UnifyFS Documentation

Release 1.1

Kathryn Mohror, Adam Moody, Sarp Oral, Feiyi Wang, Hyogi Sim,

Jan 19, 2024

USER GUIDE

CHAPTER

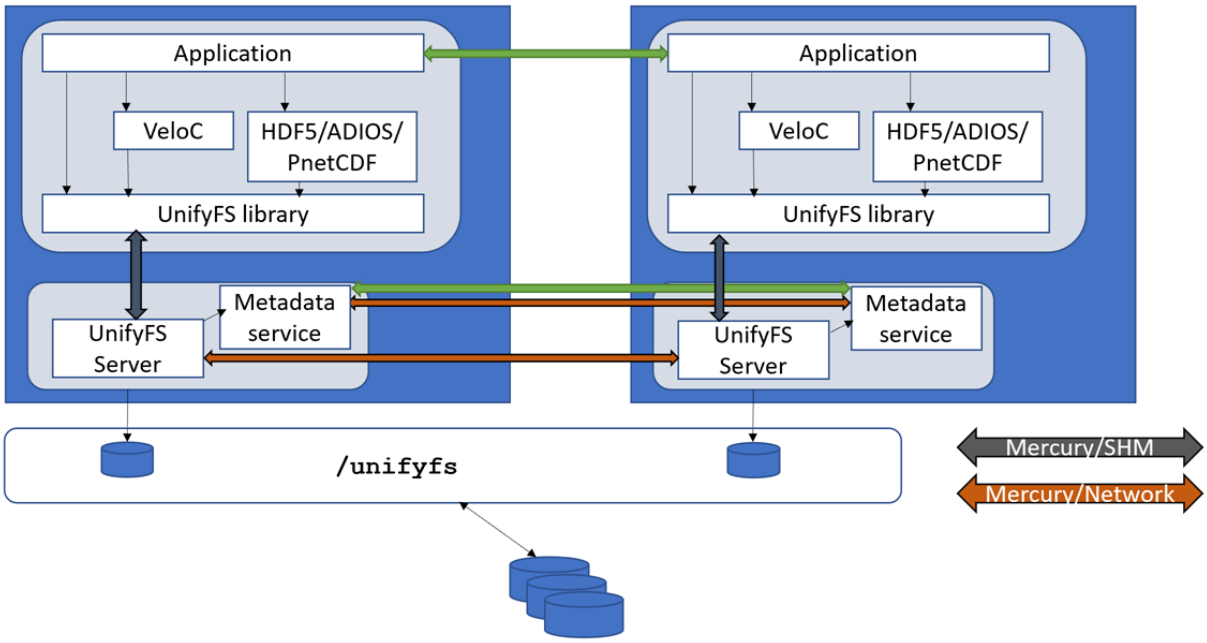
ONE

OVERVIEW

UnifyFS is a user-level file system under active development that supports shared file I/O over distributed storage on HPC systems, e.g., node-local burst buffers. With UnifyFS, applications can write to fast, scalable, node-local burst buffers as easily as they do to the parallel file system. UnifyFS is designed to support common I/O workloads such as checkpoint/restart and other bulk-synchronous I/O workloads typically performed by HPC applications.

Because the UnifyFS file system is implemented at user-level, the file system is visible only to applications linked with the UnifyFS client library. A consequence of this is that traditional file system tools (ls, cd, etc.) installed by system administrators cannot act on files in a UnifyFS file system because they are not linked against the UnifyFS client library. The lifetime of a UnifyFS file system is the duration of the execution of the UnifyFS server processes, which is typically for the duration of an HPC job. When the servers exit, the UnifyFS file system terminates. Users must copy files that need to be persisted beyond the lifetime of the job from UnifyFS to a permanent file system. UnifyFS provides an API and a utility to perform these copies.

1.1 High Level Design



This section provides a high level design of UnifyFS. UnifyFS presents a shared namespace (e.g., /unifyfs as a mount point) to all compute nodes in a job allocation. There are two main components of UnifyFS: the UnifyFS library and the UnifyFS server.

The UnifyFS library is linked into the user application. The library intercepts I/O calls from the application and sends I/O requests for UnifyFS files on to the UnifyFS server. The library forwards I/O requests for all other files on to the system. The UnifyFS library uses ECP GOTCHA as its primary mechanism to intercept I/O calls. The user application is linked with the UnifyFS client library and perhaps a high-level I/O library, like HDF5, ADIOS, or PnetCDF.

A UnifyFS server process runs on each compute node in the job allocation. The UnifyFS server handles the I/O requests from the UnifyFS library. The UnifyFS server uses ECP Mochi to communicate with user application processes and server processes on other nodes.

1.2 UnifyFS Citation

We recommend that you use this as the primary citation for UnifyFS as well as a reference for further details on the UnifyFS architecture and semantics:

> Michael Brim, Adam Moody, Seung-Hwan Lim, Ross Miller, Swen Boehm, Cameron Stanavige, Kathryn Mohror, Sarp Oral, "UnifyFS: A User-level Shared File System for Unified Access to Distributed Local Storage," 37th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2023), St. Petersburg, FL, May 2023.

1.3 UnifyFS Videos

1.3.1 Quickstart

Quick overview on what UnifyFS is and how to use it.

1.3.2 ECP Tutorial

More in-depth recording of the problems UnifyFS solves and a how-to using a pre-1.0 beta version.

UnifyFS Tutorial Slides - ECP 2022

`Download PDF.`

Changes since above ECP 2022 Tutorial:

- Video 30:04 | Slide 19 - Variants `boostsys` and `preload` have since been added.
- Video 31:08 | Slide 22 - OpenSSL is also a dependency that was missed on this slide.
- Video 41:10 | Slide 48 - User Guide url starts with *https*, not *http*.

For full changes since the pre-1.0 beta version of UnifyFS used in the May 2022 tutorial, see the UnifyFS release notes.

CHAPTER

TWO

DEFINITIONS

This section defines some terms used throughout the document.

2.1 Job

A set of commands that is issued to the resource manager and is allocated a set of nodes for some duration

2.2 Run or Job Step

A single application launch of a group of one or more application processes issued within a job

CHAPTER

THREE

ASSUMPTIONS AND SEMANTICS

In this section, we provide assumptions we make about the behavior of applications that use UnifyFS and about the file system semantics of UnifyFS.

3.1 System Requirements

The system requirements to run UnifyFS are:

- Compute nodes must be equipped with local storage device(s) that UnifyFS can use for storing file data, e.g., SSD or RAM.
- The system must support the ability for UnifyFS user-level server processes to run concurrently with user application processes on compute nodes.

3.2 Application Behavior

UnifyFS is specifically designed to support the bulk-synchronous I/O pattern that is typical in HPC applications, e.g., checkpoint/restart or output dumps. In bulk-synchronous I/O, I/O operations occur in separate write and read phases, and files are not read and written simultaneously. For example, files are written during checkpointing (a write phase) and read during recovery/restart (a read phase). Additionally, parallel writes and reads to shared files occur systematically, where processes access computable, regular offsets of files, e.g., in strided or segmented access patterns, with ordering of potential conflicting updates enforced by inter-process communication. This behavior is in contrast to other I/O patterns that may perform random, small writes and reads or overlapping writes without synchronization.

UnifyFS offers the best performance for applications that exhibit the bulk synchronous I/O pattern. While UnifyFS does support deviations from this pattern, the performance might be slower and the user may have to take additional steps to ensure correct execution of the application with UnifyFS. For more information on this topic, refer to the section on *commit consistency semantics in UnifyFS*.

3.3 Consistency Model

The UnifyFS file system does not support strict POSIX consistency semantics. (Please see Chen et al., HPDC 2021 for more details on different file system consistency semantics models.) Instead, UnifyFS supports two different consistency models: *commit consistency semantics* when a file is actively being modified; and *lamination semantics* when the file is no longer being modified by the application. These two consistency models provide opportunities for UnifyFS to provide better performance for the I/O operations of HPC applications.

3.3.1 Commit Consistency Semantics in UnifyFS

Commit consistency semantics require explicit *commit* operations to be performed before updates to a file are globally visible. We chose commit consistency semantics for UnifyFS because it is sufficient for correct execution of typical HPC applications that adhere to the bulk-synchronous I/O pattern, and it enables UnifyFS to provide better performance than with strict POSIX semantics. For example, because we assume that applications using UnifyFS will not execute concurrent modifications to the same file offset, UnifyFS does not have to employ locking to ensure sequential access to file regions. This assumption allows us to cache file modifications locally which greatly improves the write performance of UnifyFS.

The use of synchronization operations are required for applications that exhibit I/O accesses that deviate from the bulk-synchronous I/O pattern. There are two types of synchronization that are required for correct execution of parallel I/O on UnifyFS: *local synchronization* and *inter-process synchronization*. Here, *local synchronization* refers to synchronization operations performed locally by a process to ensure that its updates to a file are visible to other processes. For example, a process may update a region of a file and then execute `fflush()` so that a different process can read the updated file contents. *Inter-process synchronization* refers to synchronization operations that are performed to enforce ordering of conflicting I/O operations from multiple processes. These inter-process synchronizations occur outside of normal file I/O operations and typically involve inter-process communication, e.g., with MPI. For example, if two processes need to update the same file region and it is important to the outcome of the program that the updates occur in a particular order, then the program needs to enforce this ordering with an operation like an `MPI_Barrier()` to be sure that the first process has completed its updates before the next process begins its updates.

There are several methods by which applications can adhere to the synchronization requirements of UnifyFS.

- **Using MPI-IO**. The (MPI-IO) interface requirements are a good match for the consistency model of UnifyFS. Specifically, the MPI-IO interface requires explicit synchronization in order for updates made by processes to be globally visible. An application that utilizes the MPI-IO interface correctly will already adhere to the requirements of UnifyFS.
- **Using HDF5 and other parallel I/O libraries**. Most parallel I/O libraries hide the synchronization requirements of file systems from their users. For example, parallel (HDF5) implements the synchronization required by the MPI-IO interface so users do not need to perform any explicit synchronization operations in the code.
- **With explicit synchronization**. If an application does not use a compliant parallel I/O library or the developer wants to perform explicit synchronization, local synchronization can be achieved through adding explicit *flush* operations with calls to `fflush()`, `close()`, or `fsync()` in the application source code, or by supplying the `client.write_sync` configuration parameter to UnifyFS on startup, which will cause an implicit *flush* operation after every *write* (note: use of the `client.write_sync` mode can significantly slow down write performance). In this case, inter-process synchronization is still required for applications that perform conflicting updates to files.

During a write phase, a process can deviate from the bulk-synchronous I/O pattern and read any byte in a file, including remote data that has been written by processes executing on remote compute nodes in the

job. However, the performance will differ based on which process wrote the data that is being read:

- If the bytes being read were written by the same process that is reading the bytes, UnifyFS offers the fastest performance and no synchronization operations are needed. This kind of access is typical in some I/O libraries, e.g., HDF5, where file metadata may be updated and read by the same process. (Note: to obtain the performance benefit for this case, one must set the `client.local_extents` configuration parameter.)
- If the bytes being read were written by a process executing on the same compute node as the reading process, UnifyFS can offer slightly slower performance than the first case and the application must introduce synchronization operations to ensure that the most recent data is read.
- If the bytes being read were written by a process executing on a different compute node than the reading process, then the performance is slower than the first two cases and the application must introduce synchronization operations to ensure that the most recent data is read.

In summary, reading the local data (which has been written by processes executing on the same compute node) will always be faster than reading remote data.

Note that, as we discuss above, commit semantics also require inter-process synchronization for potentially conflicting write accesses. If an application does not enforce sequential ordering of file modifications during a write phase, e.g., with MPI synchronization, and multiple processes write concurrently to the same file offset or to an overlapping region, the result is undefined and may reflect the result of a mixture of the processes' operations to that offset or region.

The *VerifyIO* tool can be used to determine whether an application is correctly synchronized.

3.3.2 Lamination Consistency Semantics in UnifyFS

The other consistency model that UnifyFS employs is called "lamination semantics" which is intended to be applied once a file is done being modified at the end of a write phase of an application. After a file is laminated, it becomes permanently read-only and its data is accessible across all the compute nodes in the job without further synchronization. Once a file is laminated, it cannot be further modified, except for being renamed or deleted.

A typical use case for lamination is for checkpoint/restart. An application can laminate checkpoint files after they have been successfully written so that they can be read by any process on any compute node in the job in a restart operation. To laminate a file, an application can simply call chmod() to remove all the write bits, after its write phase is completed. When write bits of a file are removed, UnifyFS will laminate the file. A typical checkpoint write operation with UnifyFS will look like:

```
fd = open("checkpoint1.chk", O_WRONLY)
write(fd, <checkpoint data>, <len>)
close(fd)
chmod("checkpoint1.chk", 0444)
```

We plan for future versions of UnifyFS to support different methods for laminating files, such as a configuration option that supports laminating all files on `close()`.

We define the laminated consistency model to enable certain optimizations while supporting the typical requirements of bulk-synchronous I/O. Recall that for bulk-synchronous I/O patterns, reads and writes typically occur in distinct phases. This means that for the majority of the time, processes do not need to read arbitrary bytes of a file until the write phase is completed, which in practice is when the file is done being modified and closed and can be safely made read-only with lamination. For applications in which processes do not need to access file data modified by other processes before lamination, UnifyFS can optimize write performance by buffering all metadata and file data for processes locally, instead of performing costly

exchanges of metadata between compute nodes on every write. Also, since file contents cannot change after lamination, aggressive caching may be used during the read phase with minimal locking.

3.3.3 File System Consistency Behavior

The following summarizes the behavior of UnifyFS under our two consistency models.

Behavior before Lamination (Commit Consistency)

- `open|close`: A process may open/close a file multiple times.
- `write`: A process may write to any part of a file. If two processes write to the same location concurrently (i.e., without inter-process synchronization to enforce ordering), the result is undefined.
- `read`: A process may read bytes it has written. Reading other bytes is invalid without explicit synchronization operations.
- `rename`: A process may rename a file that is not being actively modified.
- `truncate`: A process may truncate a file. Truncation is a synchronizing operation.
- `unlink`: A process may delete a file.

Behavior after Lamination (Laminated Consistency)

- `open|close`: A process may open/close a laminated file multiple times.
- `write`: All writes to laminated files are invalid - no file modifications are permitted.
- `read`: A process may read any byte in the laminated file.
- `rename`: A process may rename a laminated file.
- `truncate`: Truncation of laminated files is invalid - no file modifications are permitted.
- `unlink`: A process may delete a laminated file.

3.4 Additional File System Behavior Considerations

The additional behavior of UnifyFS can be summarized as follows.

