HPSSFS-FUSE Administrator's Guide

Michael Theall
<mtheall@us.ibm.com>
version 600, 2014-11-12

Table of Contents

1. Overview

The High Performance Storage System™ File System FUSE (HPSSFS-FUSE) interface provides users with a standard POSIX® filesystem view of HPSS™ files. Filesystem in Userspace (FUSE) is a mechanism that allows virtual filesystems to be implemented in userspace.

The HPSSFS-FUSE interface is supported only on Red Hat® Enterprise Linux® (RHEL®) [1] It

enables HPSS to function as an additional supported filesystem type for Linux users. It allows users to access HPSS-resident files with standard POSIX semantics employed by local Linux filesystems, such as ext3 and network filesystems such as NFS. Linux users can mount an HPSS directory, traverse the directory structure, and access files as though operating on a local Linux filesystem. Access is achieved by means of POSIX function calls, such as `open(2)`, `read(2)`, `write(2)`, and UNIX® commands such as `cp(1)`. Like NFS, HPSSFS-FUSE does not require local storage resources, but is rather a convenient interface to HPSS.

The HPSSFS-FUSE interface enables existing software to access HPSS files without modification. For example, agent software, such as SAMBA™, Secure FTP, Apache®, and even native Linux NFS may be set up to access HPSS files using the HPSSFS-FUSE interface. Thus, the HPSSFS-FUSE interface becomes a means to utilize a wide variety of agents for local and remote network-connected users. Multiple agents may be employed, and even multiple instances of the same agent. For example, a site may employ several agent computers providing NFS and several others providing SAMBA. However, in most situations, the use of multiple agent computers will not be necessary.

Thus, the HPSSFS-FUSE interface serves as a high-performance, virtually-local interface for trusted Linux client nodes, such as those in a high-performance computational cluster. At the same time, it can serve as a convenient means of extending HPSS access to users outside of the main cluster, with security performed by agent software.

The HPSSFS-FUSE interface does not change the nature of the underlying HPSS hierarchical storage management software. Most HPSS sites are set up to migrate less recently used files to tape. Although the HPSSFS-FUSE interface does employ local caching and readahead logic to enhance performance, the overall operational concept for the system must take into account the latency of accessing files from tape. On the other hand, HPSS offers optional SAN enablement, referred to in HPSS documentation as HPSS 3rd Party SAN (SAN3P). SAN3P enables data to move between clients and HPSS disk without passing through an intermediate computer, but under the control of HPSS. SAN3P therefore can provide significant throughput advantages for sequential transfer of data between clients and HPSS disk.

This document is intended to provide administrators and sophisticated ("power") users with information on the installation, tuning, and use of HPSSFS-FUSE. Limitations as well as features of the HPSSFS-FUSE interface are explained so that existing best practices can be employed and new best practices discovered.

2. Availability

HPSSFS-FUSE is available as a separate package from HPSS. It can be obtained from your HPSS support representative.

2.1. Prerequisites

HPSSFS-FUSE requires the following software:

- libfuse >= 2.8.3
- Linux kernel >= 2.6.31 with FUSE kernel module
- HPSS 7.3.3 or HPSS >= 7.4.1 [2]

3. Concepts

At a high level, the HPSSFS-FUSE interface is very simple and straightforward. Almost all POSIX-based operations one can perform on a Linux filesystem can also be executed on an HPSSFS-FUSE-mounted filesystem. Even so, there are some exceptions. This section covers characteristics about HPSSFS-FUSE that should be understood by those considering its use in their environment.

3.1. HPSS and the Nature of Hierarchical Storage

While readers of this document are presumed to be familiar with HPSS, we will review here some HPSS concepts that are particularly relevant to the HPSSFS-FUSE interface. HPSS is a hierarchical storage management (HSM) software designed to manage and access petabytes of data at high data rates. HPSS is most cost effective for archives larger than 10PB. While appearing to the user as a disk filesystem, HPSS manages the life cycle of data by moving inactive data to tape and retrieving it the next time it is referenced.

HPSS is a distributed solution with file attributes stored on the Core Server, data stored on Mover systems, and HPSSFS-FUSE applications running on client nodes, among other client interfaces. The cluster aspect of HPSS combines the power of multiple computer nodes into a single, integrated storage system. The computers that comprise the HPSS platform may be of different makes and models, yet the storage system appears to its clients as a single storage service with a unified common name space.

When users access HPSS via the HPSSFS-FUSE interface or one of the other HPSS interfaces such as FTP or the Client API, they are presented with a UNIX-like filesystem view of their data. In addition to files, HPSS supports directories, symbolic and hard links, and attributes compatible with any modern UNIX filesystem. Unlike a conventional disk-based filesystem, however, HPSS must deal with the latency of accessing data on both disk and tape. Access to data may be delayed as tapes must be mounted and queued with other tape drive/library activities. Data is stored sequentially on tape media, which is different from disk-based storage that provides random access to data. While interfaces may provide conveniences and hide certain aspects of this behavior, fundamentally, the system is an HSM and those using it must understand the qualities and limitations of an HSM.

Linux provides filesystem caches for file attributes and data blocks to improve performance by temporarily storing requested information in kernel buffers. This improves performance for multiple requests of the same information or, in the case of readahead logic, sequential access to file data. The expense is a weak coherency between multiple clients and mount points. Performance gains are dependent upon configuring memory resources based on number of threads, concurrent file access, file sizes, and available system memory.

Like with other HPSS client interfaces, the configuration of HPSS is critical for optimal performance. Configuring HPSS with correct Storage Classes, Hierarchies, Classes of Service, filesets, junctions, and providing appropriate mount points will determine performance through the HPSSFS-FUSE interface. Generally, different Classes of Service are created in HPSS to balance performance with different storage media characteristics and different file sizes. Applications must understand where in the HPSS hierarchy files in different Classes of Service are located and expect different performance. For instance, files that have to be staged from tape levels of a hierarchy will encounter more latency compared to files at a disk level. Storage efficiency within HPSS is determined by Storage Class segment size and the typical file sizes stored. It is also determined by the heirarchy makeup. Applications must consider this when

storing files using the HPSSFS-FUSE interface.

So, how does this affect end users? One example is using the common UNIX command `grep(1)`. On a local Linux filesystem, such a command can be invoked recursively on a directory tree with little to no concern. However, when used on an HPSSFS-FUSE mount, such a command is unaware of the current state of the files' contents and will proceed to stage any file that is not stored in HPSS disk cache. If a user's command searches through hundreds or thousands of files that must be staged from tape, not only will it potentially take a very long time for the command to complete for the user, but there can be a detrimental effect on other users of the HPSS system as tape drives are kept busy servicing this one command. That is why it is important for administrators and end users to understand how HPSSFS-FUSE works. Remember the following when planning to use or introduce HPSSFS-FUSE to a site environment:

- File attributes are readily available, but file contents may not be.
- HPSSFS-FUSE has the look and feel of a filesystem, but it is really an interface to HPSS, which is an HSM.

3.2. Architecture

HPSSFS-FUSE is glueware that sits between HPSS and FUSE. It uses FUSE to represent HPSS as a virtual filesystem.

Figure 1 shows the Linux client software. This software is separated into userspace and kernel. The userspace is shown containing three types of client applications: a user application, a user shell such as `ksh(1)` or `bash(1)`, and agent software such as an NFS or SAMBA server. The VFS is the Linux Virtual Filesystem Switch, which forwards filesystem access to the appropriate filesystem drivers, such as FUSE, NFS, and ext3. HPSSFS-FUSE retrieves filesystem requests from the FUSE kernel module via libfuse, and forwards them to HPSS using the HPSS Client API.

The HPSS Client API is both a user interface in its own right and a building block from which other interfaces are created. The HPSS Client API supports the separation of command and data paths. The command path is usually a TCP/IP path, and the data path may be a separate TCP/IP data path, or it may be implemented as a SAN such as fibre channel. HPSS documentation refers to the SAN option as SAN3P (SAN 3rd Party), where the client application is the 3rd party performing the SAN I/O. HPSSFS-FUSE is able to use SAN3P transfers in order to take advantage of the direct client-to-disk data transfer mechanism.

