot is required to reboot the cluster.

  For information on the GFS2 withdraw function, see GFS2 filesystem unavailable to a node (the GFS2 withdraw function).

- This error may be indicative of a locking problem or bug. Gather data during one of these occurrences and open a support ticket with Red Hat Support, as described in Gathering GFS2 data for troubleshooting.

7.4. GFS2 FILE SYSTEM DOES NOT MOUNT ON NEWLY ADDED CLUSTER NODE

If you add a new node to a cluster and find that you cannot mount your GFS2 file system on that node, you may have fewer journals on the GFS2 file system than nodes attempting to access the GFS2 file system. You must have one journal per GFS2 host you intend to mount the file system on (with the exception of GFS2 file systems mounted with the **spectator** mount option set, since these do not require a journal). You can add journals to a GFS2 file system with the **gfs2_jadd** command. Adding journals to a GFS2 file system.

7.5. SPACE INDICATED AS USED IN EMPTY FILE SYSTEM

If you have an empty GFS2 file system, the **df** command will show that there is space being taken up. This is because GFS2 file system journals consume space (number of journals * journal size) on disk. If you created a GFS2 file system with a large number of journals or specified a large journal size then you will be see (number of journals * journal size) as already in use when you execute the **df** command. Even if you did not specify a large number of journals or large journals, small GFS2 file systems (in the 1GB or less range) will show a large amount of space as being in use with the default GFS2 journal size.

7.6. GATHERING GFS2 DATA FOR TROUBLESHOOTING

If your GFS2 file system hangs and does not return commands run against it and you find that you need to open a ticket with Red Hat Support, you should first gather the following data:

- The GFS2 lock dump for the file system on each node:

  ```
  cat /sys/kernel/debug/gfs2/fsname/glocks >glocks.fsname.nodename
  ```

- The DLM lock dump for the file system on each node: You can get this information with the **dlm_tool**:

  ```
  dlm_tool lockdebug -sv lsname.
  ```

  In this command, *lsname* is the lockspace name used by DLM for the file system in question. You can find this value in the output from the **group_tool** command.

- The output from the **sysrq -t** command.

- The contents of the **/var/log/messages** file.

Once you have gathered that data, you can open a ticket with Red Hat Support and provide the data you have collected.

CHAPTER 8. DEBUGGING GFS2 FILE SYSTEMS WITH GFS2 TRACEPOINTS AND THE DEBUGFS GLOCKS FILE

This section describes both the glock **debugfs** interface and the GFS2 tracepoints. It is intended for advanced users who are familiar with file system internals who would like to learn more about the design of GFS2 and how to debug GFS2-specific issues.

8.1. GFS2 TRACEPOINT TYPES

There are currently three types of GFS2 tracepoints: *glock* (pronounced "gee-lock") tracepoints, *bmap* tracepoints and *log* tracepoints. These can be used to monitor a running GFS2 file system and give additional information to that which can be obtained with the debugging options supported in previous releases of Red Hat Enterprise Linux. Tracepoints are particularly useful when a problem, such as a hang or performance issue, is reproducible and thus the tracepoint output can be obtained during the problematic operation. In GFS2, glocks are the primary cache control mechanism and they are the key to understanding the performance of the core of GFS2. The bmap (block map) tracepoints can be used to monitor block allocations and block mapping (lookup of already allocated blocks in the on-disk metadata tree) as they happen and check for any issues relating to locality of access. The log tracepoints keep track of the data being written to and released from the journal and can provide useful information on that part of GFS2.

The tracepoints are designed to be as generic as possible. This should mean that it will not be necessary to change the API during the course of Red Hat Enterprise Linux 8. On the other hand, users of this interface should be aware that this is a debugging interface and not part of the normal Red Hat Enterprise Linux 8 API set, and as such Red Hat makes no guarantees that changes in the GFS2 tracepoints interface will not occur.