- UnifyFS creates a shared file system namespace across all compute nodes in a job, even if an application process is not running on all compute nodes.
- The UnifyFS shared file system namespace is valid for the lifetime of its server processes, and thus exists across multiple application runs within a job.
- UnifyFS transparently intercepts system level I/O calls of applications and I/O libraries. As such, UnifyFS can be easily coupled with other I/O middleware such as SymphonyFS, high-level I/O libraries such as HDF5, or checkpoint libraries.
- UnifyFS stores file data exclusively in node-local storage. No data is automatically persisted to stable storage like a parallel file system. When the data needs to be persisted to an external file system, users can use the *unifyfs utility* and its data staging at file system termination support.
- UnifyFS can also be used with checkpointing libraries like SCR or VeloC to move data to stable storage periodically.
- The UnifyFS file system will be empty at job start. A user job must populate the file system with any initial data by running client applications or using the data staging support of the *unifyfs utility*.

3.4.1 Failure Behavior

- In the event of a compute node failure or node-local storage device failure, all file data from the processes running on the failed node will be lost.
- In the event of the failure of a UnifyFS server process, all file data from the client processes assigned to that server process (typically on the same compute node) will be lost.
- In the event of application process failures when the UnifyFS server processes remain running, the file data can be retrieved by the local UnifyFS server or a remote UnifyFS server.
- The UnifyFS team plans to improve the reliability of UnifyFS in the event of failures using redundancy scheme implementations available from the VeloC project as part of a future release.

CHAPTER

FOUR

LIMITATIONS AND WORKAROUNDS

4.1 General Limitations

4.1.1 Data Consistency

Overlapping write operations or simultaneous read and write operations require proper synchronization when using UnifyFS. This includes ensuring updates to a file are visible to other processes as well as inter-process communication to enforce ordering of conflicting I/O operations. Refer to the section on *commit consistency semantics in UnifyFS* for more detail.

In short, for a process to read data written by another process, the reader must wait for the writer to first flush any data it has written to the UnifyFS servers. After the writer flushes its data, there must be a synchronization operation between the writer and the reader processes, such that the reader does not attempt to read newly written data until the writer has completed its flush operation.

UnifyFS can be configured to flush data to servers at various points. A common mechanism to flush data is for the writer process to call `fsync()` or `fflush()`. Also, by default, data is flushed when a file is closed with `close()` or `fclose()`.

UnifyFS can be configured to behave more "POSIX like" by flushing newly written data to the server during every write operation. To do this, one can set `UNIFYFS_CLIENT_WRITE_SYNC=ON`. `UNIFYFS_CLIENT_WRITE_SYNC=ON` can decrease write performance as the number of data flush operations may be more than necessary.

4.1.2 File Locking

UnifyFS does not support file locking, and calls to `fcntl()` and `flock()` are not intercepted by UnifyFS. Any calls fall through to the underlying operating system, which should report the corresponding file descriptor as invalid. If not detected, an application will encounter data corruption if it depends on file locking semantics for correctness. Tracing application I/O calls with *VerifyIO* can help determine whether any file locking calls are used.

4.1.3 Directory Operations

UnifyFS does not support directory operations.

4.2 MPI-IO Limitations

4.2.1 Data Consistency

When using MPI-I/O without atomic file consistency, the MPI standard requires the application to manage data consistency by calling `MPI_File_sync()`. After data has been written, the writer must call `MPI_File_sync()`. There must then be a synchronization operation between the writer and reader processes. Finally, the reader must call `MPI_File_sync()` after its synchronization operation with the writer. A common approach is for the application to execute a *"sync-barrier-sync" construct* as shown below:

Listing 1: Sync-barrier-sync Construct

```
MPI_File_sync() //flush newly written bytes from MPI library to file system
MPI_Barrier()   //ensure all ranks have finished the previous sync
MPI_File_sync() //invalidate read cache in MPI library
```

Note: The "barrier" in "sync-barrier-sync" can be replaced by a send-recv or certain collectives that are guaranteed to be synchronized. The synchronization operation does not even need to be an MPI call. See the "Note on the third step" in the VerifyIO README for more information.

Proper data consistency synchronization is also required between MPI-I/O calls that imply write or read operations. For example, `MPI_File_set_size()` and `MPI_File_preallocate()` act as write operations, and `MPI_File_get_size()` acts as a read operation. There may be other MPI-I/O calls that imply write or read operations.

Both `MPI_File_open()` and `MPI_File_close()` implicitly call `MPI_File_sync()`.

Relaxed MPI_File_sync semantics

Data consistency in UnifyFS is designed to be compatible with MPI-I/O application-managed file consistency semantics. An application that follows proper MPI-I/O file consistency semantics using `MPI_File_sync()` should run correctly on UnifyFS, provided that the `MPI_File_sync()` implementation flushes newly written data to UnifyFS.

On POSIX-compliant parallel file systems like Lustre, many applications can run correctly even when they are missing sufficient file consistency synchronization. In contrast, to run correctly on UnifyFS, an application should make all `MPI_File_sync()` calls as required by the MPI standard.

Note: It may be labor intensive to identify and correct all places within an application where file synchronization calls are required. The *VerifyIO* tool can assist developers in this effort.

In the current UnifyFS implementation, it is actually sufficient to make a single call to `MPI_File_sync()` followed by a synchronizing call like `MPI_Barrier()`, e.g.:

```
MPI_File_sync()
MPI_Barrier()
```

Assuming that `MPI_File_sync()` calls `fsync()`, then information about any newly written data will be transferred to the UnifyFS servers. The `MPI_Barrier()` then ensures that `fsync()` will have been called by all clients that may have written data. After the `MPI_Barrier()`, a process may read data from UnifyFS that was written by any other process before that other process called `MPI_File_sync()`. A second call to `MPI_File_sync()` is not (currently) required in UnifyFS.

Furthermore, if `MPI_File_sync()` is known to be a synchronizing collective, then a separate synchronization operation like `MPI_Barrier()` is not required. In this case, an application might simplify to just the following:

```
MPI_File_sync()
```

Having stated those exceptions, it is best practice to adhere to the MPI standard and execute a full sync-barrier-sync construct. There exist potential optimizations such that future implementations of UnifyFS may require the full sequence of calls.

4.3 ROMIO Limitations

4.3.1 Data Consistency

In ROMIO, `MPI_File_sync()` calls `fsync()` and `MPI_File_close()` calls `close()`, each of which flush information about newly written data to the UnifyFS servers. When using ROMIO, an application having appropriate "sync-barrier-sync" constructs as required by the MPI standard will run correctly on UnifyFS.

ROMIO Synchronizing Flush Hint

Although `MPI_File_sync()` is an MPI collective, it is not required to be synchronizing. One can configure ROMIO such that `MPI_File_sync()` is also a synchronizing collective. To enable this behavior, one can set the following ROMIO hint through an `MPI_Info` object or within a ROMIO hints file:

```
romio_synchronizing_flush true
```

This configuration can be useful to applications that only call `MPI_File_sync()` once rather than execute a full sync-barrier-sync construct.

This hint was added starting with the ROMIO version available in the MPICH v4.0 release.

ROMIO Data Visibility Hint

Starting with the ROMIO version available in the MPICH v4.1 release, the read-only hint `romio_visibility_immediate` was added to inform the caller as to whether it is necessary to call `MPI_File_sync` to manage data consistency.

One can query the `MPI_Info` associated with a file. If this hint is defined and if its value is `true`, then the underlying file system does not require the sync-barrier-sync construct in order for a process to read data written by another process. Newly written data is visible to other processes as soon as the writer process returns from its write call. If the value of the hint is `false`, or if the hint is not defined in the `MPI_Info` object, then a sync-barrier-sync construct is required.

When using UnifyFS, an application must call `MPI_File_sync()` in all situations where the MPI standard requires it. However, since a sync-barrier-sync construct is costly on some file systems, and because POSIX-complaint file systems may not require it for correctness, one can use this hint to conditionally call `MPI_File_sync()` only when required by the underlying file system.

4.3.2 File Locking

ROMIO requires file locking with `fcntl()` to implement various functionality. Since `fcntl()` is not supported in UnifyFS, one must avoid any ROMIO features that require file locking.

MPI-I/O Atomic File Consistency

ROMIO uses `fcntl()` to implement atomic file consistency. One cannot use atomic mode when using UnifyFS. Provided an application still executes correctly without atomic mode, one can disable it by calling:

```
MPI_File_set_atomicity(fh, 0)
```

Atomic mode is often disabled by default in ROMIO.

Data Sieving

ROMIO uses `fcntl()` to support its data sieving optimization. One must disable ROMIO data sieving when using UnifyFS. To disable data sieving, one can set the following ROMIO hints:

```
romio_ds_read disable
romio_ds_write disable
```

These hints can be set in the `MPI_Info` object when opening a file, e.g.,:

```
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info, "romio_ds_read",  "disable");
MPI_Info_set(info, "romio_ds_write", "disable");
MPI_File_open(comm, filename, amode, info, &fh);
MPI_Info_free(&info);
```

or the hints may be listed in a ROMIO hints file, e.g.,:

```
>>: cat romio_hints.txt
romio_ds_read disable
romio_ds_write disable

>>: export ROMIO_HINTS="romio_hints.txt"
```

MPI-I/O Shared File Pointers

ROMIO uses file locking to support MPI-I/O shared file pointers. One cannot use MPI-I/O shared file pointers when using UnifyFS. Functions that use shared file pointers include:

```
MPI_File_write_shared()
MPI_File_read_shared()
MPI_File_write_ordered()
MPI_File_read_ordered()
```

4.4 HDF5 Limitations

HDF5 uses MPI-I/O. In addition to restrictions that are specific to HDF5, one must follow any restrictions associated with the underlying MPI-I/O implementation. In particular, if the MPI library uses ROMIO for its MPI-I/O implementation, one should adhere to any limitations noted above for both ROMIO and MPI-I/O in general.

4.4.1 Data Consistency

In HDF5, `H5Fflush()` calls `MPI_File_sync()` and `H5Fclose()` calls `MPI_File_close()`. When running HDF5 on ROMIO or on other MPI-I/O implementations where these MPI routines flush newly written data to UnifyFS, one must invoke these HDF5 functions to properly manage data consistency.

When using HDF5 with the MPI-I/O driver, for a process to read data written by another process without closing the HDF file, the writer must call `H5Fflush()` after writing its data. There must then be a synchronization operation between the writer and reader processes. Finally, the reader must call `H5Fflush()` after the synchronization operation with the writer. This executes the sync-barrier-sync construct as required by MPI. For example:

```
H5Fflush(...)
MPI_Barrier(...)
H5Fflush(...)
```

If `MPI_File_sync()` is a synchronizing collective, as with when enabling the `romio_synchronizing_flush` MPI-I/O hint, then a single call to `H5Fflush()` suffices to accomplish the sync-barrier-sync construct:

```
H5Fflush(...)
```

HDF5 FILE_SYNC

Starting with the HDF5 v1.13.2 release, HDF can be configured to call `MPI_File_sync()` after every HDF collective write operation. This configuration is enabled automatically if MPI-I/O defines the `romio_visibility_immediate` hint as `false`. One can also enable this option manually by setting the environment variable `HDF5_DO_MPI_FILE_SYNC=1`. Enabling this option can decrease write performance since it may induce more file flush operations than necessary.

4.5 PnetCDF Limitations

PnetCDF applications can utilize UnifyFS, and the semantics of the PnetCDF API align well with UnifyFS constraints.

PnetCDF uses MPI-IO to read and write files. In addition to any restrictions required when using UnifyFS with PnetCDF, one must follow any recommendations regarding UnifyFS and the underlying MPI-IO implementation.

4.5.1 Data Consistency

PnetCDF parallelizes access to NetCDF files using MPI. An MPI communicator is passed as an argument when opening a file. Any collective call in PnetCDF is global across the process group associated with the communicator used to open the file.

PnetCDF follows the data consistency model defined by MPI-IO. Specifically, from its documentation about PnetCDF data consistency:

Note: PnetCDF follows the same parallel I/O data consistency as MPI-IO standard.

If users would like PnetCDF to enforce a stronger consistency, they should add `NC_SHARE` flag when open/create the file. By doing so, PnetCDF adds `MPI_File_sync()` after each MPI I/O calls.

If `NC_SHARE` is not set, then users are responsible for their desired data consistency. To enforce a stronger consistency, users can explicitly call `ncmpi_sync()`. In `ncmpi_sync()`, `MPI_File_sync()` and `MPI_Barrier()` are called.

Upon inspection of the implementation of the PnetCDF v1.12.3 release, the following PnetCDF functions include the following calls:

```
ncmpio_file_sync
 - calls MPI_File_sync(ncp->independent_fh)
 - calls MPI_File_sync(ncp->collective_fh)
 - calls MPI_Barrier

ncmpio_sync
 - calls ncmpio_file_sync

ncmpi__enddef
 - calls ncmpio_file_sync if NC_doFsync (NC_SHARE)

ncmpio_enddef
 - calls ncmpi__enddef

ncmpio_end_indep_data
 - calls MPI_File_sync if NC_doFsync (NC_SHARE)

ncmpio_redef
  - does *NOT* call ncmpio_file_sync

ncmpio_close
 - calls ncmpio_file_sync if NC_doFsync (NC_SHARE)
 - calls MPI_File_close (MPI_File_close calls MPI_File_sync by MPI standard)
```

If a program must read data written by another process, PnetCDF users must do one of the following when using UnifyFS:

1) Add explicit calls to `ncmpi_sync()` after writing and before reading.
2) Set `UNIFYFS_CLIENT_WRITE_SYNC=1`, in which case each POSIX write operation invokes a flush.
3) Use `NC_SHARE` when opening files so that the PnetCDF library invokes `MPI_File_sync()` and `MPI_Barrier()` calls after its MPI-IO operations.

Of these options, it is recommended that one add `ncmpi_sync()` calls where necessary. Setting `UNIFYFS_CLIENT_WRITE_SYNC=1` is convenient since one does not need to change the application, but it may have a larger impact on performance. Opening or creating a file with `NC_SHARE` may work for some applications, but it

depends on whether the PnetCDF implementation internally calls `MPI_File_sync()` at all appropriate places, which is not guaranteed.

A number of PnetCDF calls invoke write operations on the underlying file. In addition to the `ncmpi_put_*` collection of calls that write data to variables or attributes, `ncmpi_enddef` updates variable definitions, and it can fill variables with default values. Users may also explicitly fill variables by calling `ncmpi_fill_var_rec()`. One must ensure necessary `ncmpi_sync()` calls are placed between any fill and write operations in case they happen to write to overlapping regions of a file.