SAN3P

There is a security vulnerability associated with the use of SAN3P. If a user is root on a machine which has access to the SAN (e.g. a client machine), then that user has the potential to access or destroy fiber-channel connected disk storage. Two areas of concern:

1. Verification that only authorized users (usually limited to only root or hpss) are granted read and write access to these resources.
2. HPSS administrators should be aware that machines, possibly owned or managed by other groups, which are added to the SAN to facilitate the use of SAN3P transfers will have access to all data on disk and tape resources. If those systems are compromised, or if there are

> individuals authorized for system privileges on those particular machines, but not necessarily authorized for HPSS administrative access, there is the potential for access and/or damage to HPSS data. These are inherent limitations of SAN implementations that have not yet been addressed by the industry and cannot be remedied by HPSS.

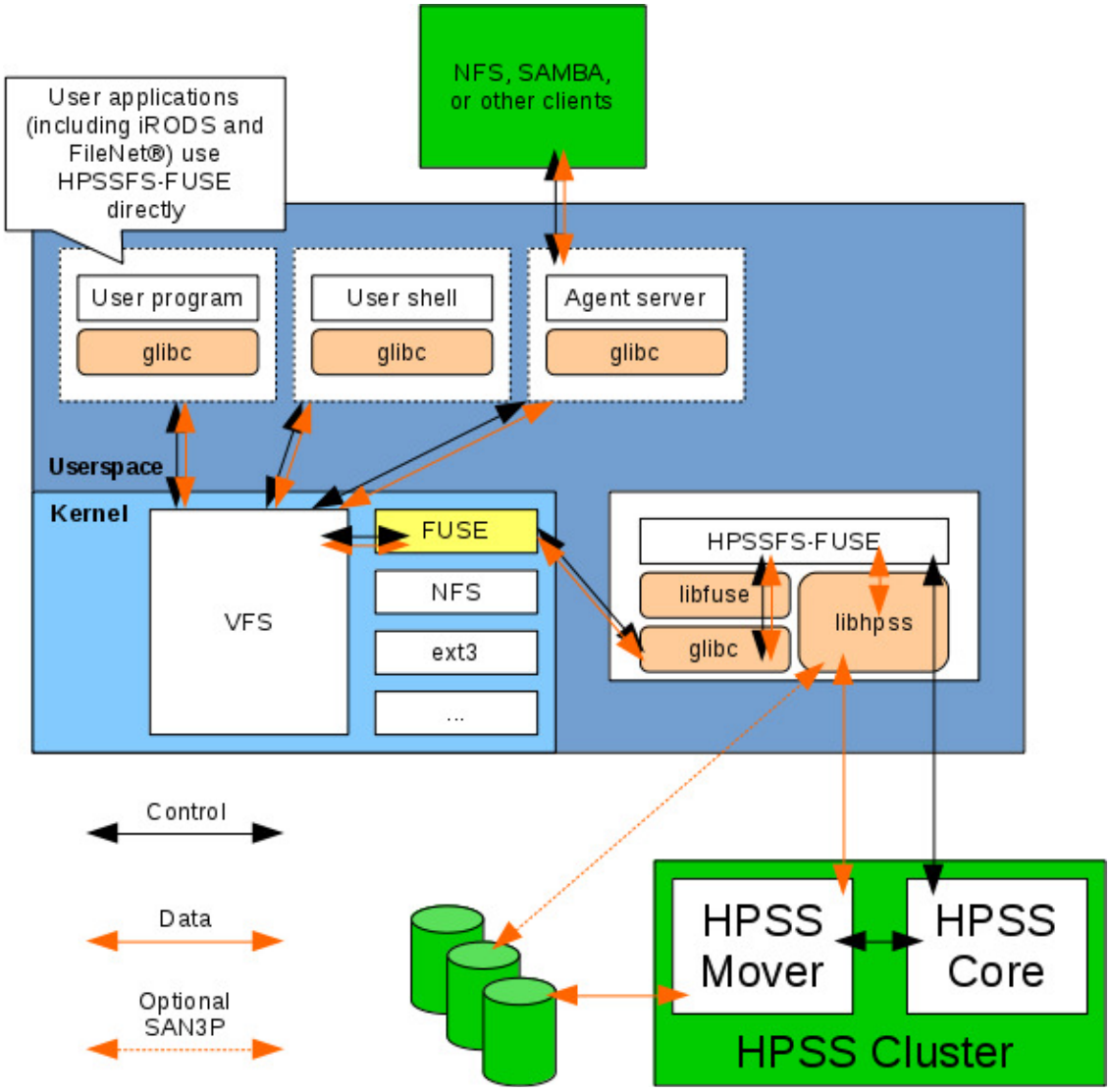

Figure 1. HPSSFS-FUSE Components

Based on experience in the field, we recommend that separate HPSSFS-FUSE mount points exist for each major application that resides on top of it. When used in a gateway configuration using agent software, there is no requirement for separate mount points, but for performance or load-balancing it may be necessary. The separate mount points allow for easier control and troubleshooting of the system.

3.3. How It Works

Here is an example of what happens when a user tries to open a file:

1. Application issues an `open(2)` call on a file.

2. The Linux VFS provides common filesystem functionality, then passes control to the FUSE kernel module.
3. libfuse retrieves the request from the FUSE kernel module, and calls a callback function in HPSSFS-FUSE to service the request.
4. HPSSFS-FUSE uses the HPSS Client API to open the file.
5. The HPSS Core Server performs the file open. If permissions, path, and attributes are valid, the file is opened.
6. The HPSS Client API receives a response from the HPSS Core Server indicating success or failure. This status is returned to HPSSFS-FUSE.
7. HPSSFS-FUSE replies to the FUSE kernel module via libfuse.
8. The FUSE kernel module returns the information back to the Linux VFS.
9. The Linux VFS returns the system call.
10. Application receives status from the system call and acts accordingly.

3.4. Supported Functionality and Limitations

Mount Options

The following references HPSSFS-FUSE mount options.

- Most HSM users access file data in sequential order. The HPSSFS-FUSE interface implements a sequential readahead algorithm to increase the probability that the next requested read will be in the HPSSFS-FUSE buffer cache. The following should be understood about this algorithm:
  - For performance reasons, when files are read sequentially, HPSSFS-FUSE will read the next sequential portion of a file before an application requests it. This helps reduce the read latency to the application. The size of the portion ranges from 128KB up to *stream* megabytes. It starts out at 128KB, and then it doubles for each successive read, with the maximum readahead window of *stream* megabytes. If the read requests are not sequential, the readahead is not performed. If the application read requests do not read the entire readahead buffer, the readahead buffer size will remain the same.
  - The default *stream* value is 8MB. This means the readahead algorithm will consume 8MB for every open file. The system RAM should be sized for the maximum readahead buffersize multiplied by the number of concurrent files being read.
  - For maximizing performance, the application should issue sequential read requests that are equal to the maximum readahead buffer size.
  - Files that are opened with `O_SYNC` or `O_DIRECT` will not use buffered I/O, and therefore will not use the readahead algorithm.
  - Using a *stream* option with the value 0 will cause all open files to use unbuffered I/O, and therefore will not use the readahead algorithm.
  - A writeback algorithm is in place similar to the readahead algorithm. It shares the same buffer with the readahead algorithm.
- The HPSS maximum for 10,000 storage segments applies. When storing a file using

HPSSFS-FUSE, use a Storage Class that supports a storage segments size that can accommodate the intended size of the file. The Storage Class used is dependent on the mount option *cos=ID* and/or the fileset where the file is being stored. An additional consideration is the mount option *maxsegsz*.