Tracepoints are a generic feature of Red Hat Enterprise Linux and their scope goes well beyond GFS2. In particular they are used to implement the **blktrace** infrastructure and the **blktrace** tracepoints can be used in combination with those of GFS2 to gain a fuller picture of the system performance. Due to the level at which the tracepoints operate, they can produce large volumes of data in a very short period of time. They are designed to put a minimum load on the system when they are enabled, but it is inevitable that they will have some effect. Filtering events by a variety of means can help reduce the volume of data and help focus on obtaining just the information which is useful for understanding any particular situation.

8.2. TRACEPOINTS

The tracepoints can be found under the **/sys/kernel/debug/tracing/** directory assuming that **debugfs** is mounted in the standard place at the **/sys/kernel/debug** directory. The **events** subdirectory contains all the tracing events that may be specified and, provided the **gfs2** module is loaded, there will be a **gfs2** subdirectory containing further subdirectories, one for each GFS2 event. The contents of the **/sys/kernel/debug/tracing/events/gfs2** directory should look roughly like the following:

```
[root@chywoon gfs2]# ls
enable          gfs2_bmap       gfs2_glock_queue        gfs2_log_flush
filter          gfs2_demote_rq  gfs2_glock_state_change  gfs2_pin
gfs2_block_alloc  gfs2_glock_put  gfs2_log_blocks         gfs2_promote
```

To enable all the GFS2 tracepoints, enter the following command:

```
[root@chywoon gfs2]# echo -n 1 >/sys/kernel/debug/tracing/events/gfs2/enable
```

To enable a specific tracepoint, there is an **enable** file in each of the individual event subdirectories. The same is true of the **filter** file which can be used to set an event filter for each event or set of events. The meaning of the individual events is explained in more detail below.

The output from the tracepoints is available in ASCII or binary format. This appendix does not currently cover the binary interface. The ASCII interface is available in two ways. To list the current content of the ring buffer, you can enter the following command:

```
[root@chywoon gfs2]# cat /sys/kernel/debug/tracing/trace
```

This interface is useful in cases where you are using a long-running process for a certain period of time and, after some event, want to look back at the latest captured information in the buffer. An alternative interface, **/sys/kernel/debug/tracing/trace_pipe**, can be used when all the output is required. Events are read from this file as they occur; there is no historical information available through this interface. The format of the output is the same from both interfaces and is described for each of the GFS2 events in the later sections of this appendix.

A utility called **trace-cmd** is available for reading tracepoint data. For more information on this utility, see the link in Section 8.10, "References". The **trace-cmd** utility can be used in a similar way to the **strace** utility, for example to run a command while gathering trace data from various sources.

8.3. GLOCKS

To understand GFS2, the most important concept to understand, and the one which sets it aside from other file systems, is the concept of glocks. In terms of the source code, a glock is a data structure that brings together the DLM and caching into a single state machine. Each glock has a 1:1 relationship with a single DLM lock, and provides caching for that lock state so that repetitive operations carried out from a single node of the file system do not have to repeatedly call the DLM, and thus they help avoid unnecessary network traffic. There are two broad categories of glocks, those which cache metadata and those which do not. The inode glocks and the resource group glocks both cache metadata, other types of glocks do not cache metadata. The inode glock is also involved in the caching of data in addition to metadata and has the most complex logic of all glocks.

Table 8.1. Glock Modes and DLM Lock Modes

Glock mode	DLM lock mode	Notes
UN	IV/NL	Unlocked (no DLM lock associated with glock or NL lock depending on I flag)
SH	PR	Shared (protected read) lock
EX	EX	Exclusive lock
DF	CW	Deferred (concurrent write) used for Direct I/O and file system freeze

Glocks remain in memory until either they are unlocked (at the request of another node or at the request of the VM) and there are no local users. At that point they are removed from the glock hash table and freed. When a glock is created, the DLM lock is not associated with the glock immediately. The DLM lock becomes associated with the glock upon the first request to the DLM, and if this request is

successful then the 'I' (initial) flag will be set on the glock. Table 8.4, "Glock flags" shows the meanings of the different glock flags. Once the DLM has been associated with the glock, the DLM lock will always remain at least at NL (Null) lock mode until the glock is to be freed. A demotion of the DLM lock from NL to unlocked is always the last operation in the life of a glock.