Note that `ncmpi_sync()` calls `MPI_File_sync()` and `MPI_Barrier()`, but it does not call `MPI_File_sync()` again after calling `MPI_Barrier()`. To execute a full sync-barrier-sync construct, one technically must call `ncmpi_sync()` twice:

```
// to accomplish sync-barrier-sync
ncmpi_sync(...) // call MPI_File_sync and MPI_Barrier
ncmpi_sync(...) // call MPI_File_sync again
```

When using UnifyFS, a single call to `ncmpi_sync()` should suffice since UnifyFS does not (currently) require the second call to `MPI_File_sync()` as noted above.

CHAPTER

FIVE

BUILD UNIFYFS

This section describes how to build UnifyFS and its dependencies, and what *configure time options* are available.

There are three build options:

- build both UnifyFS and dependencies with Spack,
- build the dependencies with Spack, but build UnifyFS with autotools
- build the dependencies with a bootstrap script, and build UnifyFS with autotools

5.1 Build UnifyFS and Dependencies with Spack

One may install UnifyFS and its dependencies with Spack. If you already have Spack, make sure you have the latest release. If you use a clone of the Spack develop branch, be sure to pull the latest changes.

Warning: Thallium, Mochi Suite, and SDS Repo Users

The available and UnifyFS-compatible Mochi-Margo versions that are in the `mochi-margo` Spack package may not match up with the latest/default versions in the Mochi Suite, SDS Repo, and `mochi-thallium` Spack packages. It is likely that a different version of `mochi-margo` will need to be specified in the install command of UnifyFS (E.g.: `spack install unifyfs ^mochi-margo@0.13.1`).

5.1.1 Install Spack

```
$ git clone https://github.com/spack/spack
$ # create a packages.yaml specific to your machine
$ . spack/share/spack/setup-env.sh
```

Use Spack's shell support to add Spack to your `PATH` and enable use of the `spack` command.

5.1.2 Build and Install UnifyFS

```
$ spack install unifyfs
$ spack load unifyfs
```

If the most recent changes on the development branch ('dev') of UnifyFS are desired, then do `spack install unifyfs@develop`.

Include or remove variants with Spack when installing UnifyFS when a custom build is desired. Run `spack info unifyfs` for more information on available variants.

Table 1: UnifyFS Build Variants

Variant	Command (`spack install <package>`)	Default	Description
Auto-mount	`unifyfs+auto-mount`	True	Enable transparent mounting
Boostsys	`unifyfs+boostsys`	False	Have Mercury use Boost
Fortran	`unifyfs+fortran`	True	Enable Fortran support
PMI	`unifyfs+pmi`	False	Enable PMI2 support
PMIx	`unifyfs+pmix`	False	Enable PMIx support
Preload	`unifyfs+preload`	False	Enable LD_PRELOAD library support
SPath	`unifyfs+spath`	True	Normalize relative paths

Attention: The initial install could take a while as Spack will install build dependencies (autoconf, automake, m4, libtool, and pkg-config) as well as any dependencies of dependencies (cmake, perl, etc.) if you don't already have these dependencies installed through Spack or haven't told Spack where they are locally installed on your system (i.e., through a custom packages.yaml). Run `spack spec -I unifyfs` before installing to see what Spack is going to do.

5.2 Build Dependencies with Spack, Build UnifyFS with Autotools

One can install the UnifyFS dependencies with Spack and build UnifyFS with autotools. This is useful if one needs to modify the UnifyFS source code between builds. Take advantage of Spack Environments to streamline this process.

5.2.1 Build the Dependencies

Once Spack is installed on your system (see *above*), the UnifyFS dependencies can then be installed.

```
$ spack install gotcha
$ spack install mochi-margo@0.13.1 ^libfabric fabrics=rxm,sockets,tcp
$ spack install spath~mpi
```

Tip: Run `spack install --only=dependencies unifyfs` to install all UnifyFS dependencies without installing UnifyFS itself.

Keep in mind this will also install all the build dependencies and dependencies of dependencies if you haven't already installed them through Spack or told Spack where they are locally installed on your system via a packages.yaml.

5.2.2 Build UnifyFS

Download the latest UnifyFS release from the Releases page or clone the develop branch ('dev') from the UnifyFS repository https://github.com/LLNL/UnifyFS.

Load the dependencies into your environment and then configure and build UnifyFS from its source code directory.

```
$ spack load gotcha
$ spack load argobots
$ spack load mercury
$ spack load mochi-margo
$ spack load spath
$
$ gotcha_install=$(spack location -i gotcha)
$ spath_install=$(spack location -i spath)
$
$ ./autogen.sh # skip if using release tarball
$ ./configure --prefix=/path/to/install --with-gotcha=${gotcha_install} --with-spath=$
↪{spath_install}
$ make
$ make install
```

Alternatively, UnifyFS can be configured using `CPPFLAGS` and `LDFLAGS`:

```
$ ./configure --prefix=/path/to/install CPPFLAGS="-I${gotcha_install}/include -I{spath_
↪install}/include" LDFLAGS="-L${gotcha_install}/lib64 -L${spath_install}/lib64
```

To see all available build configuration options, run `./configure --help` after `./autogen.sh` has been run.

5.3 Build Dependencies with Bootstrap and Build UnifyFS with Autotools

Download the latest UnifyFS release from the Releases page or clone the develop branch ('dev') from the UnifyFS repository https://github.com/LLNL/UnifyFS.

5.3.1 Build the Dependencies

UnifyFS requires MPI, GOTCHA, Margo and OpenSSL. References to these dependencies can be found on the *UnifyFS Dependencies* page.

A bootstrap.sh script in the UnifyFS source distribution downloads and installs all dependencies. Simply run the script in the top level directory of the source code.

```
$ ./bootstrap.sh
```

Note: UnifyFS requires automake version 1.15 or newer in order to build.

Before building the UnifyFS dependencies, the bootstrap.sh script will check the system's current version of automake and attempt to build the autotools suite if an older version is detected.

5.3.2 Build UnifyFS

After bootstrap.sh installs the dependencies, it prints the commands one needs to execute to build UnifyFS. As an example, the commands may look like:

```
$ export PKG_CONFIG_PATH=$INSTALL_DIR/lib/pkgconfig:$INSTALL_DIR/lib64/pkgconfig:$PKG_
↪CONFIG_PATH
$ export LD_LIBRARY_PATH=$INSTALL_DIR/lib:$INSTALL_DIR/lib64:$LD_LIBRARY_PATH
$ ./autogen.sh # skip if using release tarball
$ ./configure --prefix=/path/to/install CPPFLAGS=-I/path/to/install/include LDFLAGS="-L/
↪path/to/install/lib -L/path/to/install/lib64"
$ make
$ make install
```

Alternatively, UnifyFS can be configured using `--with` options:

```
$ ./configure --prefix=/path/to/install --with-gotcha=$INSTALL_DIR --with-spath=$INSTALL_
↪DIR
```

To see all available build configuration options, run `./configure --help` after `./autogen.sh` has been run.

Note: On Cray systems, the detection of MPI compiler wrappers requires passing the following flags to the configure command: `MPICC=cc MPIFC=ftn`

5.4 Configure Options

When building UnifyFS with autotools, a number of options are available to configure its functionality.

5.4.1 Fortran

To use UnifyFS in Fortran applications, pass the `--enable-fortran` option to configure. Note that only GCC Fortran (i.e., gfortran) is known to work with UnifyFS. There is an open ifort_issue with the Intel Fortran compiler as well as an xlf_issue with the IBM Fortran compiler.

Note: UnifyFS requires GOTCHA when Fortran support is enabled

---

5.4.2 GOTCHA

GOTCHA is the preferred method for I/O interception with UnifyFS, but it is not available on all platforms. If GOTCHA is not available on your target system, you can omit it during UnifyFS configuration by using the `--without-gotcha` configure option. Without GOTCHA, static linker wrapping is required for I/O interception, see *Link with the UnifyFS library*.

Warning: UnifyFS requires GOTCHA for dynamic I/O interception of MPI-IO functions. If UnifyFS is configured using `--without-gotcha`, support will be lost for MPI-IO (and as a result, HDF5) applications.

5.4.3 HDF5

UnifyFS includes example programs that use HDF5. If HDF5 is not available on your target system, it can be omitted during UnifyFS configuration by using the `--without-hdf5` configure option.

5.4.4 PMI2/PMIx Key-Value Store

When available, UnifyFS uses the distributed key-value store capabilities provided by either PMI2 or PMIx. To enable this support, pass either the `--enable-pmi` or `--enable-pmix` option to configure. Without PMI support, a distributed file system accessible to all servers is required.

5.4.5 SPATH

The spath library can be optionally used to normalize relative paths (e.g., ones containing ".", "..", and extra or trailing "/") and enable the support of using relative paths within an application. To enable, use the `--with-spath` configure option or provide the appropriate `CPPFLAGS` and `LDFLAGS` at configure time.

5.4.6 Transparent Mounting for MPI Applications

MPI applications written in C or C++ may take advantage of the UnifyFS transparent mounting capability. With transparent mounting, calls to `unifyfs_mount()` and `unifyfs_unmount()` are automatically performed during `MPI_Init()` and `MPI_Finalize()`, respectively. Transparent mounting always uses `/unifyfs` as the namespace mountpoint. To enable transparent mounting, use the `--enable-mpi-mount` configure option.

5.4.7 Intercepting I/O Calls from Shell Commands

An optional preload library can be used to intercept I/O function calls made by shell commands, which allows one to run shell commands as a client to interact with UnifyFS. To build this library, use the `--enable-preload` configure option. At run time, one should start the UnifyFS server as normal. One must then set the `LD_PRELOAD` environment variable to point to the installed library location within the shell. For example, a bash user can set:

```
$ export LD_PRELOAD=/path/to/install/lib/libunifyfs_preload_gotcha.so
```

One can then interact with UnifyFS through subsequent shell commands, such as:

```
$ touch /unifyfs/file1
$ cp -pr /unifyfs/file1 /unifyfs/file2
$ ls -l /unifyfs/file1
$ stat /unifyfs/file1
$ rm /unifyfs/file1
```

The default mountpoint used is `/unifyfs`. This can be changed by setting the `UNIFYFS_PRELOAD_MOUNTPOINT` environment variable.

Note: Due to the variety and variation of I/O functions that may be called by different commands, there is no guarantee that a given invocation is supported under UnifyFS semantics. This feature is experimental, and it should be used at one's own risk.

CHAPTER

SIX

INTEGRATE THE UNIFYFS API

This section describes how to use the UnifyFS API in an application.

Transparent Mount Caveat

MPI applications that take advantage of the *transparent mounting* feature (through configuring with `--enable-mpi-mount` or with `+auto-mount` through Spack) do not need to be modified in any way in order to use UnifyFS. Move on to the *Link with the UnifyFS library* section next as this step can be skipped.

Attention: Fortran Compatibility

`unifyfs_mount` and `unifyfs_unmount` are usable with GFortran. There is a known ifort_issue with the Intel Fortran compiler as well as an xlf_issue with the IBM Fortran compiler. Other Fortran compilers are currently unknown.

If using fortran, when *installing UnifyFS* with Spack, include the `+fortran` variant, or configure UnifyFS with the `--enable-fortran` option if building manually.

6.1 Include the UnifyFS Header

In C or C++ applications, include `unifyfs.h`. See writeread.c for a full example.

Listing 1: C

```c
#include <unifyfs.h>
```

In Fortran applications, include `unifyfsf.h`. See writeread.f90 for a full example.

Listing 2: Fortran

```
include 'unifyfsf.h'
```

6.2 Mounting

UnifyFS implements a file system in user space, which the system has no knowledge about. The UnifyFS library intecepts and handles I/O calls whose path matches a prefix that is defined by the user. Calls corresponding to matching paths are handled by UnifyFS and all other calls are forwarded to the original I/O routine.

To use UnifyFS, the application must register the path that the UnifyFS library should intercept by making a call to `unifyfs_mount`. This must be done once on each client process, and it must be done before the client process attempts to access any UnifyFS files.

For instance, to use UnifyFS on all path prefixes that begin with `/unifyfs` this would require a:

Listing 3: C

```
int rc = unifyfs_mount("/unifyfs", rank, rank_num);
```

Listing 4: Fortran

```
call UNIFYFS_MOUNT('/unifyfs', rank, size, ierr);
```

Here, `/unifyfs` is the path prefix for UnifyFS to intercept. The `rank` parameter specifies the MPI rank of the calling process. The `size` parameter specifies the number of MPI ranks in the user job.

6.3 Unmounting

When the application is done using UnifyFS, it should call `unifyfs_unmount`.

Listing 5: C

```
unifyfs_unmount();
```

Listing 6: Fortran

```
call UNIFYFS_UNMOUNT(ierr);
```

CHAPTER

SEVEN

LINK WITH THE UNIFYFS LIBRARY

This section describes how to link an application with the UnifyFS library. The UnifyFS library contains symbols for the UnifyFS API, like unifyfs_mount, as well as wrappers for I/O routines, like open, write, and close. In the examples below, replace <unifyfs> with the path to your UnifyFS install.

7.1 Static link

For a static link, UnifyFS utilizes the `--wrap` feature of the `ld` command. One must specify a `--wrap` option for every I/O call that is wrapped, for which there are many. To make this easier, UnifyFS installs a unifyfs-config script that one should invoke to specify those flags, e.g.,

```
$ mpicc -o test_write \
      `<unifyfs>/bin/unifyfs-config --pre-ld-flags` \
      test_write.c \
      `<unifyfs>/bin/unifyfs-config --post-ld-flags`
```

7.2 Dynamic link

A build of UnifyFS includes two different shared libraries. Which one you should link against depends on your application. If you wish to take advantage of the UnifyFS auto-mount feature (assuming the feature was enabled at compile-time), then you should link against `libunifyfs_mpi_gotcha.so`. If you are not building an MPI-enabled application, or if you want explicit control over when UnifyFS filesystem is mounted and unmounted, then link against `libunifyfs_gotcha.so`. In this case, you will also have to add calls to `unifyfs_mount` and `unifyfs_unmount` in the appropriate locations in your code. See *Integrate the UnifyFS API*.

To intercept I/O calls using gotcha, use the following syntax to link an application:

7.2.1 C

For code that uses the auto-mount feature:

```
$ mpicc -o test_write test_write.c \
    -L<unifyfs>/lib -lunifyfs_mpi_gotcha
```

For code that explicitly calls `unifyfs_mount` and `unifyfs_unmount`:

```
$ mpicc -o test_write test_write.c \
    -I<unifyfs>/include -L<unifyfs>/lib -lunifyfs_gotcha
```

Note the use of the `-I` option so that the compiler knows where to find the `unifyfs.h` header file.