- The HPSS maximum for 2,000 fragments applies. Fragments are sections of data separated by a hole where an application has not written data. Using the `lseek(2)` system call, an application can skip around in a file to write data at various offsets. HPSS does not initialize or store data for these holes; metadata is maintained to identify the holes. When a file reads at an offset that is a hole, the data values are binary zero.
- The Linux `df(1)` command statistics represent the entire Class of Service (COS) statistics. The sum of all Storage Classes in the COS Hierarchy is reported. The reported free space may not represent the amount of space that can be written, especially when there are multiple levels in the Hierarchy. A mount point may not even show up in the `df(1)` listing if the total storage for its COS is 0 (e.g. a dummy default COS).
- Security: There are no restrictions from the Core Server on which nodes can connect via HPSSFS-FUSE. Any node that can install the HPSS Client API can access HPSS.
    - HPSS provides a restricted user capability for blacklisting users based on User ID from connecting to the system. This only affects which users can be used as the principal for login credentials, so blacklisted users may still use HPSSFS-FUSE when using the hpssfs principal. See "Restricting user access to HPSS" in the HPSS Management Guide for more information.
    - Keytabs are commonly used to facilitate establishment of HPSS credentials. It is recommended to use a keytab for the *hpssfs* principal for use by HPSSFS-FUSE. This keytab should be protected to prevent unauthorized access by unprivileged users.
- ACL's: HPSSFS-FUSE does not support ACL's.
- FIFO's and other special devices: HPSS and therefore HPSSFS-FUSE does not support named pipes (FIFO's), character device files, and block devices; use a local filesystem for these purposes.
- Kernel caching and data buffering: The Linux kernel caches directory and file attributes. This may prevent retrieving up-to-date attributes from HPSS that are updated by other HPSS clients (including other HPSSFS-FUSE mounts on the same machine). Different clients may receive different information based on what is cached and when changes are made. The benefits of caching attributes and buffering data are to minimize latency to the application by not waiting to retrieve data from HPSS. Direct I/O can be used to bypass the data buffer cache, but every read and write will require transferring data from HPSS. The attribute cache timeout is controlled by the *attrtimeo* mount option. The name cache timeout is controlled by the *entrytimeo* mount option. These caches can be disabled by setting their values to 0. This will increase coherency at the expense of performance.
- The caching mechanisms help reduce latencies, but cause a weak coherency concerning external applications.
- The data buffering mechanisms help increase throughput, but at the expense of reduced coherency. For transaction-sensitive applications where data written to HPSS using the HPSSFS-FUSE interface requires guaranteed updates, the program must do one or more of the following:
    - Rely upon `fsync(2)` to flush data buffers to HPSS.
    - Open a file with the `O_SYNC` or `O_DIRECT` flags to flush data on every write.

  - Rely on the return value from the `close(2)` function as indication of successfully flushed data.

  Otherwise, a successful return code from the `write(2)` system call is not a guarantee that all data has been completely flushed to HPSS at that point in time. The application programmer must balance the performance advantages of buffering versus the requirements for data synchronization between their application and HPSS. This behavior is consistent with the POSIX standard, and true of both local storage resources (e.g. disk partitions) as well as remote storage such as HPSSFS-FUSE and NFS.

- The HPSSFS-FUSE Gateway is essentially a "store and forward" machine that should be taken into consideration when sizing any Linux gateway computers.
- HPSSFS-FUSE supports a number of extensions to the POSIX library interface to enable users to control specific HPSS attributes, such as setting the Class of Service (COS) value. The list of extensions and how to use them is documented in the Extensions section.

4. Tuning & Troubleshooting

Like most systems, HPSSFS-FUSE will require tuning to allow users/applications to perform optimally. The underlying HPSS configuration, network topology, and client systems can affect performance and the operation of the system. This section covers the major tuning components of HPSSFS-FUSE, what to look for when analyzing the performance of the system, and what troubleshooting resources and procedures are available for the administrator to use in diagnosing problems.

4.1. Expectations

Administrators and users should expect HPSSFS-FUSE to perform similarly to the HPSS Client API. In some cases the performance may be better because of the kernel caching (namespace attributes and file data), but in general the transaction and transfer performance will be in-line with HPSS Client API because HPSSFS-FUSE uses the API for its interaction with HPSS. Therefore, it is important to ensure that performance as measured by tools, such as the API Example code, are consistent with baseline numbers documented during the deployment of the system. The HPSS Test Plan and Results report or other similar testing should be reviewed and compared with results measured against the current system. If the performance of the HPSS Client API on the HPSSFS-FUSE machine is not up to expected rates, then correcting those deficiencies should be addressed before focusing on HPSSFS-FUSE performance.

4.2. Testing Procedures

During the initial deployment of an HPSS system, the support representative conducts a number of functional and performance tests on the system. These tests include procedures for checking the client interfaces to be used at a given site, including HPSSFS-FUSE, if configured at the time of the installation. The results from these tests are used as a baseline for comparing performance of the system when changes are made to HPSS or the client environment, or when troubleshooting a performance problem.

The first task is to repeat those same HPSSFS-FUSE tests to compare against the baseline results. A high-level summary of some tests that might be exercised are outlined below:

- Directory listing of namespace.

    - `ls(1)`
    - `find(1)`
- Simple file/directory operations.
    - `mkdir(1)`
    - `rmdir(1)`
    - `touch(1)`
    - `unlink(1)`
    - `mv(1)`
    - `ln(1)`
    - `cd(1)`
- Copy multiple groups of files into and out of HPSSFS-FUSE.
- Rerun the HPSSFS-FUSE performance tests to obtain new baseline results.
- Use a script to `touch(1)` numerous files in a directory, then perform an `rm -rf *` [3] at the directory level to delete all the files created.
- Use a script to exercise HPSSFS-FUSE for an extended period (24-48 hours). This can be as simple as copying files into the HPSSFS-FUSE mount point. Multiple copy operations should be performed from a single script, and if possible, multiple clients should be used.
- Perform `tar(1)` and `gzip(1)` on files located in the HPSSFS-FUSE mount point.
- Perform `dd(1)` into and out of the HPSSFS-FUSE mount point.
- Use a basic C program which creates, opens, writes, and closes files.
- Use a basic C program which reads the previously created files. If possible, read migrated/purged files (files on tape with no copy in the HPSS disk cache), to monitor how HPSSFS-FUSE handles staging.

4.3. Tuning Concepts

4.3.1. What are we tuning?

How one plans to use HPSSFS-FUSE is key to what should be done to tune the system. Is the usage primarily oriented to access the namespace and file attributes? Is the goal to optimize data I/O? What file sizes are expected? Are there a few users or many? How is load balanced? These and other questions need to be considered before starting the tuning process. If there are divergent requirements, then multiple HPSSFS-FUSE mounts may be necessary to optimize a particular access pattern.

Consider making the adjustments only when necessary. Likely, it will take some experimentation to get the right set of options. If usage conditions or requirements change, tuning options may need to be reevaluated and adjusted.

4.3.2. Configuring for efficient HPSS storage

HPSS stores portions of a file in what are called storage segments. Since each storage segment has to be tracked, there is metadata created for each storage segment. To prevent individual files from monopolizing HPSS metadata space, there is a maximum number of segments that

HPSS will allow for each file (10,000 is the maximum). Another important aspect is if the amount of data written to a storage segment is less than the storage segment size, the remaining space cannot be used for anything else (it is wasted space). The size of a storage segment is determined by the Class of Service (COS) being used and whether the mount option maxsegsz is specified.

To help with usage patterns, HPSSFS-FUSE allows you to configure mount points for a specific COS or to use the maximum segment size. By specifying a specific COS for a mount point, you can have some control over the segment size allocation and which Storage Class will be used when an application creates a file. The exception to this rule is if the file is created in a fileset. In that case, the COS set for the fileset will be used instead of the mount option COS if it is not set to NONE. The COS has an "Allocation Method" where you can choose either *Fixed*, *Maximum*, or *Variable*. Using the correct allocation method will determine how efficiently HPSS stores a file.

- *Fixed* usually will default to the minimum segment size for the Storage Class. This is most efficient when the file sizes are typically less than or very near to the Storage Class minimum segment size. It is least efficient when the file sizes are typically many multiples of the minimum segment size and the difference between minimum segment size and maximum segment size is large.
- *Maximum* will default to the maximum segment size for the Storage Class. This is most efficient when the file sizes are typically close to or greater than the maximum segment size. It is least efficient when the file sizes are typically very small because the maximum segment size will be allocated and only a very small part of the segment will be used.
- *Variable* allows for a progression of larger segments for each segment. This method is often referred to as Variable Length Segment Size (VLSS). It was introduced to help when the file sizes vary greatly and the difference between the minimum and maximum segment sizes for a Storage Class is large. With each successive storage segment allocated being double the size of the previous (up to the maximum segment size), the efficiency is greatly improved. There are fewer segments (minimizing the metadata overhead) and less wasted space (versus the *Maximum* allocation method), which allows much larger files than using the *Fixed* allocation method. To minimize the unused space in the last segment of the *Variable* allocation method, the segment size is reduced to the smallest multiple of the minimum segment size.