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.

The glock state machine is based on a work queue. For performance reasons, tasklets would be preferable; however, in the current implementation we need to submit I/O from that context which prohibits their use.

NOTE

Workqueues have their own tracepoints which can be used in combination with the GFS2 tracepoints.

Table 8.2, "Glock Modes and Data Types" shows what state may be cached under each of the glock modes and whether that cached state may be dirty. This applies to both inode and resource group locks, although there is no data component for the resource group locks, only metadata.

Table 8.2. Glock Modes and Data Types

Glock mode	Cache Data	Cache Metadata	Dirty Data	Dirty Metadata
UN	No	No	No	No
SH	Yes	Yes	No	No
DF	No	Yes	No	No
EX	Yes	Yes	Yes	Yes

8.4. THE GLOCK DEBUGFS INTERFACE

The glock **debugfs** interface allows the visualization of the internal state of the glocks and 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 associated with the glock immediately above it in the file (H: is a holder, I: an inode, and R: a resource group) . Here is an example of what the content of this file might look like:

```
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]
```

The above example is a series of excerpts (from an approximately 18MB file) generated by the command **cat /sys/kernel/debug/gfs2/unity:myfs/glocks >my.lock** during a run of the postmark benchmark on a single node GFS2 file system. 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 does not hold a DLM lock (depending on the I flag as explained above). 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 n: field (number) indicates the number associated with each item. For glocks, that is the type number followed by the glock number so that in the above example, the first glock is n:5/75320; which indicates 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.

NOTE

The glock numbers (n: field) in the debugfs glocks file are in hexadecimal, whereas the tracepoints output lists them in decimal. This is for historical reasons; glock numbers were always written in hex, but decimal was chosen for the tracepoints so that the numbers could easily be compared with the other tracepoint output (from **blktrace** for example) and with output from **stat**(1).

The full listing of all the flags for both the holder and the glock are set out in Table 8.4, "Glock flags" and Table 8.5, "Glock holder flags". The content of lock value blocks is not currently available through the glock **debugfs** interface.

Table 8.3, "Glock Types" shows the meanings of the different glock types.

Table 8.3. Glock Types

Type number	Lock type	Use
1	trans	Transaction lock
2	inode	Inode metadata and data
3	rgrp	Resource group metadata
4	meta	The superblock

Type number	Lock type	Use
5	iopen	Inode last closer detection
6	flock	**flock**(2) syscall
8	quota	Quota operations
9	journal	Journal mutex

One of the more important glock flags is the l (locked) flag. This is the bit lock that 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 through 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.

Table 8.4, "Glock flags" shows the meanings of the different glock flags.

Table 8.4. Glock flags

Flag	Name	Meaning
d	Pending demote	A deferred (remote) demote request
D	Demote	A demote request (local or remote)
f	Log flush	The log needs to be committed before releasing this glock
F	Frozen	Replies from remote nodes ignored - recovery is in progress.
i	Invalidate in progress	In the process of invalidating pages under this glock
I	Initial	Set when DLM lock is associated with this glock
l	Locked	The glock is in the process of changing state
L	LRU	Set when the glock is on the LRU list`

Flag	Name	Meaning
o	Object	Set when the glock is associated with an object (that is, an inode for type 2 glocks, and a resource group for type 3 glocks)
p	Demote in progress	The glock is in the process of responding to a demote request
q	Queued	Set when a holder is queued to a glock, and cleared when the glock is held, but there are no remaining holders. Used as part of the algorithm the calculates the minimum hold time for a glock.
r	Reply pending	Reply received from remote node is awaiting processing
y	Dirty	Data needs flushing to disk before releasing this glock

When a remote callback is received from a node that wants to get a lock in a mode that 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).

8.5. GLOCK HOLDERS

Table 8.5, “Glock holder flags” shows the meanings of the different glock holder flags.