7.2.2 Fortran

For code that uses the auto-mount feature:

```
$ mpif90 -o test_write test_write.F \
    -L<unifyfs>/lib -lunifyfs_mpi_gotcha
```

For code that explicitly calls `unifyfs_mount` and `unifyfs_unmount`:

```
$ mpif90 -o test_write test_write.F \
    -I<unifyfs>/include -L<unifyfs>/lib -lunifyfsf -lunifyfs_gotcha
```

Note the use of the `-I` option to specify the location of the `unifyfsf.h` header. Also note the use of the `unifyfsf` library. This library provides the Fortran bindings for the `unifyfs_mount` and `unifyfs_unmount` functions.

7.3 LD_PRELOAD

In the case where an application doesn't need to be rebuilt in order to use UnifyFS (e.g., files paths are set wth arguments/configs), `LD_PRELOAD` can be used at runtime to allow for UnifyFS interception of file I/O.

```
$ srun -N2 -n8 --export=ALL,LD_PRELOAD=$UNIFYFS_LIB/libunifyfs_mpi_gotcha.so␣
↪myApplication
```

CHAPTER

EIGHT

UNIFYFS CONFIGURATION

Here, we explain how users can customize the runtime behavior of UnifyFS. In particular, UnifyFS provides the following ways to configure:

- Configuration file: `$INSTALL_PREFIX/etc/unifyfs/unifyfs.conf`
- Environment variables
- Command line options to `unifyfsd`

All configuration settings have corresponding environment variables, but only certain settings have command line options. When defined via multiple methods, the command line options have the highest priority, followed by environment variables, and finally config file options from `unifyfs.conf`.

The system-wide configuration file is used by default when available. However, users can specify a custom location for the configuration file using the `-f` command-line option to `unifyfsd` (see below). There is a sample `unifyfs.conf` file in the installation directory under `etc/unifyfs/`. This file is also available in the `extras` directory in the source repository.

The unified method for providing configuration control is adapted from CONFIGURATOR. Configuration settings are grouped within named sections, and each setting consists of a key-value pair with one of the following types:

- `BOOL`: `0|1`, `y|n`, `Y|N`, `yes|no`, `true|false`, `on|off`
- `FLOAT`: scalars convertible to C double, or compatible expression
- `INT`: scalars convertible to C long, or compatible expression
- `STRING`: quoted character string

8.1 System Configuration File (`unifyfs.conf`)

`unifyfs.conf` specifies the system-wide configuration options. The file is written in INI language format, as supported by the inih parser.

The config file has several sections, each with a few key-value settings. In this description, we use `section.key` as shorthand for the name of a given section and key.

Table 1: `[unifyfs]` section - main configuration settings

Key	Type	Description
cleanup	BOOL	cleanup storage on server exit (default: off)
configfile	STRING	path to custom configuration file
daemonize	BOOL	enable server daemonization (default: off)
mountpoint	STRING	mountpoint path prefix (default: /unifyfs)

Table 2: [client] section - client settings

Key	Type	Description
cwd	STRING	effective starting current working directory
excl_private	BOOL	create node-local private files when given O_EXCL (default: on)
fsync_persist	BOOL	persist data to storage on fsync() (default: on)
local_extents	BOOL	service reads from local data (default: off)
max_files	INT	maximum number of open files per client process (default: 128)
node_local_extents	BOOL	service reads from node local data for laminated files (default: off)
super_magic	BOOL	whether to return UNIFYFS (on) or TMPFS (off) statfs magic (default: on)
unlink_usecs	INT	number of microseconds to sleep after initiating unlink rpc (default: 0)
write_index_size	INT	maximum size (B) of memory buffer for storing write log metadata
write_sync	BOOL	sync data to server after every write (default: off)

The client.cwd setting is used to emulate the behavior one expects when changing into a working directory before starting a job and then using relative file names within the application. If set, the application changes its working directory to the value specified in client.cwd when unifyfs_mount() is called. The value specified in client.cwd must be within the directory space of the UnifyFS mount point.

Enabling the client.local_extents optimization may significantly improve read performance for extents written by the same process. However, it should not be used by applications in which different processes write to the same byte offset within a file, nor should it be used with applications that truncate files.

Table 3: [log] section - logging settings

Key	Type	Description
dir	STRING	path to directory to contain server log file
file	STRING	log file base name (rank will be appended)
on_error	BOOL	increase log verbosity upon encountering an error (default: off)
verbosity	INT	logging verbosity level [0-5] (default: 0)

Table 4: [logio] section - log-based write data storage settings

Key	Type	Description
chunk_size	INT	data chunk size (B) (default: 4 MiB)
shmem_size	INT	maximum size (B) of data in shared memory (default: 256 MiB)
spill_size	INT	maximum size (B) of data in spillover file (default: 4 GiB)
spill_dir	STRING	path to spillover data directory

Table 5: [margo] section - margo server NA settings

Key	Type	Description
tcp	BOOL	Use TCP for server-to-server rpcs (default: on, turn off to enable libfabric RMA)
client_timeout	INT	timeout in milliseconds for rpcs between client and server (default: 5000)
server_timeout	INT	timeout in milliseconds for rpcs between servers (default: 15000)

Table 6: `[runstate]` section - server runstate settings

Key	Type	Description
dir	STRING	path to directory to contain server-local state

Table 7: `[server]` section - server settings

Key	Type	Description
hostfile	STRING	path to server hostfile
init_timeout	INT	timeout in seconds to wait for servers to be ready for clients (default: 120)
local_extents	BOOL	use server extents to service local reads without consulting file owner

Table 8: `[sharedfs]` section - server shared files settings

Key	Type	Description
dir	STRING	path to directory to contain server shared files

8.2 Environment Variables

All environment variables take the form `UNIFYFS_SECTION_KEY`, except for the `[unifyfs]` section, which uses `UNIFYFS_KEY`. For example, the setting `log.verbosity` has a corresponding environment variable named `UNIFYFS_LOG_VERBOSITY`, while `unifyfs.mountpoint` corresponds to `UNIFYFS_MOUNTPOINT`.

8.3 Command Line Options

For server command line options, we use `getopt_long()` format. Thus, all command line options have long and short forms. The long form uses `--section-key=value`, while the short form `-<optchar> value`, where the short option character is given in the below table.

Note that for configuration options of type BOOL, the value is optional. When not provided, the `true` value is assumed. If the short form option is used, the value must immediately follow the option character (e.g., `-Cyes`).

Table 9: `unifyfsd` command line options

LongOpt	ShortOpt
`--unifyfs-cleanup`	`-C`
`--unifyfs-configfile`	`-f`
`--unifyfs-daemonize`	`-D`
`--log-verbosity`	`-v`
`--log-file`	`-l`
`--log-dir`	`-L`
`--runstate-dir`	`-R`
`--server-hostfile`	`-H`
`--sharedfs-dir`	`-S`
`--server-init_timeout`	`-t`

CHAPTER

NINE

RUN UNIFYFS

This section describes the mechanisms to start and stop the UnifyFS server processes within a job allocation.

Overall, the steps to run an application with UnifyFS include:

1. Allocate nodes using the system resource manager (i.e., start a job)
2. Update any desired UnifyFS server configuration settings
3. Start UnifyFS servers on each allocated node using `unifyfs`
4. Run one or more UnifyFS-enabled applications
5. Terminate the UnifyFS servers using `unifyfs`

9.1 Start UnifyFS

First, one must start the UnifyFS server process (`unifyfsd`) on the nodes in the job allocation. UnifyFS provides the `unifyfs` command line utility to simplify this action on systems with supported resource managers. The easiest way to determine if you are using a supported system is to run `unifyfs start` within an interactive job allocation. If no compatible resource management system is detected, the utility reports an error message to that effect.

In `start` mode, the `unifyfs` utility automatically detects the allocated nodes and launches a server on each node. For example, the following script could be used to launch the `unifyfsd` servers with a customized configuration. On systems with resource managers that propagate environment settings to compute nodes, the environment variables override any settings in `/etc/unifyfs/unifyfs.conf`. See *UnifyFS Configuration* for further details on customizing the UnifyFS runtime configuration.

```
#!/bin/bash

# spillover data to node-local ssd storage
export UNIFYFS_LOGIO_SPILL_DIR=/mnt/ssd/$USER/data

# store server logs in job-specific scratch area
export UNIFYFS_LOG_DIR=$JOBSCRATCH/logs

unifyfs start --share-dir=/path/to/shared/file/system
```

`unifyfs` provides command-line options to select the shared file system path, choose the client mountpoint, and control stage-in and stage-out of files. The full usage for `unifyfs` is as follows:

```
[prompt]$ unifyfs --help

Usage: unifyfs <command> [options...]

<command> should be one of the following:
  start       start the UnifyFS server daemons
  terminate   terminate the UnifyFS server daemons

Common options:
  -d, --debug               enable debug output
  -h, --help                print usage

Command options for "start":
  -e, --exe=<path>           [OPTIONAL] <path> where unifyfsd is installed
  -m, --mount=<path>         [OPTIONAL] mount UnifyFS at <path>
  -s, --script=<path>        [OPTIONAL] <path> to custom launch script
  -t, --timeout=<sec>        [OPTIONAL] wait <sec> until all servers become ready
  -S, --share-dir=<path>     [REQUIRED] shared file system <path> for use by servers
  -c, --cleanup              [OPTIONAL] clean up the UnifyFS storage upon server exit
  -i, --stage-in=<manifest>  [OPTIONAL] stage in file(s) listed in <manifest> file
  -P, --stage-parallel       [OPTIONAL] use parallel stage-in
  -T, --stage-timeout=<sec>  [OPTIONAL] timeout for stage-in operation

Command options for "terminate":
  -o, --stage-out=<manifest> [OPTIONAL] stage out file(s) listed in <manifest> on␣
↪termination
  -P, --stage-parallel       [OPTIONAL] use parallel stage-out
  -T, --stage-timeout=<sec>  [OPTIONAL] timeout for stage-out operation
  -s, --script=<path>        [OPTIONAL] <path> to custom termination script
  -S, --share-dir=<path>     [REQUIRED for --stage-out] shared file system <path> for␣
↪use by servers
```

After UnifyFS servers have been successfully started, you may run your UnifyFS-enabled applications as you normally would (e.g., using mpirun). Only applications that explicitly call `unifyfs_mount()` and access files under the specified mountpoint prefix will utilize UnifyFS for their I/O. All other applications will operate unchanged.

9.2 Stop UnifyFS

After all UnifyFS-enabled applications have completed running, use `unifyfs terminate` to terminate the servers. Pass the `--cleanup` option to `unifyfs start` to have the servers remove temporary data locally stored on each node after termination.

9.3 Resource Manager Job Integration

UnifyFS includes optional support for integrating directly with compatible resource managers to automatically start and stop servers at the beginning and end of a job when requested by users. Resource manager integration requires administrator privileges to deploy.

Currently, only IBM's Platform LSF with Cluster System Manager (LSF-CSM) is supported. LSF-CSM is the resource manager on the CORAL2 IBM systems at ORNL and LLNL. The required job prologue and epilogue scripts, along with a README documenting the installation instructions, is available within the source repository at `util/scripts/lsfcsm`.

Support for the SLURM resource manager is under development.

9.4 Transferring Data In and Out of UnifyFS

Data can be transferred in/out of UnifyFS during server startup and termination, or at any point during a job using two stand-alone applications.

9.4.1 Transfer at Server Start/Terminate

The transfer subsystem within UnifyFS can be invoked by providing the `-i|--stage-in` option to `unifyfs start` to transfer files into UnifyFS:

```
$ unifyfs start --stage-in=/path/to/input/manifest/file --share-dir=/path/to/shared/file/
↪system
```

and/or by providing the `-o|--stage-out` option to `unifyfs terminate` to transfer files out of UnifyFS:

```
$ unifyfs terminate --stage-out=/path/to/output/manifest/file --share-dir=/path/to/
↪shared/file/system
```

The argument to both staging options is the path to a manifest file that contains the source and destination file pairs. Both stage-in and stage-out also require passing the `-S|--share-dir=<path>` option.

Manifest File

UnifyFS's file staging functionality requires a manifest file in order to move data.

The manifest file contains one or more file copy requests. Each line in the manifest corresponds to one transfer request, and it contains both the source and destination file paths. Directory copies are currently not supported.

Each line is formatted as: `<source filename> <whitespace> <destination filename>`.

If either of the filenames contain whitespace or special characters, then both filenames should be surrounded by double-quote characters (") (ASCII character 34 decimal). The double-quote and linefeed end-of-line characters are not supported in any filenames used in a manifest file. Any other characters are allowed, including control characters. If a filename contains any characters that might be misinterpreted, we suggest enclosing the filename in double-quotes. Comment lines are also allowed, and are indicated by beginning a line with the # character.

Here is an example of a valid stage-in manifest file:

```
$ [prompt] cat example_stage_in.manifest

/scratch/users/me/input_data/input_1.dat /unifyfs/input/input_1.dat
# example comment line
/home/users/me/configuration/run_12345.conf /unifyfs/config/run_12345.conf
"/home/users/me/file with space.dat" "/unifyfs/file with space.dat"
```

9.4.2 Transfer During Job

Data can also be transferred in/out of UnifyFS using the `unifyfs-stage` helper program. This is the same program used internally by `unifyfs` to provide file staging during server startup and termination.

The helper program can be invoked at any time while the UnifyFS servers are up and responding to requests. This allows for bringing in new input and/or transferring results out to be verified before the job terminates.

UnifyFS Stage Executable

The `unifyfs-stage` program is installed in the same directory as the `unifyfs` utility (i.e., `$UNIFYFS_INSTALL/bin`).

A manifest file (see *above*) needs to be provided as an argument to use this approach.

```
[prompt]$ unifyfs-stage --help

Usage: unifyfs-stage [OPTION]... <manifest file>

Transfer files between unifyfs volume and external file system.
The <manifest file> should contain list of files to be transferred,
and each line should be formatted as

  /source/file/path /destination/file/path

OR in the case of filenames with spaces or special characters:

  "/source/file/path" "/destination/file/path"

One file per line; Specifying directories is not currently supported.

Available options:
  -c, --checksum           Verify md5 checksum for each transfer
                           (default: off)
  -h, --help               Print usage information
  -m, --mountpoint=<mnt>   Use <mnt> as UnifyFS mountpoint
                           (default: /unifyfs)
  -p, --parallel           Transfer all files concurrently
                           (default: off, use sequential transfers)
  -s, --skewed             Use skewed data distribution for stage-in
                           (default: off, use balanced distribution)
  -S, --status-file=<path> Create stage status file at <path>
  -v, --verbose            Print verbose information
                           (default: off)

By default, each file in the manifest will be transferred in sequence (i.e.,
```

(continues on next page)

(continued from previous page)

```
only a single file will be in transfer at any given time). If the
'-p, --parallel' option is specified, files in the manifest will be
transferred concurrently. The number of concurrent transfers is limited by
the number of parallel ranks used to execute unifyfs-stage.
```

Examples:

Listing 1: Sequential Transfer using a Single Client

```
$ srun -N 1 -n 1 unifyfs-stage $MY_MANIFEST_FILE
```

Listing 2: Parallel Transfer using 8 Clients (up to 8 concurrent file transfers)

```
$ srun -N 4 -n 8 unifyfs-stage --parallel $MY_MANIFEST_FILE
```

9.5 UnifyFS LS Executable

The `unifyfs-ls` program is installed in the same directory as the `unifyfs` utility (i.e., `$UNIFYFS_INSTALL/bin`). This tool will provide information about any files the *local* server process knows about. Users may find this helpful when debugging their applications and want to know if the files they think are being managed by UnifyFS really are.

```
[prompt]$ unifyfs-ls --help
Usage:
  unifyfs-ls [ -v | --verbose ] [ -m <dir_name> | --mount_point_dir=<dir_name> ]

  -v | --verbose: show verbose information(default: 0)
  -m | --mount_point: the location where unifyfs is mounted (default: /unifyfs)
```

CHAPTER

TEN

EXAMPLE PROGRAMS

There are several examples available on ways to use UnifyFS. These examples build into static, GOTCHA, and pure POSIX (not linked with UnifyFS) versions depending on how they are linked. Several of the example programs are also used in the UnifyFS *intregraton testing*.