The top level Storage Class definition determines the actual minimum and maximum segment sizes to be used. Configuring the mount point to a COS which uses a Storage Class that is appropriate based on the sizes of the files to be created will greatly influence the HPSS efficiency. The Storage Class will also greatly influence the allowable sizes of files that can be stored. As indicated above, the Fixed allocation method will use the Storage Class minimum segment size. This will limit the maximum file size to be the Storage Class minimum segment size multiplied by the maximum number of Bitfile segments that HPSS can support. HPSSFS-FUSE does support an override of using the Fixed allocation method minimum segment size, however the override is to use the Storage Class maximum segment size (from one extreme to the other).

HPSSFS-FUSE does allow an application to override the mount point specification for a COS. The caveat is an extra system call has to be made to HPSSFS-FUSE by the application to accomplish this. A limitation of using standard Linux applications (e.g. `cp(1)` command) is they do not support setting the COS explicitly. Because of this, it is critical to understand application file creation patterns and setting up COS and Storage Class that support the applications. It may be necessary for multiple mount points to be used to get the COS and Storage Class combinations correct for different application usage patterns. For this reason, it is

sometimes best to use multiple HPSSFS-FUSE mounts to provide different optimization options to the same HPSS namespace.

See "Storage Configuration" in the HPSS Management Guide for more information.

4.4. Troubleshooting

There are several sources of information available for the administrator to look at when troubleshooting an HPSSFS-FUSE problem. The following section documents where this information is stored and what can be done to monitor and control the level of output.

Client API

Keep in mind the following about HPSSFS-FUSE: it is built upon the HPSS Client API. If there are basic communication problems or performance issues with the HPSS Client API, there is little point to troubleshooting HPSSFS-FUSE itself. It is recommended that the administrator perform a set of basic operations using scrub or the API example programs to verify the function and performance of the system. There may very well be problems with HPSSFS-FUSE in the end, but troubleshooting the operating system and HPSSFS-FUSE prerequisites commonly saves a lot of time and effort.

Because HPSSFS-FUSE is built upon the HPSS Client API, it is useful to set the API debug/logging environment variables (`/var/hpss/etc/env.conf`):

- `HPSS_API_DEBUG=<level>`
- `HPSS_API_DEBUG_PATH=<stderr|/path/file>`

HPSS_API_DEBUG

The HPSS_API_DEBUG value can be increased up to 7 to produce output that is more detailed. HPSSFS-FUSE will need to be restarted for the environment variables to take effect, meaning the mount points will have to be remounted.

See "Tuning and Troubleshooting" in the HPSS Programmer's Reference for more information.

4.4.1. Syslog

The most important resource for monitoring HPSSFS-FUSE mount points is the Linux syslog. Linux system error and diagnostic messages are logged to `/var/log/messages`. This file is only directly readable by root; any non-privileged user can view it using the `dmesg(1)` command. When this file grows larger than some configured size (see `logrotate(8)`), it is rotated to a file name that is post-fixed with an integer value that indicates its relative age.

HPSSFS-FUSE has a number of logging message classes. These include ERROR and 5 TRACE levels. The trace class messages must be enabled in order to appear in the syslog. The trace level is controlled by a mount option and at runtime via the system.hpssfs.trace xattr. The ERROR class is intended to indicate a potentially disastrous error. The TRACE class is intended to give

increased level of detail for diagnosing issues, and should be set to 0 except when directed otherwise by HPSS support.

4.4.2. Foreground Logging

If the -f mount option is used, HPSSFS-FUSE will run in the foreground. All HPSSFS-FUSE ERROR and TRACE messages will be printed to stderr instead of the syslog in this case. This is mainly useful for when a developer needs to assist in diagnostics.

4.4.3. Specialized Logging

The support team may provide a special HPSSFS-FUSE build at the developer's discretion that has extremely detailed log messages for specific subsystems in HPSSFS-FUSE in order to facilitate diagnostics. Please do not use these builds in a production environment once diagnostics are complete.

4.4.4. HPSS Logs and Alarm & Events Display

One reason for insisting that all HPSS servers and client machines be time-synced is to help the administrator determine what HPSS errors, as reported in the main HPSS error logging facility, correspond to problems logged on the client machines. By matching the date and timestamps, HPSSFS-FUSE errors such as a write -5, the ambiguous "something went wrong" I/O error, can further be analyzed on the HPSS server side. Such analysis can help determine if the error is network-related, maybe a sporadic outage between the HPSSFS-FUSE client and HPSS, or maybe a tape has a permanent error and the user's HPSSFS-FUSE request simply cannot be fulfilled.

If there doesn't seem to be any corresponding information in the HPSS logs, it may be advantageous to repeat the user request on another HPSSFS-FUSE client, or even use another HPSS interface such as PFTP to help isolate what part of the overall system is not working correctly or performing poorly.

4.4.5. Core Dumps

Core dumps should be enabled in case HPSSFS-FUSE happens to crash. If this occurs, please send the core dump to your support representative.

If using `abrtd(8)`, it may be useful to adjust `/etc/abrt/abrt.conf` in order for it to generate a full crash report.

- `MaxCrashReportsSize` — may need to increase or set to unlimited.
- `OpenGPGCheck = no` — if you have installed an unsigned HPSSFS-FUSE package.
- `ProcessUnpackaged = yes` — if you have installed HPSSFS-FUSE from source.

4.4.6. Force Unmount

Due to exceptional circumstances, it may be necessary to perform a force unmount to unmount an HPSSFS-FUSE mount point. This can be achieved with the *-f* flag in the `umount(8)` command:

```
$ umount -f /mnt/hpss
```

In rare situations, this may be insufficient. It may be necessary to issue an abort through FUSE's debugfs interface.

```
$ grep "/mnt/hpss" /proc/self/mountinfo | cut -d' ' -f3 | cut -d':' -f2
47
$ echo 1 > /sys/fs/fuse/connections/47/abort
$ umount /mnt/hpss
```

5. Unprivileged Mounts

FUSE allows unprivileged mounts. This means mounts performed by unprivileged users. It achieves this by having a helper set-uid program `fusermount(1)` perform mounts for FUSE filesystems. On some systems, the default permissions only allow users in the group fuse to execute this program. This is recommended to isolate unprivileged mounts to trusted users only.

Although this allows unprivileged users to mount HPSSFS-FUSE, they must still provide valid HPSS credentials for the mount to succeed. Only a principal which has the Core Server Control ACL (such as hpssfs) can perform operations on behalf of other users, so unprivileged mounts should be limited to principals which do not have the Core Server Control ACL. It is recommended not to use the allow_other mount option on unprivileged mounts because without the Core Server Control ACL, all operations will be performed on behalf of the principal used for the mount. Furthermore, it is recommended that unprivileged mounts perform the mount as the user which is supplied as the principal, otherwise FUSE may prevent access to your files due to the uid mismatch.

SAN3P

SAN3P transfers may not work with unprivileged mounts since they require read-write access to the SAN devices.

Checksum

The checksum feature may not work with unprivileged mounts since it requires read-write access to HPSS's root directory and to the files being opened.

6. Uses

The HPSSFS-FUSE interface provides users with the ability to use commonly available file transport mechanisms. This simplifies the use of HPSS by allowing users to access HPSS via interfaces they are familiar utilizing. This section covers some of these applications, their use, hints at how they might be configured for use with HPSSFS-FUSE, describes any known limitations or changes required, and recommendations or lessons learned from field experience.

6.1. General

6.1.1. Overview

If you have not read Concepts, you need to review it and have a good understanding about the

differences between a filesystem (i.e. LFS, GPFS, etc) and an HSM (HPSS). It must be stressed that HPSSFS-FUSE looks like a filesystem, but it is an interface to HPSS, which is an HSM. Those differences can have a significant impact to applications that expect 100% compatibility with a true filesystem. Users who run large programs successfully on a shared filesystem like GPFS, may run into issues with their application when files are not immediately available (e.g. must be staged from tape) or where too many simultaneous open files, small block, or random I/O operations are occurring. HPSSFS-FUSE is a convenience for accessing HPSS, but it will not hide the realities of the storage system behind it.