Table 8.5. Glock holder flags

Flag	Name	Meaning
a	Async	Do not wait for glock result (will poll for result later)

Flag	Name	Meaning
A	Any	Any compatible lock mode is acceptable
c	No cache	When unlocked, demote DLM lock immediately
e	No expire	Ignore subsequent lock cancel requests
E	Exact	Must have exact lock mode
F	First	Set when holder is the first to be granted for this lock
H	Holder	Indicates that requested lock is granted
p	Priority	Enqueue holder at the head of the queue
t	Try	A "try" lock
T	Try 1CB	A "try" lock that sends a callback
W	Wait	Set while waiting for request to complete

The most important holder flags are H (holder) and W (wait) as mentioned earlier, since they are set on granted 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 that triggers the next state change. Since demote requests are always considered higher priority than requests from the file system, that might not always directly result in a change to the state requested.

The glock subsystem supports two kinds of "try" lock. 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 just what its name indicates; it is a "try" lock that does not do anything special. The T (**try 1CB**) lock, on the other hand, is identical to the t lock except that the DLM will send a single callback to current incompatible lock holders. One use of the T (**try 1CB**) lock is with the **iopen** locks, which are used to arbitrate among the nodes when an inode's **i_nlink** count is zero, and determine which of the nodes will be responsible for deallocating the inode. The **iopen** glock is normally held in the shared state, but when the **i_nlink** count becomes zero and →**evict_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 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 that 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 the ext3 file system3's orphan list in that it allows any subsequent reader of the bitmap to know that there is potentially space that might be reclaimed, and to attempt to reclaim it.

8.6. GLOCK TRACEPOINTS

The tracepoints are also designed to be 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 **gfs2_glock_state_change** tracepoint is the most important one to understand. It tracks every state change of the glock from initial creation right through to the final demotion which ends with **gfs2_glock_put** and the final NL to unlocked transition. The l (locked) glock flag is always set before a state change occurs and will not be cleared until after it has finished. There are never any granted holders (the H glock holder flag) during a state change. If there are any queued holders, they will always be in the W (waiting) state. When the state change is complete then the holders may be granted which is the final operation before the l glock flag is cleared.

The **gfs2_demote_rq** tracepoint keeps track of demote requests, both local and remote. Assuming that there is enough memory on the node, the local demote requests will rarely be seen, and most often they will be created by **umount** or by occasional memory reclaim. The number of remote demote requests is a measure of the contention between nodes for a particular inode or resource group.

The **gfs2_glock_lock_time** tracepoint provides information on the time taken by requests to the DLM. The blocking (**b**) flag was introduced into the glock specifically to be used in combination with this tracepoint.

When a holder is granted a lock, **gfs2_promote** is called, this occurs as the final stages of a state change or when a lock is requested which can be granted immediately due to the glock state already caching a lock of a suitable mode. If the holder is the first one to be granted for this glock, then the f (first) flag is set on that holder. This is currently used only by resource groups.

8.7. BMAP TRACEPOINTS

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

The **gfs2_bmap** tracepoint 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. This makes it easy to match the requests and results together and measure the time taken to map blocks in different parts of the file system, different file offsets, or even of different files. It is also possible to see what the average extent sizes being returned are in comparison to those being requested.

The **gfs2_rs** tracepoint traces block reservations as they are created, used, and destroyed in the block allocator.

To keep track of allocated blocks, **gfs2_block_alloc** is called not only on allocations, but also on freeing of blocks. Since the allocations are all referenced according to the inode for which the block is intended, this can be used to track which physical blocks belong to which files in a live file system. This is

particularly useful when combined with **blktrace**, which will show problematic I/O patterns that may then be referred back to the relevant inodes using the mapping gained by means this tracepoint.

Direct I/O (**iomap**) is an alternative cache policy which allows file data transfers to happen directly between disk and the user's buffer. This has benefits in situations where cache hit rate is expected to be low. Both **gfs2_iomap_start** and **gfs2_iomap_end** tracepoints trace these operations and can be used to keep track of mapping using Direct I/O, the positions on the file system of the Direct I/O along with the operation type.

8.8. LOG TRACEPOINTS

The tracepoints 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 debug journaling performance issues.