10.1 Locations of Examples

The example programs can be found in two locations, where UnifyFS is built and where UnifyFS is installed.

10.1.1 Install Location

Upon installation of UnifyFS, the example programs are installed into the `<install_prefix_dir>/libexec` directory.

Installed with Spack

The Spack installation location of UnifyFS can be found with the command `spack location -i unifyfs`.

To easily navigate to this location and find the examples, do:

```
$ spack cd -i unifyfs
$ cd libexec
```

Installed with Autotools

The autotools installation of UnifyFS will place the example programs in the `libexec` subdirectory of the path provided to `--prefix=/path/to/install` during the configure step of *building and installing*.

10.1.2 Build Location

Built with Spack

The Spack build location of UnifyFS (on a successful install) only exists when `--keep-stage` in included during installation or if the build fails. This location can be found with the command `spack location unifyfs`.

To navigate to the location of the static and POSIX examples, do:

```
$ spack install --keep-stage unifyfs
$ spack cd unifyfs
$ cd spack-build/examples/src
```

The GOTCHA examples are one directory deeper in `spack-build/examples/src/.libs`.

Note: If you installed UnifyFS with any variants, in order to navigate to the build directory you must include these variants in the `spack cd` command. E.g.:

`spack cd unifyfs+hdf5 ^hdf5~mpi`

Built with Autotools

The autotools build of UnifyFS will place the static and POSIX example programs in the `examples/src` directory and the GOTCHA example programs in the `examples/src/.libs` directory of your build directory.

Manual Build from Installed UnifyFS

On some systems, particularly those using compiler wrappers (e.g., HPE/Cray systems), the autotools build of the example programs will fail due to a longstanding issue with the way that `libtool` reorders compiler and linker flags. A Makefile suitable for manually building the examples from a previously installed version of UnifyFS is available at `examples/src/Makefile.examples`. This Makefile also serves as a good reference for how to use the UnifyFS `pkg-config` support to build and link client programs. The following commands will build the example programs using this Makefile.

```
$ cd <source_dir>/examples/src
$ make -f Makefile.examples
```

10.2 Running the Examples

In order to run any of the example programs you first need to start the UnifyFS server daemon on the nodes in the job allocation. To do this, see *Run UnifyFS*.

Each example takes multiple arguments and so each has its own `--help` option to aid in this process.

```
[prompt]$ ./write-static --help

Usage: write-static [options...]

Available options:
```

(continues on next page)

(continued from previous page)

```
-a, --library-api           use UnifyFS library API instead of POSIX I/O
                            (default: off)
-A, --aio                   use asynchronous I/O instead of read|write
                            (default: off)
-b, --blocksize=<bytes>     I/O block size
                            (default: 16 MiB)
-c, --chunksize=<bytes>     I/O chunk size for each operation
                            (default: 1 MiB)
-d, --debug                 for debugging, wait for input (at rank 0) at start
                            (default: off)
-D, --destfile=<filename>   transfer destination file name (or path) outside mountpoint
                            (default: none)
-f, --file=<filename>       target file name (or path) under mountpoint
                            (default: 'testfile')
-k, --check                 check data contents upon read
                            (default: off)
-l, --laminate              laminate file after writing all data
                            (default: off)
-L, --listio                use lio_listio instead of read|write
                            (default: off)
-m, --mount=<mountpoint>    use <mountpoint> for unifyfs
                            (default: /unifyfs)
-M, --mpiio                 use MPI-IO instead of POSIX I/O
                            (default: off)
-n, --nblocks=<count>       count of blocks each process will read|write
                            (default: 32)
-N, --mapio                 use mmap instead of read|write
                            (default: off)
-o, --outfile=<filename>    output file name (or path)
                            (default: 'stdout')
-p, --pattern=<pattern>     'n1' (N-to-1 shared file) or 'nn' (N-to-N file per process)
                            (default: 'n1')
-P, --prdwr                 use pread|pwrite instead of read|write
                            (default: off)
-r, --reuse-filename        remove and reuse the same target file name
                            (default: off)
-S, --stdio                 use fread|fwrite instead of read|write
                            (default: off)
-t, --pre-truncate=<size>   truncate file to size (B) before writing
                            (default: off)
-T, --post-truncate=<size>  truncate file to size (B) after writing
                            (default: off)
-u, --unlink                unlink target file
                            (default: off)
-U, --disable-unifyfs       do not use UnifyFS
                            (default: enable UnifyFS)
-v, --verbose               print verbose information
                            (default: off)
-V, --vecio                 use readv|writev instead of read|write
                            (default: off)
-x, --shuffle               read different data than written
                            (default: off)
```

One form of running this example could be:

```
$ srun -N4 -n4 write-static -m /unifyfs -f myTestFile
```

10.3 Producer-Consumer Workflow

UnifyFS can be used to support producer/consumer workflows where processes in a job perform loosely synchronized communication through files such as in coupled simulation/analytics workflows.

The *write.c* and *read.c* example programs can be used as a basic test in running a producer-consumer workflow with UnifyFS.

Listing 1: All hosts in allocation

```
$ # start unifyfs
$
$ # write on all hosts
$ srun -N4 -n16 write-gotcha -f testfile
$
$ # read on all hosts
$ srun -N4 -n16 read-gotcha -f testfile
$
$ # stop unifyfs
```

Listing 2: Disjoint hosts in allocation

```
$ # start unifyfs
$
$ # write on half of hosts
$ srun -N2 -n8 --exclude=$hostlist_subset1 write-gotcha -f testfile
$
$ # read on other half of hosts
$ srun -N2 -n8 --exclude=$hostlist_subset2 read-gotcha -f testfile
$
$ # stop unifyfs
```

Note: Producer/consumer support with UnifyFS has been tested using POSIX and MPI-IO APIs on x86_64 (MVAPICH) and Power 9 systems (Spectrum MPI).

These scenarios have been tested using both the same and disjoint sets of hosts as well as using a shared file and a file per process for I/O.

CHAPTER

ELEVEN

UNIFYFS API FOR I/O MIDDLEWARE

This section describes the purpose, concepts, and usage of the UnifyFS library API.

11.1 Library API Purpose

The UnifyFS library API provides a direct interface for UnifyFS configuration, namespace management, and batched file I/O and transfer operations. The library is primarily targeted for use by I/O middleware software such as HDF5 and VeloC, but is also useful for user applications needing programmatic control and interactions with UnifyFS.

Note: Use of the library API is *not* required for most applications, as UnifyFS will transparently intercept I/O operations made by the application. See *Example Programs* for examples of typical application usage.

11.2 Library API Concepts

11.2.1 Namespace (aka Mountpoint)

All UnifyFS clients provide the mountpoint prefix (e.g., "/unifyfs") that is used to distinguish the UnifyFS namespace from other file systems available to the client application. All absolute file paths that include the mountpoint prefix are treated as belonging to the associated UnifyFS namespace.

Using the library API, an application or I/O middleware system can operate on multiple UnifyFS namespaces concurrently.

11.2.2 File System Handle

All library API methods require a file system handle parameter of type `unifyfs_handle`. Users obtain a valid handle via an API call to `unifyfs_initialize()`, which specifies the mountpoint prefix and configuration settings associated with the handle.

Multiple handles can be acquired by the same client. This permits access to multiple namespaces, or different configured behaviors for the same namespace.

11.2.3 Global File Identifier

A global file identifier (gfid) is a unique integer identifier for a given absolute file path within a UnifyFS namespace. Clients accessing the exact same file path are guaranteed to obtain the same gfid value when creating or opening the file. I/O operations use the gfid to identify the target file.

Note that unlike POSIX file descriptors, a gfid is strictly a unique identifier and has no associated file state such as a current file position pointer. As such, it is valid to obtain the gfid for a file in a single process (e.g., via file creation), and then share the resulting gfid value among other parallel processes via a collective communication mechanism.

11.3 Library API Types

The file system handle type is a pointer to an opaque client structure that records the associated mountpoint and configuration.

Listing 1: File system handle type

```
/* UnifyFS file system handle (opaque pointer) */
typedef struct unifyfs_client* unifyfs_handle;
```

I/O requests take the form of a `unifyfs_io_request` structure that includes the target file gfid, the specific I/O operation (`unifyfs_ioreq_op`) to be applied, and associated operation parameters such as the file offset or user buffer and size. The structure also contains fields used for tracking the status of the request (`unifyfs_req_state`) and operation results (`unifyfs_ioreq_result`).

Listing 2: File I/O request types

```
/* I/O request structure */
typedef struct unifyfs_io_request {
    /* user-specified fields */
    void* user_buf;
    size_t nbytes;
    off_t offset;
    unifyfs_gfid gfid;
    unifyfs_ioreq_op op;

    /* status/result fields */
    unifyfs_req_state state;
    unifyfs_ioreq_result result;
} unifyfs_io_request;

/* Enumeration of supported I/O request operations */
typedef enum unifyfs_ioreq_op {
    UNIFYFS_IOREQ_NOP = 0,
    UNIFYFS_IOREQ_OP_READ,
    UNIFYFS_IOREQ_OP_WRITE,
    UNIFYFS_IOREQ_OP_SYNC_DATA,
    UNIFYFS_IOREQ_OP_SYNC_META,
    UNIFYFS_IOREQ_OP_TRUNC,
    UNIFYFS_IOREQ_OP_ZERO,
} unifyfs_ioreq_op;

/* Enumeration of API request states */
```

(continues on next page)

(continued from previous page)

```
typedef enum unifyfs_req_state {
    UNIFYFS_REQ_STATE_INVALID = 0,
    UNIFYFS_REQ_STATE_IN_PROGRESS,
    UNIFYFS_REQ_STATE_CANCELED,
    UNIFYFS_REQ_STATE_COMPLETED
} unifyfs_req_state;

/* Structure containing I/O request result values */
typedef struct unifyfs_ioreq_result {
    int error;
    int rc;
    size_t count;
} unifyfs_ioreq_result;
```

For the `unifyfs_ioreq_result` structure, successful operations will set the `rc` and `count` fields as applicable to the specific operation type. All operational failures are reported by setting the `error` field to a non-zero value corresponding the the operation failure code, which is often a POSIX `errno` value.

File transfer requests use a `unifyfs_transfer_request` structure that includes the source and destination file paths, transfer mode, and a flag indicating whether parallel file transfer should be used. Similar to I/O requests, the structure also contains fields used for tracking the request status and transfer operation result.

Listing 3: File transfer request types

```
/* File transfer request structure */
typedef struct unifyfs_transfer_request {
    /* user-specified fields */
    const char* src_path;
    const char* dst_path;
    unifyfs_transfer_mode mode;
    int use_parallel;

    /* status/result fields */
    unifyfs_req_state state;
    unifyfs_transfer_result result;
} unifyfs_transfer_request;

/* Enumeration of supported I/O request operations */
typedef enum unifyfs_transfer_mode {
    UNIFYFS_TRANSFER_MODE_INVALID = 0,
    UNIFYFS_TRANSFER_MODE_COPY, // simple copy to destination
    UNIFYFS_TRANSFER_MODE_MOVE  // copy, then remove source
} unifyfs_transfer_mode;

/* File transfer result structure */
typedef struct unifyfs_transfer_result {
    int error;
    int rc;
    size_t file_size_bytes;
    double transfer_time_seconds;
} unifyfs_transfer_result;
```

11.4 Example Library API Usage

To get started using the library API, please add the following to your client source code files that will make calls to API methods. You will also need to modify your client application build process to link with the `libunifyfs_api` library.

Listing 4: Including the API header

```
#include <unifyfs/unifyfs_api.h>
```

The common pattern for using the library API is to initialize a UnifyFS file system handle, perform a number of operations using that handle, and then release the handle. As previously mentioned, the same client process may initialize multiple file system handles and use them concurrently, either to work with multiple namespaces, or to use different configured behaviors with different handles sharing the same namespace.

11.4.1 File System Handle Initialization and Finalization

To initialize a handle to UnifyFS, the client application uses the `unifyfs_initialize()` method as shown below. This method takes the namespace mountpoint prefix and an array of optional configuration parameter settings as input parameters, and initializes the value of the passed file system handle upon success.

In the example below, the `logio.chunk_size` configuration parameter, which controls the size of the log-based I/O data chunks, is set to the value of 32768. See *UnifyFS Configuration* for further options for customizing the behavior of UnifyFS.

Listing 5: UnifyFS handle initialization

```
int n_configs = 1;
unifyfs_cfg_option chk_size = { .opt_name = "logio.chunk_size",
                                .opt_value = "32768" };

const char* unifyfs_prefix = "/my/unifyfs/namespace";
unifyfs_handle fshdl = UNIFYFS_INVALID_HANDLE;
int rc = unifyfs_initialize(unifyfs_prefix, &chk_size, n_configs, &fshdl);
```

Once all UnifyFS operation using the handle have been completed, the client application should call `unifyfs_finalize()` as shown below to release the resources associated with the handle.

Listing 6: UnifyFS handle finalization

```
int rc = unifyfs_finalize(fshdl);
```

11.4.2 File Creation, Use, and Removal

New files should be created by a single client process using `unifyfs_create()` as shown below. Note that if multiple clients attempt to create the same file, only one will succeed.

Note: Currently, the `create_flags` parameter is unused; it is reserved for future use to indicate file-specific UnifyFS behavior.

Listing 7: UnifyFS file creation

```
const char* filename = "/my/unifyfs/namespace/a/new/file";
int create_flags = 0;
unifyfs_gfid gfid = UNIFYFS_INVALID_GFID;
int rc = unifyfs_create(fshdl, create_flags, filename, &gfid);
```

Existing files can be opened by any client process using `unifyfs_open()`.