6.1.2. Applications

The HPSS team supports the HPSSFS-FUSE interface and will assist administrators (based on the contract or SOW that exists with a site) with its use. However, HPSS does not provide support for applications that reside on top of HPSSFS-FUSE. Several applications are mentioned in this section including the popular SAMBA interface that provides file sharing across a number of different operating systems. Many sites have been able to successfully use SAMBA and other tools with HPSSFS-FUSE. Even so, the HPSS team itself does not provide support for installing, configuring, or maintaining 3rd party applications. Before sites use these applications, they must be prepared to support themselves or obtain support from other sources. If there are problems using one of the applications, and it can be shown that the underlying problem is because HPSSFS-FUSE is mishandling an operation, HPSS support will submit a bug report to development and look for ways to address the issue. It is imperative that the administrator provide as much detail as possible when reporting a problem and have performed due-diligence in ensuring the problem is not with the application or how the end user is using the application.

6.1.3. End-User Access to HPSSFS-FUSE

If there are to be end users directly accessing HPSSFS-FUSE who are not necessarily aware of HPSS and its HSM characteristics, it is suggested that certain UNIX commands that recursively perform name-space operations on files be aliased with scripts or programs to test what filesystems they are accessing. In the case of `grep(1)` or `fgrep(1)`, a warning or limitation should be in place to ensure that users don't accidentally search for a string in files and induce a large number of file stages from tape as the command recursively navigates the directory tree. It is likely impossible to prevent all such possible accidents by users, and certainly in no way will prevent intentional misuse of the system, but such precautions will quickly pay for the extra up-front effort by redirecting common filesystem commands that aren't necessarily "HSM friendly".

`cp(1)` and `mv(1)` Commands

The `cp(1)` and `mv(1)` commands from coreutils, by default, attempt to optimize I/O by skipping parts of a file that are heuristically determined to be sparse, i.e. contain large sequences of zeros. If a sparse section of a file is encountered while reading, the corresponding part of the destination file is skipped (using `lseek(2)`), and writing is resumed where the chunk of zeros ends. This has the potential to significantly reduce the amount of writing performed.

In the case that the destination file is in HPSSFS-FUSE, sparse files tend to produce issues. When writing to a file, skipping over a section (using `lseek(2)`) and then writing causes a new Bitfile segment to be created (it would otherwise extend the current Bitfile segment). If this is done frequently, you may eventually run into an HPSS limit on the number of Bitfile segments. If this happens, then no additional Bitfile segments may be created. Therefore, it is

recommended that when using the `cp(1)` command, you use the `--sparse=never` option, which switches off the optimization described earlier. This causes `cp(1)` to actually write the sparse sections to the destination file, effectively writing the entire file in a single Bitfile segment. However, the `mv(1)` command has no equivalent option, so it is recommended to `cp --sparse=never` into HPSSFS-FUSE and unlink the source file instead of trying to use `mv(1)`.

6.2. SAMBA

SAMBA is a suite of UNIX applications that speak the SMB/CIFS protocol. Microsoft Windows® operating systems and the OS/2® operating system use SMB to perform client-server networking for file and printer sharing and associated operations. By supporting this protocol, SAMBA enables computers running UNIX to get in on the action, communicating with the same networking protocol as Microsoft Windows and appearing as another Windows system on the network from the perspective of a Windows client. A SAMBA server offers the following services:

- Share one or more directory trees
- Share one of more Distributed File System (DFS) trees
- Share printers installed on the server among Windows clients on the network
- Assist clients with network browsing
- Authenticate clients logging onto a Windows domain
- Provide or assist with Windows Internet Name Service (WINS) name-server resolution

The SAMBA suite also includes client tools that allow users on a UNIX system to access folders and printers that Windows systems and SAMBA servers offer on the network.

6.2.1. Configuration and Code Modification Suggestions

One site added a patch which disables the feature where Windows can set a "sticky" file modification time. This causes the file modification time to be updated after every received block (4KB-64KB depending), which is a round trip to the metadata server. If the HPSSFS-FUSE Gateway machine is not local, but attached to HPSS via a WAN, this type of change is important to maintain high transaction performance.

This is a change that sites would like to see in the SAMBA baseline, but as it stands today, such a change which would benefit other non-local filesystems (e.g. NFS) has not been adopted by the keepers of the SAMBA code. Local modifications to the SAMBA code are required.

Sites may want to make a modification to SAMBA to check for and delete a file before creating it using an open for write with truncate. This allows a site to perform a Class of Service (COS) change on an existing file. Otherwise, specifying a different COS (either by an explicit ioctl call or using an alternate HPSSFS-FUSE mount) is ignored.

6.3. NFS

6.3.1. Overview

Network File System (NFS) is an RPC protocol used to share files and directories across a network.

6.3.2. Configuration Suggestions

At present, few HPSS sites are currently using NFS over HPSSFS-FUSE in production. Based on past experimentation, however, we recommend the following:

- The *nfs3* mount option should be included for HPSSFS-FUSE mounts exported for use by NFSv3 clients. Alternative HPSSFS-FUSE exports with the *nfs4* mount option should be provided for NFSv4 clients.
- Since NFS is incompatible with junctions, the *nfs3* and *nfs4* mount options disable junctions. It is possible to mount fileset roots directly, avoiding the need for junctions. Secondary mount points may be overlaid on an existing mount to provide a continous namespace that resembles the HPSS namespace.
- A large number of `nfsd(8)` processes has not been shown to improve NFS performance with HPSSFS-FUSE. It is recommended that the administrator starts with no more than 4 or 8 `nfsd(8)` processes and adjust upwards only after conferring with HPSS support.

6.4. Secure FTP

SFTP is the SSH® File Transfer Protocol (sometime referred to as the Secure File Transfer Protocol). Some sites use SFTP clients to access the HPSS namespace via the HPSSFS-FUSE interface of HPSS. This allows for a secure, encrypted access from client machines that are not supported via the Client API, or just as a more general interface for users that do not want to run the HPSS Client API.

6.4.1. Configuration and Code Modification Suggestions

Sites may want to consider making a small patch to the SFTP code to delete a file before creating it using an open for write with truncate. This allows a different Class of Service (COS) to be used if the same file is rewritten. This was done in the `sftp-server(8)` and `scp(1)` Linux code at one of the HPSS sites.

6.5. Apache

6.5.1. Overview

The Apache HTTP Server Project is an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows operating systems. The goal of this project is to provide a secure, efficient, and extensible server that provides HTTP services in sync with the current HTTP standards. Some sites use the HTTP server to run a CGI program to give their users an interface to upload and download files from their HPSSFS-FUSE system.

6.5.2. Configuration Suggestions

Some sites use a CGI program to provide their users with the ability to upload and download files though a web interface. There were no code or configuration changes made to HPSSFS-FUSE in order to get this to work. It was suggested that the following line in `httpd.conf` be uncommented:

- `#EnableSendfile off`

6.5.3. Recommendations

Apache on top of HPSSFS-FUSE works well for deep archive-type access where infrequently

used data can be back-stored in HPSS. For frequently accessed data, or frequently updated information as found on most web-services (e.g. news or sales-oriented site), HPSSFS-FUSE is probably not a good fit unless there is substantial HPSS disk cache and files rarely need to be staged back from tape.

7. Mount Options

HPSSFS-FUSE has a multitude of options to configure mount points.

7.1. Credentials

These are mount options related to setting up HPSS credentials.

Option	Description	Example	Default
auth	Primary authenticator.	`auth=auth_keytab:/var/hpss/etc/hpss.unix.keytab`	`$HPSS_PRIMARY_AUTHENTICATOR`
authmech	Authentication mechanism.	`authmech=unix`	`$HPSS_PRIMARY_AUTHN_MECH`
authtype	Authentication type.	`authtype=auth_keytab`	Derived from *auth* value
princ	Principal name.	`princ=hpssfs`	`$HPSS_PRINCIPAL_FS`

7.2. HPSS Options

These are the mount options related to HPSS.

Option	Description	Example	Default
cos	COS ID on newly created file.	`cos=1`	`0` (default COS)
family	Family ID on newly created files.	`family=1`	`0` (None)
maxfsz	Maximum offset to allow writing in MB.	`maxfsz=1024`	`0` (unlimited)
[no]maxsegz	Use the maximum COS storage segment size when creating a new file.	`maxsegsz`	`nomaxsegsz`

Option	Description	Example	Default
[no]san [4]	Enable SAN3P.	`san`	Derived from `$HPSS_API_SAN3P`
[no]stage	Enable staging files on open.	`nostage`	`stage`

7.3. Checksum Options

These are the mount options related to checksum.