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

The **gfs2_ail_flush** tracepoint is similar to the **gfs2_log_flush** tracepoint in that it keeps track of the start and end of flushes of the AIL list. The AIL list contains buffers which have been through the log, but have not yet been written back in place and this is periodically flushed in order to release more log space for use by the file system, or when a process requests a **sync** or **fsync**.

8.9. GLOCK STATISTICS

GFS2 maintains statistics that can help track what is going on within the file system. This allows you to spot performance issues.

GFS2 maintains two counters:

- **dcount**, which counts the number of DLM operations requested. This shows how much data has gone into the mean/variance calculations.
- **qcount**, which counts the number of **syscall** level operations requested. Generally **qcount** will be equal to or greater than **dcount**.

In addition, GFS2 maintains three mean/variance pairs. The mean/variance pairs are smoothed exponential estimates and the algorithm used is the one used to calculate round trip times in network code.

The mean and variance pairs maintained in GFS2 are not scaled, but are in units of integer nanoseconds.

- srtt/srttvar: Smoothed round trip time for non-blocking operations
- srttb/srttvarb: Smoothed round trip time for blocking operations
- irtt/irttvar: Inter-request time (for example, time between DLM requests)

A non-blocking request is one which will complete right away, whatever the state of the DLM lock in question. That currently means any requests when (a) the current state of the lock is exclusive (b) the requested state is either null or unlocked or (c) the "try lock" flag is set. A blocking request covers all the other lock requests.

Larger times are better for IRTTs, whereas smaller times are better for the RTTs.

Statistics are kept in two **sysfs** files:

- The **glstats** file. This file is similar to the **glocks** file, except that it contains statistics, with one glock per line. The data is initialized from "per cpu" data for that glock type for which the glock is created (aside from counters, which are zeroed). This file may be very large.

- The **lkstats** file. This contains "per cpu" stats for each glock type. It contains one statistic per line, in which each column is a cpu core. There are eight lines per glock type, with types following on from each other.

8.10. REFERENCES

For more information about tracepoints and the GFS2 **glocks** file, see the following resources:

- For information on glock internal locking rules, see http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/filesystems/gfs2-glocks.txt;h=0494f78d87e40c225eb1dc1a1489acd891210761;hb=HEAD.

- For information on event tracing, see https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/trace/eve

- For information on the **trace-cmd** utility, see http://lwn.net/Articles/341902/.

CHAPTER 9. MONITORING AND ANALYZING GFS2 FILE SYSTEMS USING PERFORMANCE CO-PILOT (PCP)

This section provides information on using Performance Co-Pilot (PCP) to help with monitoring and to analyze GFS2 file systems. Monitoring of GFS2 file systems in PCP is provided by the GFS2 PMDA module in Red Hat Enterprise Linux which is available through the **pcp-pmda-gfs2** package.

The GFS2 PMDA provides a number of metrics given by the GFS2 statistics provided in the **debugfs** subsystem. When installed, the PMDA exposes values given in the **glocks**, **glstats**, and **sbstats** files. These report sets of statistics on each mounted GFS2 filesystem. The PMDA also makes use of the GFS2 kernel tracepoints exposed by the Kernel Function Tracer (**ftrace**).

9.1. INSTALLING THE GFS2 PDMA

In order to operate correctly, The GFS2 PMDA requires that the **debugfs** file system is mounted. If the **debugfs** file system is not mounted, run the following commands before installing the GFS2 PMDA:

```
# mkdir /sys/kernel/debug
# mount -t debugfs none /sys/kernel/debug
```

The GFS2 PMDA is not enabled as part of the default installation. In order to make use of GFS2 metric monitoring through PCP you must enable it after installation.