Listing 8: UnifyFS file use

```
const char* filename = "/my/unifyfs/namespace/an/existing/file";
unifyfs_gfid gfid = UNIFYFS_INVALID_GFID;
int access_flags = O_RDWR;
int rc = unifyfs_open(fshdl, access_flags, filename, &gfid);
```

When no longer required, files can be deleted using `unifyfs_remove()`.

Listing 9: UnifyFS file removal

```
const char* filename = "/my/unifyfs/namespace/an/existing/file";
int rc = unifyfs_remove(fshdl, filename);
```

11.4.3 Batched File I/O

File I/O operations in the library API use a batched request interface similar to POSIX `lio_listio()`. A client application dispatches an array of I/O operation requests, where each request identifies the target file gfid, the operation type (e.g., read, write, or truncate), and associated operation parameters. Upon successful dispatch, the operations will be executed by UnifyFS in an asynchronous manner that allows the client to overlap other computation with I/O. The client application must then explicitly wait for completion of the requests in the batch. After an individual request has been completed (or canceled by the client), the request's operation results can be queried.

When dispatching a set of requests that target the same file, there is an order imposed on the types of operations. First, all read operations are processed, followed by writes, then truncations, and finally synchronization operations. Note that this means a read request will not observe any data written in the same batch.

A simple use case for batched I/O is shown below, where the client dispatches a batch of requests including several rank-strided write operations followed by a metadata sync to make those writes visible to other clients, and then immediately waits for completion of the entire batch.

Listing 10: Synchronous Batched I/O

```
/* write and sync file metadata */
size_t n_chks = 10;
size_t chunk_size = 1048576;
size_t block_size = chunk_size * total_ranks;
size_t n_reqs = n_chks + 1;
unifyfs_io_request my_reqs[n_reqs];
for (size_t i = 0; i < n_chks; i++) {
    my_reqs[i].op = UNIFYFS_IOREQ_OP_WRITE;
    my_reqs[i].gfid = gfid;
    my_reqs[i].nbytes = chunk_size;
    my_reqs[i].offset = (off_t)((i * block_size) + (my_rank * chunk_size));
```

(continues on next page)

(continued from previous page)

```
    my_reqs[i].user_buf = my_databuf + (i * chksize);
}
my_reqs[n_chks].op = UNIFYFS_IOREQ_OP_SYNC_META;
my_reqs[n_chks].gfid = gfid;

rc = unifyfs_dispatch_io(fshdl, n_reqs, my_reqs);
if (rc == UNIFYFS_SUCCESS) {
    int waitall = 1;
    rc = unifyfs_wait_io(fshdl, n_reqs, my_reqs, waitall);
    if (rc == UNIFYFS_SUCCESS) {
        for (size_t i = 0; i < n_reqs; i++) {
            assert(my_reqs[i].result.error == 0);
        }
    }
}
```

11.4.4 Batched File Transfers

File transfer operations in the library API also use a batched request interface. A client application dispatches an array of file transfer requests, where each request identifies the source and destination file paths and the transfer mode. Two transfer modes are currently supported:

1. COPY - Copy source file to destination path.
2. MOVE - Copy source file to destination path, then remove source file.

Upon successful dispatch, the transfer operations will be executed by UnifyFS in an asynchronous manner that allows the client to overlap other computation with I/O. The client application must then explicitly wait for completion of the requests in the batch. After an individual request has been completed (or canceled by the client), the request's operation results can be queried.

A simple use case for batched transfer is shown below, where the client dispatches a batch of requests and then immediately waits for completion of the entire batch.

Listing 11: Synchronous Batched File Transfers

```
/* move output files from UnifyFS to parallel file system */
const char* destfs_prefix = "/some/parallel/filesystem/location";
size_t n_files = 3;
unifyfs_transfer_request my_reqs[n_files];
char src_file[PATHLEN_MAX];
char dst_file[PATHLEN_MAX];
for (int i = 0; i < (int)n_files; i++) {
    snprintf(src_file, sizeof(src_file), "%s/file.%d", unifyfs_prefix, i);
    snprintf(dst_file, sizeof(src_file), "%s/file.%d", destfs_prefix, i);
    my_reqs[i].src_path = strdup(src_file);
    my_reqs[i].dst_path = strdup(dst_file);
    my_reqs[i].mode = UNIFYFS_TRANSFER_MODE_MOVE;
    my_reqs[i].use_parallel = 1;
}

rc = unifyfs_dispatch_transfer(fshdl, n_files, my_reqs);
if (rc == UNIFYFS_SUCCESS) {
```

(continues on next page)

(continued from previous page)

```
    int waitall = 1;
    rc = unifyfs_wait_transfer(fshdl, n_files, my_reqs, waitall);
    if (rc == UNIFYFS_SUCCESS) {
        for (int i = 0; i < (int)n_files; i++) {
            assert(my_reqs[i].result.error == 0);
        }
    }
}
```

11.4.5 More Examples

Additional examples demonstrating use of the library API can be found in the unit tests (see api-unit-tests).

CHAPTER

TWELVE

UNIFYFS DEPENDENCIES

12.1 Required

- Automake version 1.15 (or later)
- GOTCHA version 1.0.4 (or later)
- Margo version 0.13.1 and its dependencies:
  - Argobots version 1.1 (or later)
  - Mercury version 2.2.0 (or later)
    - libfabric (avoid versions 1.13 and 1.13.1) or bmi
  - JSON-C
- OpenSSL

Important: Margo uses pkg-config to ensure it compiles and links correctly with all of its dependencies' libraries. When building manually, you'll need to set the `PKG_CONFIG_PATH` environment variable to include the paths of the directories containing the `.pc` files for Margo, Mercury, Argobots, and OpenSSL.

12.2 Optional

- spath for normalizing relative paths

CHAPTER

THIRTEEN

UNIFYFS ERROR CODES

Wherever sensible, UnifyFS uses the error codes defined in POSIX errno.h.

UnifyFS specific error codes are defined as follows:

Value	Error	Description
1001	BADCONFIG	Configuration has invalid setting
1002	GOTCHA	Gotcha operation error
1003	KEYVAL	Key-value store operation error
1004	MARGO	Mercury/Argobots operation error
1005	NYI	Not yet implemented
1006	PMI	PMI2/PMIx error
1007	SHMEM	Shared memory region init/access error
1008	THREAD	POSIX thread operation failed
1009	TIMEOUT	Operation timed out

CHAPTER

FOURTEEN

VERIFYIO: DETERMINE UNIFYFS COMPATIBILITY

14.1 Recorder and VerifyIO

VerifyIO can be used to determine an application's compatibility with UnifyFS as well as aid in narrowing down what an application may need to change to become compatible with UnifyFS.

VerifyIO is a tool within the Recorder tracing framework that takes the application traces from Recorder and determines whether I/O synchronization is correct based on the underlying file system semantics (e.g., POSIX, commit) and synchronization semantics (e.g., POSIX, MPI).

Run VerifyIO with commit semantics on the application's traces to determine compatibility with UnifyFS.

14.2 VerifyIO Guide

To use VerifyIO, the Recorder library needs to be installed. See the Recorder README for full instructions on how to build, run, and use Recorder.

14.2.1 Build

Clone the `pilgrim` (default) branch of Recorder:

Listing 1: Clone

```
$ git clone https://github.com/uiuc-hpc/Recorder.git
```

Determine the install locations of the MPI-IO and HDF5 libraries being used by the application and pass those paths to Recorder at configure time.

Listing 2: Configure, Make, and Install

```
$ deps_prefix="${mpi_install};${hdf5_install}"
$ mkdir -p build install

$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=../install -DCMAKE_PREFIX_PATH=$deps_prefix ../Recorder
$ make
$ make install
```

(continues on next page)

(continued from previous page)

```
# Capture Recorder source code and install locations
$ export RECORDER_SRC=/path/to/Recorder/source/code
$ export RECORDER_ROOT=/path/to/Recorder/install
```

Python3 and the `recorder-viz` and `networkx` packages are also required to run the final VerifyIO verification code.

Listing 3: Install Python Packages

```
$ module load python/3.x.x
$
$ pip3 install recorder-viz --user
$ pip3 install networkx --user
```

14.2.2 Run

Before capturing application traces, it is recommended to disable data sieving as VerifyIO will flag this as incompatible under commit semantics.

Listing 4: Disable Data Sieving

```
echo -e "romio_ds_write disable\nromio_ds_read disable" > /path/to/romio_hints
export ROMIO_HINTS=/path/to/romio_hints
export ROMIO_PRINT_HINTS=1   #optional
```

Run the application with Recorder to capture the traces using the appropriate environment variable export option for the available workload manager.

Listing 5: Capture Traces

```
srun -N $nnodes -n $nprocs --export=ALL,LD_PRELOAD=$RECORDER_ROOT/lib/librecorder.so␣
↪example_app_executable
```

Recorder places the trace files in a folder within the current working directory named `hostname-username-appname-pid-starttime`.

If desired (e.g., for debugging), use the recorder2text tool to generate human-readable traces from the captured trace files.

Listing 6: Generate Human-readable Traces

```
$RECORDER_ROOT/bin/recorder2text /path/to/traces &> recorder2text.out
```

This will generate text-format traces in the folder `path/to/traces/_text`.

Next, run the Recorder conflict detector to capture **potential** conflicts. The `--semantics=` option needs to match the semantics of the intended underlying file system. In the case of UnifyFS, use `commit` semantics.

Listing 7: Capture Potential Conflicts

```
$RECORDER_ROOT/bin/conflict_detector /path/to/traces --semantics=commit &> conflict_
↪detector_commit.out
```

The potential conflicts will be recorded to the file `path/to/traces/conflicts.txt`.

Lastly, run VerifyIO with the traces and potential conflicts to determine whether all I/O operations are properly synchronized under the desired standard (e.g., POSIX, MPI).

Listing 8: Run VerifyIO

```
# Evaluate using POSIX standard
python3 $RECORDER_SRC/tools/verifyio/verifyio.py /path/to/traces /path/to/traces/
↪conflicts.txt --semantics=posix &> verifyio_commit_results.posix

# Evaluate using MPI standard
python3 $RECORDER_SRC/tools/verifyio/verifyio.py /path/to/traces /path/to/traces/
↪conflicts.txt --semantics=mpi &> verifyio_commit_results.mpi
```

14.2.3 Interpreting Results

In the event VerifyIO shows an incompatibility, or the results are not clear, don't hesitate to contact the UnifyFS team mailing list for aid in determining a solution.

Conflict Detector Results

When there are no potential conflicts, the conflict detector output simply states as much:

```
[prompt]$ cat conflict_detector_commit.out
Check potential conflicts under Commit Semantics
...
No potential conflict found for file /path/to/example_app_outfile
```

When potential conflicts exist, the conflict detector prints a list of each conflicting pair. For each operation within a pair, the output contains the process rank, sequence ID, offset the conflict occurred at, number of bytes affected by the operation, and whether the operation was a write or a read. This format is printed at the top of the output.

```
[prompt]$ cat conflict_detector_commit.out
Check potential conflicts under Commit Semantics
Format:
Filename, io op1(rank-seqId, offset, bytes, isRead), io op2(rank-seqId, offset, bytes,␣
↪isRead)

/path/to/example_app_outfile, op1(0-244, 0, 800, write), op2(0-255, 0, 96, write)
/path/to/example_app_outfile, op1(0-92, 4288, 2240, write), op2(0-148, 4288, 2216, read)
/path/to/example_app_outfile, op1(1-80, 6528, 2240, write), op2(1-136, 6528, 2216, read)
...
/path/to/example_app_outfile, op1(0-169, 18480, 4888, write), op2(3-245, 18848, 14792,␣
↪read)
/path/to/example_app_outfile, op1(0-169, 18480, 4888, write), op2(3-246, 18848, 14792,␣
↪write)
/path/to/example_app_outfile, op1(0-231, 18480, 16816, write), op2(3-245, 18848, 14792,␣
↪read)
/path/to/example_app_outfile, Read-after-write (RAW): D-2,S-5, Write-after-write (WAW):␣
↪D-1,S-2
```

The final line printed contains a summary of all the potential conflicts. This consists of the total number of read-after-write (RAW) and write-after-write (WAW) potentially conflicting operations performed by different processes (D-#) or the same process (S-#).

VerifyIO Results

VerifyIO takes the traces and potential conflicts and checks if each conflicting pair is properly synchronized. Refer to the VerifyIO README for a description on what determines proper synchronization for a conflicting I/O pair.

Compatible with UnifyFS

In the event that there are no potential conflicts, or each potential conflict pair was performed by the same rank, VerifyIO will report the application as being properly synchronized and therefore compatible with UnifyFS.

```
[prompt]$ cat verifyio_commit_results.posix
Rank: 0, intercepted calls: 79, accessed files: 5
Rank: 1, intercepted calls: 56, accessed files: 2
Building happens-before graph
Nodes: 46, Edges: 84

Properly synchronized under posix semantics


[prompt]$ cat verifyio_commit_results.mpi
Rank: 0, intercepted calls: 79, accessed files: 5
Rank: 1, intercepted calls: 56, accessed files: 2
Building happens-before graph
Nodes: 46, Edges: 56

Properly synchronized under mpi semantics
```

When there are potential conflicts from different ranks but the proper synchronization has occurred, VerifyIO will also report the application as being properly synchronized.

```
[prompt]$ cat verifyio_commit_results.posix
Rank: 0, intercepted calls: 510, accessed files: 8
Rank: 1, intercepted calls: 482, accessed files: 5
Rank: 2, intercepted calls: 481, accessed files: 5
Rank: 3, intercepted calls: 506, accessed files: 5
Building happens-before graph
Nodes: 299, Edges: 685
Conflicting I/O operations: 0-169-write <--> 3-245-read, properly synchronized: True
Conflicting I/O operations: 0-169-write <--> 3-246-write, properly synchronized: True

Properly synchronized under posix semantics
```

Incompatible with UnifyFS

In the event there are potential conflicts from different ranks but the proper synchronization has **not** occurred, VerifyIO will report the application as not being properly synchronized and therefore incompatible*[0] with UnifyFS.

Each operation involved in the conflicting pair is listed in the format `rank-sequenceID-operation` followed by the whether that pair is properly synchronized.