Option	Description	Example	Default
cksum	Algorithm to use for checksum processing. Valid options (case-insensitive): • `none` • `adler32` • `crc32` • `md5` • `sha1` • `sha224` • `sha256` • `sha384` • `sha512`	`cksum=md5`	`none` (no checksum processing)
nch	What to do when a non-checksummed file is opened. • `f` — Fail to open • `g` — Generate a new checksum • `i` — Do not perform checksum processing	`nch=g`	`f`
rvl [5]	Seconds for how long a file is valid since it was successfully verified.	`rvl=3600`	`0`

Option	Description	Example	Default
ckstyle [6]	Where to store checksum attributes. • `filehash` — Store in File Hash metadata • `uda` — Store in UDA metadata • `hybrid` — Store in both File Hash and UDA metadata	`ckstyle=filehash`	`hybrid`

7.4. Other HPSSFS-FUSE Options

Option	Description	Example	Default
attrtimeo	Seconds to keep cached file attributes.	`attrtimeo=60`	`60`
entrytimeo	Seconds to keep cached entry names.	`entrytimeo=30`	`30`
stagetimeo	Seconds to wait for stage completion.	`stagetimeo=3600`	`3600`
trace	Level of detail for logging.	`trace=1`	`0`
ip	Set API hostname. Value provided can be a hostname, IP address, or network interface.	`ip=eth0`	`$HPSS_API_HOSTNAME`
stream	Buffer size for readahead/writeback in megabytes.	`stream=8`	`8`
nostream	Use unbuffered I/O (equivalent to stream=0).	`nostream`	Not used
maxfsz	Maximum offset to allow writing in megabytes.	`maxfsz=1024`	`0` (unlimited)

Option	Description	Example	Default
autopurgelock	Maximum file size in bytes to auto-purge-lock. See Auto Purge Lock for more information.	`autopurgelock=1048576`	`0` (disabled)
[no]dio	Whether to allow files to be opened with `O_DIRECT`.	`dio`	`nodio`
[no]nfs3	Whether to turn on optimizations for NFSv3. See NFS for more information.	`nfs3`	`nonfs3`
[no]nfs4	Whether to turn on optimizations for NFSv4. See NFS for more information.	`nfs4`	`nonfs4`

ip Option

Avoid using loopback addresses for the *ip* mount option. HPSSFS-FUSE will use this address for stage callbacks and for Mover connections. If a Core Server or Mover cannot connect to the address provided, stage callbacks and Mover I/O will fail.

nfs3 and *nfs4* Options

The *nfs3* and *nfs4* mount options are mutually exclusive. Please provide separate exports for use by NFSv3 and NFSv4 clients.

7.5. FUSE Options

These are options that are passed through to the FUSE filesystem. See `mount.fuse(8)` for more information.

Option	Description
-d [z]	Enable FUSE debugging. Implies **-f**.
-f [z]	Run in foreground.
-s [z]	Make FUSE requests single-threaded.

Option	Description
allow_other [8]	Allow other users to access the mount point. This option is recommended for privileged mounts which use the hpssfs principal.
allow_root [8]	Allow root user to access the mount point.
auto_unmount [9]	Automatically unmount if FUSE server process dies.
debug	Enable FUSE debugging. Same as -d.
nonempty	Allow mount even if mount point is not empty.
readdirplus [10]	Enable readdirplus.

7.6. Kernel Options

These are options available to any mount point. See `mount(8)` for more information.

Option	Description
ro	Mount as read-only.
rw	Mount as read-write.
[no]atime	Whether to update inode access times.
[no]dev [11]	Whether to allow access to special devices. HPSS does not support special devices, so this option has no effect.
[no]exec [11]	Whether to allow programs to be executed.
[no]suid	Whether to honor the set-uid bit on programs.
[a]sync	Whether to perform synchronous I/O.
dirsync	Complete all directory updates synchronously.
context **defcontext** **fscontext** **rootcontext**	Default SELinux labels [12].

8. Extensions

HPSSFS-FUSE supports a number of extensions to the POSIX library interface to enable users

to control specific HPSS attributes, such as setting the Class of Service (COS) value. It also supports additional operations that occur on the opening and closing of files.

8.1. `ioctl(2)` Interface

Command	Description	Example
`HPSSFS_GET_COS`	Get COS	getcos.c
`HPSSFS_SET_COS_HINT`	Set COS hints by COS ID	setcoshint.c
`HPSSFS_SET_FSIZE_HINT`	Set COS hints by file size	setfsizehint.c
`HPSSFS_SET_MAXSEGSZ_HINT`	Set HINTS_FORCE_MAX_SSEG COS hints flag	setmaxsegszhint.c
`HPSSFS_PURGE_CACHE`	Purge file data from the kernel cache	purge_cache.c
`HPSSFS_PURGE_LOCK`	Purge lock or unlock a file	purge_lock.c

8.1.1. Examples

getcos.c

```c
/* getcos.c */
#include <fcntl.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/hpssfs.h>

int main(int argc, char *argv[])
{
  int         fd, rc;
  const char *filename;
  uint32_t    cos;

  if((filename = *++argv) == NULL)
  {
    fprintf(stderr, "Usage: %s <filename>", argv[0]);
    return EXIT_FAILURE;
  }

  fd = open(filename, O_RDONLY|O_NONBLOCK);
  if(fd < 0)
  {
    perror("open");
    return EXIT_FAILURE;
  }

  rc = ioctl(fd, HPSSFS_GET_COS, &cos);
  if(rc != 0)
  {
    perror("ioctl");
    close(fd);
    return EXIT_FAILURE;
  }

  close(fd);

  printf("COS is %" PRIu32 "\n", cos);
  return EXIT_SUCCESS;
}
```

setcoshint.c

```c
/* setcoshint.c */
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/hpssfs.h>

int main(int argc, char *argv[])
{
  int           rc, fd;
  const char    *filename, *cosstr;
  unsigned long val;
  uint32_t      cos;

  if((filename = *++argv) == NULL
  || (cosstr = *++argv) == NULL)
  {
    fprintf(stderr, "Usage: %s <filename> <cos-id>", argv[0]);
    return EXIT_FAILURE;
  }

  errno = 0;
  val = strtoul(cosstr, NULL, 0);
  if(val > UINT32_MAX || errno != 0)
  {
    fprintf(stderr, "Invalid COS ID '%s'\n", cosstr);
    return EXIT_FAILURE;
  }
  cos = val;

  fd = open(filename, O_RDWR|O_CREAT|O_NONBLOCK, 0644);
  if(fd < 0)
  {
    perror("open");
    return EXIT_FAILURE;
  }

  rc = ioctl(fd, HPSSFS_SET_COS_HINT, &cos);
  if(rc != 0)
  {
    perror("ioctl");
    close(fd);
    return EXIT_FAILURE;
  }

  close(fd);

  return EXIT_SUCCESS;
}
```

setfsizehint.c

```c
/* setfsizehint.c */
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/hpssfs.h>

int main(int argc, char *argv[])
{
  int                rc, fd;
  const char         *filename, *size;
  unsigned long long val;
  uint64_t           filesize;

  if((filename = *++argv) == NULL
  || (size = *++argv) == NULL)
  {
    fprintf(stderr, "Usage: %s <filename> <size>", argv[0]);
    return EXIT_FAILURE;
  }

  errno = 0;
  val = strtoull(size, NULL, 0);
  if(val > UINT64_MAX || errno != 0)
  {
    fprintf(stderr, "Invalid size '%s'\n", size);
    return EXIT_FAILURE;
  }
  filesize = val;

  fd = open(filename, O_RDWR|O_CREAT|O_NONBLOCK, 0644);
  if(fd < 0)
  {
    perror("open");
    return EXIT_FAILURE;
  }

  rc = ioctl(fd, HPSSFS_SET_FSIZE_HINT, &filesize);
  if(rc != 0)
  {
    perror("ioctl");
    close(fd);
    return EXIT_FAILURE;
  }

  close(fd);

  return EXIT_SUCCESS;
}
```