Run the following commands to install PCP and enable the GFS2 PMDA. Note that the PMDA install script must be run as root.

```
# yum install pcp pcp-pmdas-gfs2
# cd /var/lib/pcp/pmdas/gfs2
# ./Install
Updating the Performance Metrics Name Space (PMNS) ...
Terminate PMDA if already installed ...
Updating the PMCD control file, and notifying PMCD ...
Check gfs2 metrics have appeared ... 346 metrics and 255 values
```

9.2. DISPLAYING INFORMATION ABOUT THE AVAILABLE PERFORMANCE METRICS WITH THE PMINFO TOOL

The **pminfo** tool displays information about the available performance metrics. The following sections show examples of different GFS2 metrics you can display with this tool.

9.2.1. Examining the number of glock structures that currently exist per file system

The GFS2 glock metrics give insights to the number of glock structures currently incore for each mounted GFS2 file system and their locking states. In GFS2, a glock is a data structure that brings together the DLM and caching into a single state machine. Each glock has a 1:1 mapping with a single DLM lock and provides caching for the lock states so that repetitive operations carried out on a single node do not have to repeatedly call the DLM, reducing unnecessary network traffic.

The following **pminfo** command displays a list of the number of glocks per mounted GFS2 file system by their lock mode.

```
# pminfo -f gfs2.glocks
```

```
gfs2.glocks.total
    inst [0 or "afc_cluster:data"] value 43680
    inst [1 or "afc_cluster:bin"] value 2091

gfs2.glocks.shared
    inst [0 or "afc_cluster:data"] value 25
    inst [1 or "afc_cluster:bin"] value 25

gfs2.glocks.unlocked
    inst [0 or "afc_cluster:data"] value 43652
    inst [1 or "afc_cluster:bin"] value 2063

gfs2.glocks.deferred
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0

gfs2.glocks.exclusive
    inst [0 or "afc_cluster:data"] value 3
    inst [1 or "afc_cluster:bin"] value 3
```

9.2.2. Examining the number of glock structures that exist per file system by type

The GFS2 glstats metrics give counts of each type of glock which exist for each files ystem, a large number of these will normally be of either the inode (inode and metadata) or resource group (resource group metadata) type.

The following **pminfo** command displays a list of the number of each type of Glock per mounted GFS2 file system.

```
# pminfo -f gfs2.glstats

gfs2.glstats.total
    inst [0 or "afc_cluster:data"] value 43680
    inst [1 or "afc_cluster:bin"] value 2091

gfs2.glstats.trans
    inst [0 or "afc_cluster:data"] value 3
    inst [1 or "afc_cluster:bin"] value 3

gfs2.glstats.inode
    inst [0 or "afc_cluster:data"] value 17
    inst [1 or "afc_cluster:bin"] value 17

gfs2.glstats.rgrp
    inst [0 or "afc_cluster:data"] value 43642
    inst [1 or "afc_cluster:bin"] value 2053

gfs2.glstats.meta
    inst [0 or "afc_cluster:data"] value 1
    inst [1 or "afc_cluster:bin"] value 1

gfs2.glstats.iopen
    inst [0 or "afc_cluster:data"] value 16
    inst [1 or "afc_cluster:bin"] value 16
```

```
gfs2.glstats.flock
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0

gfs2.glstats.quota
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0

gfs2.glstats.journal
    inst [0 or "afc_cluster:data"] value 1
    inst [1 or "afc_cluster:bin"] value 1
```

9.2.3. Checking the number of glock structures that are in a wait state

The most important holder flags are H (holder: indicates that requested lock is granted) and W (wait: set while waiting for request to complete). These flags are set on granted lock requests and queued lock requests, respectively.

The following **pminfo** command displays a list of the number of glocks with the Wait (W) holder flag for each mounted GFS2 file system.

```
# pminfo -f gfs2.holders.flags.wait

gfs2.holders.flags.wait
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0
```

If you do see a number of waiting requests queued on a resource group lock there may be a number of reasons for this. One is that there are a large number of nodes compared to the number of resource groups in the file system. Another is that the file system may be very nearly full (requiring, on average, longer searches for free blocks). The situation in both cases can be improved by adding more storage and using the **gfs2_grow** command to expand the file system.