[0] Incompatible here does not mean the application cannot work with UnifyFS at all, just under the default configuration. There are *workarounds* available that could very easily change this result (VerifyIO plans to have options to run under the assumption some workarounds are in place). Should your outcome be an incompatible result, please contact the UnifyFS mailing list for aid in finding a solution.

```
[prompt]$ cat verifyio_commit_results.mpi
Rank: 0, intercepted calls: 510, accessed files: 8
Rank: 1, intercepted calls: 482, accessed files: 5
Rank: 2, intercepted calls: 481, accessed files: 5
Rank: 3, intercepted calls: 506, accessed files: 5
Building happens-before graph
Nodes: 299, Edges: 427
0-169-write --> 3-245-read, properly synchronized: False
0-169-write --> 3-246-write, properly synchronized: False

Not properly synchronized under mpi semantics
```

Debugging a Conflict

The *recorder2text output* can be used to aid in narrowing down where/what is causing a conflicting pair. In the incompatible example above, the first pair is a `write()` from rank 0 with the sequence ID of 169 and a `read()` from rank 3 with the sequence ID of 245.

The sequence IDs correspond to the order in which functions were called by that particular rank. In the recorder2text output, this ID will then correspond to line numbers, but off by +1 (i.e., seqID 169 -> line# 170).

Listing 9: recorder2text output

```
    #rank 0
    ...
167 0.1440291 0.1441011 MPI_File_write_at_all 1 1 ( 0-0 0 %p 1 MPI_TYPE_UNKNOWN [0_0] )
168 0.1440560 0.1440679 fcntl 2 0 ( /path/to/example_app_outfile 7 1 )
169 0.1440700 0.1440750 pread 2 0 ( /path/to/example_app_outfile %p 4888 18480 )
170 0.1440778 0.1440909 pwrite 2 0 ( /path/to/example_app_outfile %p 4888 18480 )
171 0.1440918 0.1440987 fcntl 2 0 ( /path/to/example_app_outfile 6 2 )
    ...

    #rank 3
    ...
244 0.1539204 0.1627174 MPI_File_write_at_all 1 1 ( 0-0 0 %p 1 MPI_TYPE_UNKNOWN [0_0] )
245 0.1539554 0.1549513 fcntl 2 0 ( /path/to/example_app_outfile 7 1 )
246 0.1549534 0.1609544 pread 2 0 ( /path/to/example_app_outfile %p 14792 18848 )
247 0.1609572 0.1627053 pwrite 2 0 (/path/to/example_app_outfile %p 14792 18848 )
248 0.1627081 0.1627152 fcntl 2 0 ( /path/to/example_app_outfile 6 2 )
    ...
```

Note that in this example the `pread()`/`pwrite()` calls from rank 3 operate on overlapping bytes from the `pwrite()` call from rank 0. For this example, data sieving was left enabled which results in "fcntl-pread-pwrite-fcntl" I/O sequences. Refer to *Limitations and Workarounds* for more on the file locking limitations of UnifyFS.

The format of the recorder2text output is: `<start-time> <end-time> <func-name> <call-level> <func-type> (func-parameters)`

Note: The `<call-level>` value indicates whether the function was called directly by the application or by an I/O library. The `<func-type>` value shows the Recorder-tracked function type.

Value	Call Level	Value	Function Type
0	Called by application directly	0	RECORDER_POSIX
1	• Called by HDF5 • Called by MPI (no HDF5)	1	RECORDER_MPIIO
2	Called by MPI, which was called by HDF5	2	RECORDER_MPI
		3	RECORDER_HDF5
		4	RECORDER_FTRACE

CHAPTER

FIFTEEN

CONTRIBUTING GUIDE

First of all, thank you for taking the time to contribute!

By using the following guidelines, you can help us make UnifyFS even better.

15.1 Getting Started

15.1.1 Get UnifyFS

You can build and run UnifyFS by following *these instructions*.

15.1.2 Getting Help

To contact the UnifyFS team, send an email to the mailing list.

15.2 Reporting Bugs

Please contact us via the mailing list if you are not certain that you are experiencing a bug.

You can open a new issue and search existing issues using the issue tracker.

If you run into an issue, please search the issue tracker first to ensure the issue hasn't been reported before. Open a new issue only if you haven't found anything similar to your issue.

Important: When opening a new issue, please include the following information at the top of the issue:

- What operating system (with version) you are using
- The UnifyFS version you are using
- Describe the issue you are experiencing
- Describe how to reproduce the issue
- Include any warnings or errors
- Apply any appropriate labels, if necessary

When a new issue is opened, it is not uncommon for developers to request additional information.

In general, the more detail you share about a problem the more quickly a developer can resolve it. For example, providing a simple test case is extremely helpful. Be prepared to work with the developers investigating your issue. Your assistance is crucial in providing a quick solution.

15.3 Suggesting Enhancements

Open a new issue in the issue tracker and describe your proposed feature. Why is this feature needed? What problem does it solve? Be sure to apply the *enhancement* label to your issue.

15.4 Pull Requests

- All pull requests must be based on the current *dev* branch and apply without conflicts.
- Please attempt to limit pull requests to a single commit which resolves one specific issue.
- Make sure your commit messages are in the correct format. See the *Commit Message Format* section for more information.
- When updating a pull request, squash multiple commits by performing a rebase (squash).
- For large pull requests, consider structuring your changes as a stack of logically independent patches which build on each other. This makes large changes easier to review and approve which speeds up the merging process.
- Try to keep pull requests simple. Simple code with comments is much easier to review and approve.
- Test cases should be provided when appropriate.
- If your pull request improves performance, please include some benchmark results.
- The pull request must pass all regression tests before being accepted.
- All proposed changes must be approved by a UnifyFS project member.

15.5 Testing

All help is appreciated! If you're in a position to run the latest code, consider helping us by reporting any functional problems, performance regressions, or other suspected issues. By running the latest code on a wide range of realistic workloads, configurations, and architectures we're better able to quickly identify and resolve issues.

15.6 Documentation

As UnifyFS is continually improved and updated, it is easy for documentation to become out-of-date. Any contributions to the documentation, no matter how small, is always greatly appreciated. If you are not in a position to update the documentation yourself, please notify us via the mailing list of anything you notice that is missing or needs to be changed.

CHAPTER

SIXTEEN

DEVELOPER DOCUMENTATION

Here is our current documentation of how the internals of UnifyFS function for several basic operations.

UnifyFS Developer's Documentation

Download PDF.

CHAPTER

SEVENTEEN

STYLE GUIDES

17.1 Coding Conventions

UnifyFS follows the Linux kernel coding style except that code is indented using four spaces per level instead of tabs. Please run `make checkstyle` to check your patch for style problems before submitting it for review.

17.1.1 Styling Code

The astyle tool can be used to apply much of the required code styling used in the project.

Listing 1: To apply style to the source file foo.c:

```
astyle --options=scripts/unifyfs.astyle foo.c
```

The unifyfs.astyle file specifies the options used for this project. For a full list of available astyle options, see http://astyle.sourceforge.net/astyle.html.

17.1.2 Verifying Style Checks

To check that uncommitted changes meet the coding style, use the following command:

```
git diff | ./scripts/checkpatch.sh
```

Tip: This command will only check specific changes and additions to files that are already tracked by git. Run the command `git add -N [<untracked_file>...]` first in order to style check new files as well.

17.2 Commit Message Format

Commit messages for new changes must meet the following guidelines:

- In 50 characters or less, provide a summary of the change as the first line in the commit message.
- A body which provides a description of the change. If necessary, please summarize important information such as why the proposed approach was chosen or a brief description of the bug you are resolving. Each line of the body must be 72 characters or less.

An example commit message for new changes is provided below.

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug."  This convention matches up with commit messages generated
by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay

- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

CHAPTER

EIGHTEEN

TESTING GUIDE

We can never have enough testing. Any additional tests you can write are always greatly appreciated.

18.1 Unit Tests

18.1.1 Implementing Tests

The UnifyFS Test Suite uses the Test Anything Protocol (TAP) and the Automake test harness. This test suite has two types of TAP tests (shell scripts and C) to allow for testing multiple aspects of UnifyFS.

Shell Script Tests

Test cases in shell scripts are implemented with sharness, which is included in the UnifyFS source distribution. See the file sharness.sh for all available test interfaces. UnifyFS-specific sharness code is implemented in scripts in the directory sharness.d. Scripts in sharness.d are primarily used to set environment variables and define convenience functions. All scripts in sharness.d are automatically included when your script sources sharness.sh.

The most common way to implement a test case with sharness is to use the `test_expect_success()` function. Your script must first set a test description and source the sharness library. After all tests are defined, your script should call `test_done()` to print a summary of the test run.

Test cases that demonstrate known breakage should use the sharness function `test_expect_failure()` to alert developers about the problem without causing the overall test suite to fail. Failing test cases should be tracked with github issues.

Here is an example of a sharness test:

```
#!/bin/sh

test_description="My awesome test cases"

. $(dirname $0)/sharness.sh

test_expect_success "Verify some critical invariant" '
    test 1 -eq 1
'

test_expect_failure "Prove this someday" '
    test "P" == "NP"
'
```

(continues on next page)

(continued from previous page)

```

# Various tests available to use inside test_expect_success/failure
test_expect_success "Show various available tests" '
    test_path_is_dir /somedir
    test_must_fail test_dir_is_empty /somedir
    test_path_is_file /somedir/somefile
'

# Use test_set_prereq/test_have_prereq to conditionally skip tests
[[ -n $(which h5cc 2>/dev/null) ]] && test_set_prereq HAVE_HDF5
if test_have_prereq HAVE_HDF5; then
    # run HDF5 tests
fi

# Can also check for prereq in individual test
test_expect_success HAVE_HDF5 "Run HDF5 test" '
    # Run HDF5 test
'

test_done
```

C Program Tests

C programs use the libtap library to implement test cases. All available testing functions are viewable in the libtap README. Convenience functions common to test cases written in C are implemented in the library lib/testutil.c. If your C program needs to use environment variables set by sharness, it can be wrapped in a shell script that first sources sharness.d/00-test-env.sh and sharness.d/01-unifyfs-settings.sh. Your wrapper shouldn't normally source sharness.sh itself because the TAP output from sharness might conflict with that from libtap.

The most common way to implement a test with libtap is to use the `ok()` function. TODO test cases that demonstrate known breakage are surrounded by the libtap library calls `todo()` and `end_todo()`.

Here are some examples of libtap tests:

```
#include "t/lib/tap.h"
#include "t/lib/testutil.h"
#include <string.h>

int main(int argc, char *argv[])
{
    int result;

    result = (1 == 1);
    ok(result, "1 equals 1: %d", result);

    /* Or put a function call directly in test */
    ok(somefunc() == 42, "somefunc() returns 42");
    ok(somefunc() == -1, "somefunc() should fail");

    /* Use pass/fail for more complex code paths */
    int x = somefunc();
    if (x > 0) {
```

(continues on next page)

(continued from previous page)

```
        pass("somefunc() returned a valid value");
    } else {
        fail("somefunc() returned an invalid value");
    }

    /* Use is/isnt for string comparisions */
    char buf[64] = {0};
    ok(fread(buf, 12, 1, fd) == 1, "read 12 bytes into buf);
    is(buf, "hello world", "buf is \"hello world\"");

    /* Use cmp_mem to test first n bytes of memory */
    char* a = "foo";
    char* b = "bar";
    cmp_mem(a, b, 3);

    /* Use like/unlike to string match to a POSIX regex */
    like("stranger", "^s.(r).*\\1$", "matches the regex");

    /* Use dies_ok/lives_ok to test whether code causes an exit */
    dies_ok({int x = 0/0;}, "divide by zero crashes");

    /* Use todo for failing tests to be notified when they start passing */
    todo("Prove this someday");
    result = strcmp("P", "NP");
    ok(result == 0, "P equals NP: %d", result);
    end_todo;

    /* Use skip/end_skip when a feature isn't implemented yet, or to
    conditionally skip when a resource isn't available */
    skip(TRUE, 2, "Reason for skipping tests");
    ok(1);
    ok(2);
    end_skip;

    #ifdef HAVE_SOME_FEATURE
        ok(somefunc());
        ok(someotherfunc());
    #else
        skip(TRUE, 2, "Don't have SOME_FEATURE");
        end_skip;
    #endif

    done_testing();
}
```

Tip: Including the file and line number, as well as any useful variable values, in each test output can be very helpful when a test fails or needs to be debugged.

```
ok(somefunc() == 42, "%s:%d somefunc() returns 42", __FILE__,
__LINE__);
```

Also note that `errno` is only set when an error occurs and is never set back to `0` implicitly by the system. When testing

for a failure and using `errno` as part of the test, setting `errno = 0` before the test will ensure a previous test error will not affect the current test. In the following example, we also assign `errno` to another variable `err` for use in constructing the test message. This is needed because the `ok()` macro may use system calls that set `errno`.

```
int err, rc;
errno = 0;
rc = systemcall();
err = errno;
ok(rc == -1 && err == ENOTTY,
   "%s:%d systemcall() should fail (errno=%d): %s",
   __FILE__, __LINE__, err, strerror(err));
```

18.1.2 Adding Tests

The UnifyFS Test Suite uses the Test Anything Protocol (TAP) and the Automake test harness. By convention, test scripts and programs that output TAP are named with a ".t" extension.

To add a new test case to the test harness, follow the existing examples in t/Makefile.am. In short, add your test program to the list of tests in the `TESTS` variable. If it is a shell script, also add it to `check_SCRIPTS` so that it gets included in the source distribution tarball.

Test Suites

If multiple tests fit within the same category (i.e., tests for creat and mkdir both fall under tests for sysio) then create a test suite to run those tests. This makes it so less duplication of files and code is needed in order to create additional tests.

To create a new test suite, look at how it is currently done for the sysio_suite in t/Makefile.am and t/sys/sysio_suite.c:

If you're testing C code, you'll need to use environment variables set by sharness.

- Create a shell script, *<####-suite-name>.t* (the #### indicates the order in which they should be run by the tap-driver), that wraps your suite and sources sharness.d/00-test-env.sh and sharness.d/01-unifyfs-settings.sh
- Add this file to t/Makefile.am in the `TESTS` and `check_SCRIPTS` variables and add the name of the file (but with a .t extension) this script runs to the `libexec_PROGRAMS` variable

You can then create the test suite file and any tests to be run in this suite.

- Create a <test_suite_name>.c file (i.e., *sysio_suite.c*) that will contain the main function and mpi job that drives your suite
  - Mount unifyfs from this file
  - Call testing functions that contain the test cases (created in other files) in the order desired for testing, passing the mount point to those functions
- Create a <test_suite_name>.h file that declares the names of all the test functions to be run by this suite and `include` this in the <test_suite_name>.c file
- Create <test_name>.c files (i.e., *open.c*) that contains the testing function (i.e., `open_test(char* unifyfs_root)`) that houses the variables and libtap tests needed to test that individual function
  - Add the function name to the <test_suite_name>.h file

  - Call the function from the <test_suite_name>.c file

The source files and flags for the test suite are then added to the bottom of t/Makefile.am.