setmaxsegszhint.c

```c
/* setmaxsegszhint.c */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/hpssfs.h>

int main(int argc, char *argv[])
{
  int         rc, fd;
  const char *filename;

  if((filename = *++argv) == NULL)
  {
    fprintf(stderr, "Usage: %s <filename>", argv[0]);
    return EXIT_FAILURE;
  }

  fd = open(filename, O_RDWR|O_CREAT|O_NONBLOCK, 0644);
  if(fd < 0)
  {
    perror("open");
    return EXIT_FAILURE;
  }

  rc = ioctl(fd, HPSSFS_SET_MAXSEGSZ_HINT);
  if(rc != 0)
  {
    perror("ioctl");
    close(fd);
    return EXIT_FAILURE;
  }

  close(fd);

  return EXIT_SUCCESS;
}
```

purge_cache.c

```c
/* purge_cache.c */
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/hpssfs.h>

int main(int argc, char *argv[])
{
  int         fd, rc, failed = 0;
  const char *filename;

  if((filename = *++argv) == NULL)
  {
    fprintf(stderr, "Usage: %s file1 [file2...]\n", argv[0]);
    return EXIT_FAILURE;
  }

  do
  {
    fd = open(filename, O_RDONLY|O_NONBLOCK);
    if(fd < 0)
    {
      fprintf(stderr, "open(%s): %s\n", filename, strerror(errno));
      failed = 1;
    }

    rc = ioctl(fd, HPSSFS_PURGE_CACHE);
    if(rc != 0)
    {
      fprintf(stderr, "ioctl(%s, HPSSFS_PURGE_CACHE): %s\n",
              filename, strerror(errno));
      failed = 1;
    }
    else
      fprintf(stdout, "purged %s\n", filename);

    close(fd);
  } while((filename = *++argv) != NULL)

  if(failed)
    return EXIT_FAILURE;

  return EXIT_SUCCESS;
}
```

purge_lock.c

```c
/* purge_lock.c */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/hpssfs.h>

int main(int argc, char *argv[])
{
  int         rc, fd;
  const char *filename, *cmd;
  uint32_t    lock;

  if((filename = *++argv) == NULL
  || (cmd = *++argv) == NULL)
  {
    fprintf(stderr, "Usage: %s <filename> <lock|unlock>", argv[0]);
    return EXIT_FAILURE;
  }

  if(strcasecmp(cmd, "lock") == 0)
    lock = 1;
  else if(strcasecmp(cmd, "unlock") == 0)
    lock = 0;
  else
  {
    fprintf(stderr, "Usage: %s <filename> lock|unlock", argv[0]);
    return EXIT_FAILURE;
  }

  fd = open(filename, O_RDONLY|O_NONBLOCK);
  if(fd < 0)
  {
    perror("open");
    return EXIT_FAILURE;
  }

  rc = ioctl(fd, HPSSFS_PURGE_LOCK, &lock);
  if(rc != 0)
  {
    perror("ioctl");
    close(fd);
    return EXIT_FAILURE;
  }

  close(fd);

  return EXIT_SUCCESS;
}
```

HPSSFS-FUSE supports the `fallocate(2)` system call [13].

8.2. `fallocate(2)`

The `fallocate(2)` system call allows the user to perform two operations.

8.2.1. Preallocate

This operation allows the user to preallocate disk space on a disk Storage Class at the top of the file's COS Hierarchy. If this operation succeeds, a write to the file up to the preallocated size cannot fail due to insufficient space.

8.2.2. Punch Hole

This operation allows the user to punch a hole in a file. Essentially in HPSS, this means removing the specified portion of Bitfile segments, which consequently makes that portion of the file filled with zeros. As a side effect, some storage segments may also be freed.

8.3. Linux Extended Attributes

HPSSFS-FUSE supports Linux Extended Attributes (xattrs). These are manipulated using the `getxattr(2)`, `setxattr(2)`, `listxattr(2)`, and `removexattr(2)` system calls; the `attr_get(3)`, `attr_set(3)`, `attr_multi(3)`, and `attr_remove(3)` library calls; and the `getfattr(1)`, `setfattr(1)`, and `attr(1)` commands. See `attr(5)` for more information.

8.3.1. Features and Limitations

Improvements Over HPSSFS-VFS

- HPSSFS-FUSE supports xattrs with binary values. Previously, xattr values were limited to text-only.
- HPSSFS-FUSE supports xattrs with values up to 64KB (enforced by the kernel). Previously, xattr values were limited to under 1KB.

Limitations

- In order to support maximum-sized xattr values, the Core Server must have the following options in `/var/hpss/etc/env.conf`:
  - `HPSS_API_XMLSIZE_LIMIT=131072` (or greater)
  - `HPSS_API_XMLREQUEST_LIMIT=131072` (or greater)
- POSIX ACL's and SELinux labels are not supported at this time due to limitations with the FUSE kernel module.

8.3.2. *system* Namespace

HPSSFS-FUSE supports arbitrary xattrs in the *system* namespace. However, the *system.posix_acl_access* and *system.posix_acl_default* xattrs are currently disabled because FUSE does not correctly support them. Additionally, the *system.hpssfs* namespace is reserved for HPSSFS-FUSE runtime settings, and the *system.hpss* namespace is reserved for HPSS attributes.

Name	Description	Access

Name	Description	Access
system.hpssfs.trace	View or set the current trace level for the mount point. This xattr only exists on the mounted directory.	Read/Write
system.hpss.account	HPSS Account ID	Read/Write; files only
system.hpss.bitfile	HPSS Bitfile ID	Read; files only
system.hpss.comment	HPSS Comment	Read/Write
system.hpss.cos	HPSS COS ID	Read/Write; files only
system.hpss.family	HPSS File Family ID	Read/Write; files only
system.hpss.fileset	HPSS Fileset	Read
system.hpss.level	HPSS Level Data	Read
system.hpss.opens	HPSS Opens	Read; files only
system.hpss.optimum	HPSS Optimum Access Size	Read; files only
system.hpss.reads	HPSS Reads	Read; files only
system.hpss.realm	HPSS Realm ID	Read
system.hpss.subsys	HPSS Subsys ID	Read
system.hpss.writes	HPSS Writes	Read; files only
system.hpss.hash [6]	HPSS File Hash Metadata	Read/Write; files only
system.hpss.trash.parent [14]	HPSS Trash Parent ID	Read
system.hpss.trash.uid [14]	HPSS Trash User ID	Read
system.hpss.trash.timedeleted [14]	HPSS Trash Time Deleted	Read
system.hpss.trash.timecreated [14]	HPSS Trash Time Created	Read
system.hpss.trash.timelastread [14]	HPSS Trash Time Last Read	Read

Name	Description	Access
system.hpss.trash.timemodified [14]	HPSS Trash Time Last Modified	Read
system.hpss.trash.path [14]	HPSS Trash Path	Read
system.hpss.trash.name [14]	HPSS Trash Name	Read

8.3.3. *trusted* Namespace

HPSSFS-FUSE supports arbitrary xattrs in the trusted namespace for super users. These xattrs are stored in HPSS UDA metadata under the XPath `/hpss/fs`, e.g. the xattr *trusted.name* will be located at the XPath `/hpss/fs/trusted.name`.

8.3.4. *security* Namespace

HPSSFS-FUSE supports arbitrary xattrs in the security namespace for all users. However, the *security.selinux* and *security.capability* xattrs are currently disabled because FUSE does not properly support them.

8.3.5. *user* Namespace

HPSSFS-FUSE supports arbitrary xattrs in the user namespace for all users. Most of these xattrs are stored in the HPSS UDA metadata under the XPath `/hpss/fs`, e.g. the xattr *user.name* will be located at the XPath `/hpss/fs/user.name`. The checksum attributes are stored in a separate XPath for interoperability with other interfaces.

8.4. Checksum

HPSSFS-FUSE Checksum feature is a file-level checksumming mechanism which generates file checksums when files are created and written. When files are later opened, their contents are verified against the generated checksum. If the checksum does not match, the file fails to open.

8.4.1. Operation

This section will briefly describe the operations of the HPSSFS-FUSE Checksum feature, assuming the checksum option is enabled.

File Creation and Inline Checksumming

When a file is created, a new hash context is created which uses the algorithm specified by the cksum mount option. As data is appended to the file, the data is also appended to the hash context, and the context's offset is moved forward. If the file offset of an incoming write is past the current context's offset, then a zero-filled buffer is appended to the context in order to fill the gap. These two operations are inline checksumming. Once the file is closed, the context is finalized and the resulting digest is stored.

If the file offset of an incoming write is before the current context's offset, then inline checksumming is disabled. No more checksum processing will be performed until the file is closed.