9.2.4. Checking file system operation latency using the kernel tracepoint based metrics

The GFS2 PMDA supports collecting of metrics from the GFS2 kernel tracepoints. By default the reading of these metrics is disabled. Activating these metrics turns on the GFS2 kernel tracepoints when the metrics are collected in order to populate the metric values. This could have a small effect on performance throughput when these Kernel Tracepoint metrics are enabled.

PCP provides the **pmstore** tool, which allows you to modify PMDA settings based on metric values. The **gfs2.control.*** metrics allow the toggling of GFS2 kernel tracepoints. The following example uses the **pmstore** command to enable all of the GFS2 kernel tracepoints.

```
# pmstore gfs2.control.tracepoints.all 1
gfs2.control.tracepoints.all old value=0 new value=1
```

When this command is run, the PMDA switches on all of the GFS2 tracepoints in the **debugfs** file system. Table 9.1, “Complete Metric List” explains each of the control tracepoints and their usage, An explanation on the effect of each control tracepoint and its available options is also available through the help switch in **pminfo**.

The GFS2 promote metrics count the number of promote requests on the file system. These requests are separated by the number of requests that have occurred on the first attempt and “others” which are granted after their initial promote request. A drop in the number of first time promotes with a rise in “other” promotes can indicate issues with file contention.

The GFS2 demote request metrics, like the promote request metrics, count the number of demote requests which occur on the file system. These, however, are also split between requests that have come from the current node and requests that have come from other nodes on the system. A large number of demote requests from remote nodes can indicate contention between two nodes for a given resource group.

The **pminfo** tool displays information about the available performance metrics. This procedure displays a list of the number of glocks with the Wait (W) holder flag for each mounted GFS2 file system. The following **pminfo** command displays a list of the number of glocks with the Wait (W) holder flag for each mounted GFS2 file system.

```
# pminfo -f gfs2.latency.grant.all gfs2.latency.demote.all

gfs2.latency.grant.all
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0

gfs2.latency.demote.all
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0
```

It is a good idea to determine the general values observed when the workload is running without issues to be able to notice changes in performance when these values differ from their normal range.

For example, you might notice a change in the number of promote requests waiting to complete rather than completing on first attempt, which the output from following command would allow you to determine.

```
# pminfo -f gfs2.latency.grant.all gfs2.latency.demote.all

gfs2.tracepoints.promote.other.null_lock
     inst [0 or "afc_cluster:data"] value 0
     inst [1 or "afc_cluster:bin"] value 0

gfs2.tracepoints.promote.other.concurrent_read
     inst [0 or "afc_cluster:data"] value 0
     inst [1 or "afc_cluster:bin"] value 0

gfs2.tracepoints.promote.other.concurrent_write
     inst [0 or "afc_cluster:data"] value 0
     inst [1 or "afc_cluster:bin"] value 0

gfs2.tracepoints.promote.other.protected_read
     inst [0 or "afc_cluster:data"] value 0
     inst [1 or "afc_cluster:bin"] value 0

gfs2.tracepoints.promote.other.protected_write
     inst [0 or "afc_cluster:data"] value 0
     inst [1 or "afc_cluster:bin"] value 0
```

```
gfs2.tracepoints.promote.other.exclusive
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0
```

The output from following command would allow you to determine a large increase in remote demote requests (especially if from other cluster nodes).

```
# pminfo -f gfs2.tracepoints.demote_rq.requested

gfs2.tracepoints.demote_rq.requested.remote
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0

gfs2.tracepoints.demote_rq.requested.local
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0
```

The output from the following command could indicate an unexplained increase in log flushes.

```
# pminfo -f gfs2.tracepoints.log_flush.total

gfs2.tracepoints.log_flush.total
    inst [0 or "afc_cluster:data"] value 0
    inst [1 or "afc_cluster:bin"] value 0
```

9.3. COMPLETE LISTING OF AVAILABLE METRICS FOR GFS2 IN PCP

Table 9.1, "Complete Metric List" describes the full list of performance metrics given by the **pcp-pmda-gfs2** package for GFS2 file systems.

Table 9.1. Complete Metric List

Metric Name	Description
gfs2.glocks.*	Metrics regarding the information collected from the glock stats file (**glocks**) which count the number of glocks in each state that currently exists for each GFS2 file system currently mounted on the system.
gfs2.glocks.flags.*	Range of metrics counting the number of glocks that exist with the given glocks flags
gfs2.holders.*	Metrics regarding the information collected from the glock stats file (**glocks**) which counts the number of glocks with holders in each lock state that currently exists for each GFS2 file system currently mounted on the system.
gfs2.holders.flags.*	Range of metrics counting the number of glocks holders with the given holder flags
gfs2.sbstats.*	Timing metrics regarding the information collected from the superblock stats file (**sbstats**) for each GFS2 file system currently mounted on the system.

Metric Name	Description
gfs2.glstats.*	Metrics regarding the information collected from the glock stats file (**glstats**) which count the number of each type of glock that currently exists for each GFS2 file system currently mounted on the system.
gfs2.latency.grant.*	A derived metric making use of the data from both the **gfs2_glock_queue** and **gfs2_glock_state_change** tracepoints to calculate an average latency in microseconds for glock grant requests to be completed for each mounted file system. This metric is useful for discovering potential slowdowns on the file system when the grant latency increases.
gfs2.latency.demote.*	A derived metric making use of the data from both the **gfs2_glock_state_change** and **gfs2_demote_rq** tracepoints to calculate an average latency in microseconds for glock demote requests to be completed for each mounted file system. This metric is useful for discovering potential slowdowns on the file system when the demote latency increases.
gfs2.latency.queue.*	A derived metric making use of the data from the **gfs2_glock_queue** tracepoint to calculate an average latency in microseconds for glock queue requests to be completed for each mounted file system.
gfs2.worst_glock.*	A derived metric making use of the data from the **gfs2_glock_lock_time** tracepoint to calculate a perceived "current worst glock" for each mounted file system. This metric is useful for discovering potential lock contention and file system slowdown if the same lock is suggested multiple times.
gfs2.tracepoints.*	Metrics regarding the output from the GFS2 **debugfs** tracepoints for each file system currently mounted on the system. Each sub-type of these metrics (one of each GFS2 tracepoint) can be individually controlled whether on or off using the control metrics.
gfs2.control.*	Configuration metrics which are used to switch on or off metric recording in the PMDA. Conrol metricsare toggled by means of the **pmstore** tool.

9.4. PERFORMING MINIMAL PCP SETUP TO GATHER FILE SYSTEM DATA

The following procedure outlines instructions on how to install a minimal PCP setup to collect statistics on Red Hat Enterprise Linux. This setup involves adding the minimum number of packages on a production system needed to gather data for further analysis.

The resulting **tar.gz** archive of the **pmlogger** output can be analyzed by using further PCP tools and can be compared with other sources of performance information.

1. Install the required PCP packages.

```
# yum install pcp pcp-pmdas-gfs2
```

2. Activate the GFS2 module for PCP.
   # **cd /var/lib/pcp/pmdas/gfs2** # **./Install**

3. Start both the **pmcd** and **pmlogger** services.
   # **systemctl start pmcd.service** # **systemctl start pmlogger.service**

4. Perform operations on the GFS2 file system.

5. Stop both the **pmcd** and **pmlogger** services.
   # **systemctl stop pmcd.service** # **systemctl stop pmlogger.service**

6. Collect the output and save it to a **tar.gz** file named based on the host name and the current date and time.
   # **cd /var/log/pcp/pmlogger** # **tar -czf $(hostname).$(date+%F-%Hh%M).pcp.tar.gz $(hostname)**

9.5. REFERENCES

- For more information on performance monitoring on GFS2 in general, see Debugging GFS2 file systems with GFS2 tracepoints and the debugfs glocks file.
- For more general information on using Performance Co-Pilot to monitor system performance see Monitoring performance with Performance Co-Pilot on the Red Hat Customer Portal.
- For general information about Performance Co-Pilot in Red Hat Enterprise Linux see Index of Performance Co-Pilot(PCP) articles, solutions, tutorials and white papers on the Red Hat Customer Portal.