File Open Readback

When a file is opened, the entire file is read. The file's contents are checksummed and verified against the checksum metadata. If the checksums do not match, the file fails to open. The file can resume inline checksumming with the context's offset pointed at the end of the file.

File Close Readback

A file may be read back upon close if any of the following conditions are met:

- Inline checksumming was canceled due to writing prior to the context's offset (otherwise known as "random I/O").
- Multiple users have opened the file.

In these cases, the file needs to be read back in order to generate its checksum. Once the file has been processed, its checksum metadata is updated.

Checksum Readback

Readbacks for the purpose of generating new checksum information can happen in one of two ways:

1. Generate on open if no checksum information exists and *nch=g*
2. Generate on close if random I/O or concurrent users is detected

In both cases, we must rely on the data which resides in HPSS to generate the checksum. You should minimize these cases because the checksum will be generated based on the data read from HPSS. It is possible that the data could have already been corrupted by the time we read it, resulting in a checksum that matches the corrupted data. From then on, integrity checks will continue to pass as long as the generated checksum matches the corrupted data.

Supported Algorithms

The HPSSFS-FUSE Checksum feature supports the following hashing algorithms:

- *Adler32*
- *CRC32*
- *MD5*
- *SHA1*
- *SHA224*
- *SHA256*
- *SHA384*
- *SHA512*

Concurrency

The HPSSFS-FUSE Checksum feature is designed to consider several forms of concurrency. They are all implemented by using UDA's to create a persistent lock and persistent leases. When a file is opened for checksum processing, the mount point acquires a UDA lock and lease. All threads/processes which open the file on a single mount point have their own context, and so

can be viewed as separate instances from the standpoint of concurrency. Each time a file is opened, the open count for the file will be incremented. Upon close, the open count will be decremented. If you reach an open count of zero and detect that other users had opened the file, then you will perform a readback-on-close to regenerate the checksum metadata. Similarly, readback-on-open will only be performed if you are the first to open a file.

8.4.2. Configuration

Mount Options

There are several mount options that control the HPSSFS-FUSE Checksum feature:

- `cksum` — This options chooses which algorithm to use for checksum processing when a new checksummed file is created. The algorithm is always determined by checksum metadata for existing checksummed files. This option is required to enable checksum. If it is not specified, checksum processing will never take place on this mount point. The supported values (case-insensitive) are:
  - `cksum=none` — Disable checksum; default
  - `cksum=adler32` — Use Adler32 algorithm
  - `cksum=crc32` — Use CRC32 algorithm
  - `cksum=md5` — Use MD5 algorithm
  - `cksum=sha1` — Use SHA1 algorithm
  - `cksum=sha224` — Use SHA224 algorithm
  - `cksum=sha256` — Use SHA256 algorithm
  - `cksum=sha384` — Use SHA384 algorithm
  - `cksum=sha512` — Use SHA512 algorithm
- `nch` — This option chooses what to do when a non-checksummed file is opened. Otherwise, normal checksumming operations occur. The supported values are:
  - `nch=i` — Do not perform any checksum processing; allow non-checksummed files to open successfully. Concurrency bookkeeping will still occur, and if concurrency is detected, this will still perform readback-on-close checksumming.
  - `nch=g` — Generate a new checksum. This will perform readback-on-open checksumming and apply the generated checksum to the metadata.
  - `nch=f` — If the file is non-checksummed, the open will fail; default
- `rvl` — Revalidation timeout: number of seconds that a checksum is considered valid since it was last successfully verified. The default is 0, so checksums are verified on every open. A non-zero value allows subsequent opens to succeed without performing a readback if they occur within this timeout since the last verification by this mount point. This option must have a non-zero value for checksumming to work when using the *nfs3* or *nfs4* mount options for NFS optimizations.
- `ckstyle` [6] — Where to store checksum metadata. The supported values are:
  - `ckstyle=filehash` — Store in File Hash metadata
  - `ckstyle=uda` — Store in UDA metadata
  - `ckstyle=hybrid` — Store in both File Hash and UDA metadata; default

Relation to Other Mount Options

Readbacks occur separately from normal file activity. Due to this, some mount options apply differently to readbacks.

- `[no]stage` — Has no effect; readbacks always stage the file

8.4.3. External Application Interoperability

HPSSFS-FUSE Checksum is designed to be compatible with other applications which use HPSS Checksums, including HSI, hpsssum, and HPSSFS-VFS. These programs use a unified UDA path for storing checksum metadata. There is no mechanism to ensure coherency between HPSSFS-FUSE and HSI/hpsssum.

8.4.4. Checksum UDA Paths

The following is a list of UDA paths used for checksum and their purposes:

XPath	xattr	Description
`/hpss/user/cksum/checksum`	*user.hash.checksum*	The hash value of the file using the specified algorithm
`/hpss/user/cksum/algorithm`	*user.hash.algorithm*	The algorithm used to calculate the hash
`/hpss/user/cksum/state`	*user.hash.state*	The state of the current checksum value. • *Valid* — The current checksum is valid • *Invalid* — The digest did not match the readback digest • *Error* — An error occurred when trying to readback the file • *NoEntry* — Not all of the required checksum UDA's were present during the last readback
`/hpss/user/cksum/lastupdate`	*user.hash.lastupdate*	A UNIX timestamp of the last time UDA's were updated
`/hpss/user/cksum/errors`	*user.hash.errors*	Number of readback errors since the last successful readback
`/hpss/user/cksum/filesize`	*user.hash.filesize*	Size of the file
`/hpss/user/cksum/app`	*user.hash.app*	Name of the application which last updated the checksum UDA's

HPSSFS-FUSE-Specific UDA Paths

The following is a list of UDA paths which are only used by HPSSFS-FUSE and HPSSFS-VFS. They should not be modified by end users, otherwise unexpected checksum behavior may occur.

XPath	Description
`/hpss/fs /user.open.total`	Number of concurrent opens on this file
`/hpss/fs/user.mounts/*`	List of mount points that have this file open
`/hpss/fs /user.open.lock`	Lock to serialize UDA access
`/hpss/fs/user.leases/*`	List of mount point leases. This attribute only applies to HPSS's root directory. It is the "heartbeat" of checksum mount points. If a lease expires, then any lock held by that mount point is invalid.

8.5. Auto Purge Lock

Auto Purge Lock is a feature that prevents files under a given size from being purged after migration. It is controlled via the autopurgelock mount option.

When enabled, if a file is written to, it becomes a candidate for Auto Purge Lock. Once the file is closed, if its size is less than or equal to the size specified by the autopurgelock mount option, then the file is automatically purge locked. The file can still be migrated, but it will not be purged while it remains purge locked.

9. References

- HPSS Management Guide
- HPSS Installation Guide
- HPSS Programmer's Reference

10. Trademarks

Apache® is a registered trademark of Apache Software Foundation.

Arch™ is a trademark of Aaron Griffin and/or Judd Vinet.

CentOS™ is a trademark of Red Hat, Inc.

Debian® is a registered trademark of Software in the Public Interest, Inc.

Gentoo® is a registered trademark of Gentoo Foundation, Inc.

High Performance Storage System™ and HPSS™ are trademarks of International Business Machines Corporation.

---

1. HPSSFS-FUSE is only officially supported on RHEL® (Intel® and PowerPC® versions.). Minimal testing has been performed on all major Linux distributions (Arch Linux™, CentOS™, Debian®, Fedora®, Gentoo®, Linux Mint™, Mageia™, openSUSE®, Oracle® Linux, slackware®, SUSE®, and Ubuntu®), but only RHEL goes through a full testing cycle.
2. HPSSFS-FUSE releases are only tested against the latest release of HPSS.
3. Be extra careful with this command, especially if running as root!
4. SAN3P transfers are only available for privileged mounts.
5. Required to be non-zero for NFS.
6. Requires HPSS File Hash (E2EDI) feature.
7. These options are only useful for diagnostic purposes.
8. Availability of this option is controlled by `/etc/fuse.conf`.
9. Requires FUSE 2.9 or later.
10. Requires FUSE 3.0 or later.
11. Can only be overridden by a privileged user.
12. FUSE does not appear to allow these options at this time.
13. Requires FUSE 2.9.1 or later.
14. Requires HPSS Trashcan feature.

Version 600
Last updated 2015-01-07 10:05:41 CST