- Add the <test_suite_name>.c and <test_suite_name>.h files to the `<test_suite>_SOURCES` variable
- Add additional <test_name>.c files to the `<test_suite>_SOURCES` variable as they are created
- Add the associated flags for the test suite (if the suite is for testing wrappers, add a suite and flags for both a gotcha and a static build)

Test Cases

For testing C code, test cases are written using the libtap library. See the *C Program Tests* section above on how to write these tests.

To add new test cases to any existing suite of tests:

1. Simply add the desired tests (order matters) to the appropriate <test_name>.c file

If the test cases needing to be written don't already have a file they belong in (i.e., testing a wrapper that doesn't have any tests yet):

1. Creata a <function_name>.c file with a function called <function_name>_test(char* unifyfs_root) that contains the desired libtap test cases
2. Add the <function_name>_test to the corresponding <test_suite_name>.h file
3. Add the <function_name>.c file to the bottom of t/Makefile.am under the appropriate `<test_suite>_SOURCES` variable(s)
4. The <function_name>_test function can now be called from the <test_suite_name>.c file

---

18.1.3 Running the Tests

To manually run the UnifyFS unit test suite, simply run `make check` in a single-node allocation from inside the t/ directory of wherever you built UnifyFS. E.g., if you built in a separate build/ directory, then do:

```
$ cd build/t
$ make check
```

If on a system where jobs are launched on a separate compute node, then use your systems local MPI job launch command to run the unit tests:

```
$ cd build/t
$ srun -N1 -n1 make check
```

If changes are made to existing files in the test suite, the tests can be run again by simply doing `make clean` followed by another `make check`.

Individual tests may be run by hand. The test *0001-setup.t* should normally be run first to start the UnifyFS daemon. E.g., to run just the *0100-sysio-gotcha.t* tests, do:

```
$ make check TESTS='0001-setup.t 0100-sysio-gotcha.t 9010-stop-unifyfsd.t 9999-cleanup.t'
```

Note: Running Unit Tests from Spack Install

If using Spack to install UnifyFS there are two ways to manually run the units tests:

1. Upon installation with Spack

   ```
   spack install -v --test=root unifyfs
   ```

2. Manually from Spack's build directory

   Open the Spack config file:

   ```
   spack config edit config
   ```

   Provide Spack a staging path that is visible from a job allocation:

   ```
   config:
     build_stage:
     - /visible/path/allocated/node
     # or build directly inside Spack's install directory
     - $spack/var/spack/stage
   ```

   Include the `--keep-stage` option when installing:

   ```
   spack install --keep-stage unifyfs
   ```

   ```
   spack cd unifyfs
   ```

   ```
   cd spack-build/t
   ```

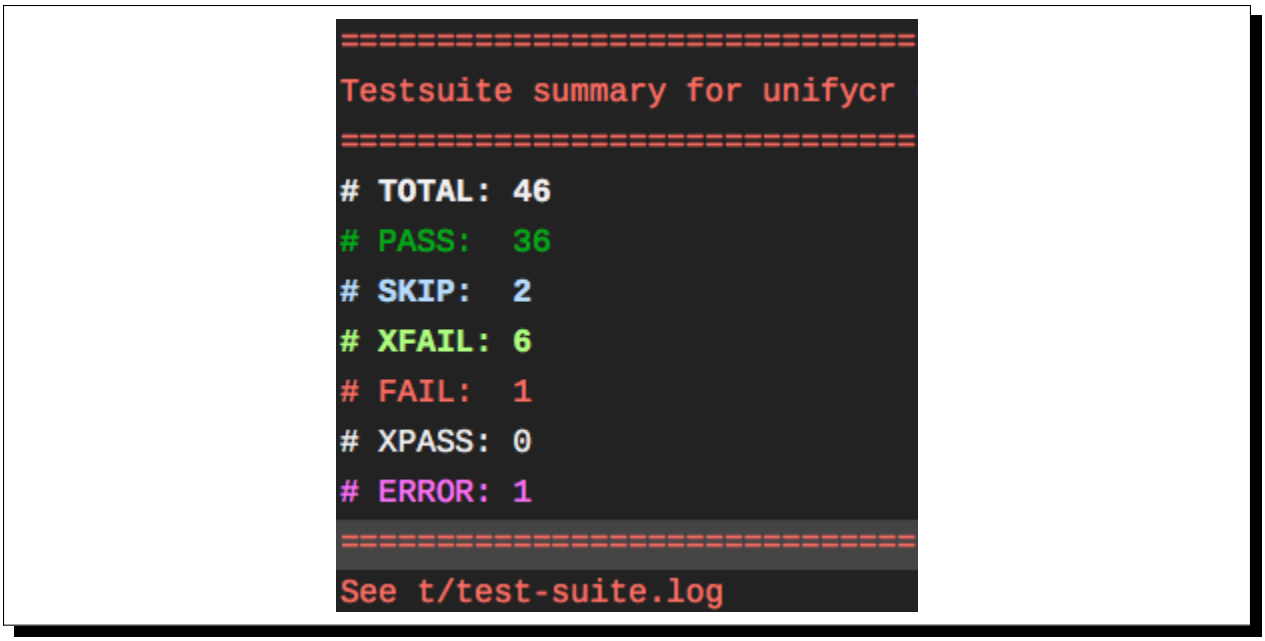
   Run the tests from the package's build environment:

   ```
   spack build-env unifyfs make check
   ```

The tests in https://github.com/LLNL/UnifyFS/tree/dev/t are run automatically using GitHub Actions along with the *style checks* when a pull request is created or updated. All pull requests must pass these tests before they will be accepted.

Interpreting the Results

TAP Output



After a test runs, its result is printed out consisting of its status followed by its description and potentially a TODO/SKIP message. Once all the tests have completed (either from being run manually or by GitHub Actions), the overall results are printed out, as shown in the image on the right.

There are six possibilities for the status of each test: PASS, FAIL, XFAIL, XPASS, SKIP, and ERROR.

PASS
The test had the desired result.

FAIL
The test did not have the desired result. These must be fixed before any code changes can be accepted.

If a FAIL occurred after code had been added/changed then most likely a bug was introduced that caused the test to fail. Some tests may fail as a result of earlier tests failing. Fix bugs that are causing earlier tests to fail first as, once they start passing, subsequent tests are likely to start passing again as well.

XFAIL
The test was expected to fail, and it did fail.

An XFAIL is created by surrounding a test with `todo()` and `end_todo`. These are tests that have identified a bug that was already in the code, but the cause of the bug hasn't been found/resolved yet. An optional message can be passed to the `todo("message")` call which will be printed after the test has run. Use this to explain how the test should behave or any thoughts on why it might be failing. An XFAIL is not meant to be used to make a failing test start "passing" if a bug was introduced by code changes.

XPASS
A test passed that was expected to fail. These must be fixed before any code changes can be accepted.

The relationship of an XPASS to an XFAIL is the same as that of a FAIL to a PASS. An XPASS will typically occur when a bug causing an XFAIL has been fixed and the test has started passing. If this is the case, remove the surrounding `todo()` and `end_todo` from the failing test.

SKIP
The test was skipped.

Tests are skipped because what they are testing hasn't been implemented yet, or they apply to a feature/variant that wasn't included in the build (i.e., HDF5). A SKIP is created by surrounding the test(s) with `skip(test,`

`n, message)` and `end_skip` where the `test` is what determines if these tests should be skipped and `n` is the number of subsequent tests to skip. Remove these if it is no longer desired for those tests to be skipped.

ERROR
A test or test suite exited with a non-zero status.

When a test fails, the containing test suite will exit with a non-zero status, causing an ERROR. Fixing any test failures should resolve the ERROR.

Running the Examples

To run any of these examples manually, refer to the *Example Programs* documentation.

The UnifyFS examples are also being used as integration tests with continuous integration tools such as Bamboo or GitLab CI.

18.2 Integration Tests

The UnifyFS examples are being used as integration tests with continuation integration tools such as Bamboo or GitLab CI.

To run any of these examples manually, refer to the *Example Programs* documentation.

18.2.1 Configuration Variables

Along with the already provided *UnifyFS Configuration* options/environment variables, there are environment variables used by the integration testing suite that can also be set in order to change the default behavior.

Key Variables

These environment variables can be set prior to sourcing the *t/ci/001-setup.sh* script and will affect how the overall integration suite operates.

`UNIFYFS_INSTALL`

USAGE: `UNIFYFS_INSTALL=/path/to/dir/containing/UnifyFS/bin/directory`

The full path to the directory containing the *bin/* and *libexec/* directories for your UnifyFS installation. Set this envar to prevent the integration tests from searching for a UnifyFS installation automatically. Where the automatic search starts can be altered by setting the `$BASE_SEARCH_DIR` variable.

UNIFYFS_CI_NPROCS

USAGE: `UNIFYFS_CI_NPROCS=<number-of-process-per-node>`

The number of processes to use per node inside a job allocation. This defaults to 1 process per node. This can be adjusted if more processes are desired on multiple nodes or multiple processes are desired on a single node.

UNIFYFS_CI_TEMP_DIR

USAGE: `UNIFYFS_CI_TEMP_DIR=/path/for/temporary/files/created/by/UnifyFS`

Can be used as a shortcut to set `UNIFYFS_RUNSTATE_DIR` and `UNIFYFS_META_DB_PATH` to the same path. This envar defaults to `UNIFYFS_CI_TEMP_DIR=${TMPDIR}/unifyfs.${USER}.${JOB_ID}`.

UNIFYFS_CI_LOG_CLEANUP

USAGE: `UNIFYFS_CI_LOG_CLEANUP=yes|YES|no|NO`

In the event `$UNIFYFS_LOG_DIR` has **not** been set, the logs will be put in `$SHARNESS_TRASH_DIRECTORY`, as set up by sharness.sh, and cleaned up automatically after the tests have run. The logs will be in a *<system-name>_<jobid>/* subdirectory. Should any tests fail, sharness does not clean up the trash directory for debugging purposes. Setting `UNIFYFS_CI_LOG_CLEANUP=no|NO` will move the *<system-name>_<jobid>/* logs directory to `$UNIFYFS_CI_DIR` (the directory containing the integration testing scripts) to allow them to persist even when all tests pass. This envar defauls to `yes`.

Note: Setting `$UNIFYFS_LOG_DIR` will put all created logs in the designated path and will not clean them up.

UNIFYFS_CI_HOST_CLEANUP

USAGE: `UNIFYFS_CI_HOST_CLEANUP=yes|YES|no|NO`

After all tests have run, the nodes on which the tests were ran will automatically be cleaned up. This cleanup includes ensuring `unifyfsd` has stopped and deleting any files created by UnifyFS or its dependencies. Set `UNIFYFS_CI_HOST_CLEANUP=no|NO` to skip cleaning up. This envar defaults to `yes`.

Note: PDSH is required for cleanup and cleaning up is simply skipped if not found.

UNIFYFS_CI_CLEANUP

USAGE: `UNIFYFS_CI_CLEANUP=yes|YES|no|NO`

Setting this to `no|NO` sets both `$CI_LOG_CLEANUP` and `$UNIFYFS_CI_HOST_CLEANUP` to `no|NO`.

UNIFYFS_CI_TEST_POSIX

USAGE: `UNIFYFS_CI_TEST_POSIX=yes|YES|no|NO`

Determines whether any `<example-name>-posix` tests should be run since they require a real mountpoint to exist.

This envar defaults to `no`. Setting this to `yes` will run the posix version of tests along with the regular tests. When `$UNIFYFS_MOUNTPOINT` is set to a existing directory, this option is set to `no`. This is to allow running the tests a first time with a fake mountpoint while the posix tests use an existing mountpoint. Then the regular tests can be run again using an existing mountpoint and the posix tests won't be run twice.

An example of testing a posix example can be see *below*.

Note: The posix mountpoint envar, `UNIFYFS_CI_POSIX_MP`, is set to be located inside `$SHARNESS_TRASH_DIRECTORY` automatically and cleaned up afterwards. However, this envar can be set before running the integration tests as well. If setting this, ensure that it is a shared file system that all allocated nodes can see.

Additional Variables

After sourcing the *t/ci/001-setup.sh* script there will be additional variables available that may be useful when writing/adding additional tests.

Directory Structure

File structure here is assuming UnifyFS was cloned to `$HOME`.

UNIFYFS_CI_DIR
: Directory containing the CI testing scripts. *$HOME/UnifyFS/t/ci/*

SHARNESS_DIR
: Directory containing the base sharness scripts. *$HOME/UnifyFS/t/*

UNIFYFS_SOURCE_DIR
: Directory containing the UnifyFS source code. *$HOME/UnifyFS/*

BASE_SEARCH_DIR
: Parent directory containing the UnifyFS source code. Starting place to auto search for UnifyFS install when `$UNIFYFS_INSTALL` isn't provided. *$HOME/*

Executable Locations

UNIFYFS_BIN
: Directory containing `unifyfs` and `unifyfsd`. *$UNIFYFS_INSTALL/bin*

UNIFYFS_EXAMPLES
: Directory containing the compiled examples. *$UNIFYFS_INSTALL/libexec*

Resource Managers

JOB_RUN_COMMAND
The base MPI job launch command established according to the detected resource manager, number of allocated nodes, and `$UNIFYFS_CI_NPROCS`.

The LSF variables below will also affect the default version of this command when using that resource manager.

JOB_RUN_ONCE_PER_NODE
MPI job launch command to only run a single process on each allocated node established according to the detected resource manager.

JOB_ID
The ID assigned to the current CI job as established by the detected resource manager.

LSF

Additional variables used by the LSF resource manager to determine how jobs are launched with `$JOB_RUN_COMMAND`. These can also be set prior to sourcing the *t/ci/001-setup.sh* script and will affect how the integration tests run.

UNIFYFS_CI_NCORES
Number of cores-per-resource-set to use. Defaults to 20.

UNIFYFS_CI_NRS_PER_NODE
Number of resource-sets-per-node to use. Defaults to 1.

UNIFYFS_CI_NRES_SETS
Total number of resource sets to use. Defaults to (number_of_nodes) * (`$UNIFYFS_CI_NRS_PER_NODE`).

Misc

KB
2^{10}

MB
2^{20}

GB
2^{30}

18.2.2 Running the Tests

Attention: UnifyFS's integration test suite requires MPI and currently only supports `srun` and `jsrun` MPI launch commands.

UnifyFS's integration tests are primarly set up to run distinct suites of tests, however they can also all be run at once or manually for more fine-grained control.

The testing scripts in t/ci depend on sharness, which is set up in the containing *t/* directory. These tests will not function properly if moved or if the sharness files cannot be found.

Before running any tests, ensure either compute nodes have been interactively allocated or run via a batch job submission.

Make sure all *dependencies* are installed and loaded.

The *t/ci/RUN_CI_TESTS.sh* script is designed to simplify running various suites of tests.

RUN_CI_TESTS.sh Script

```
Usage: ./RUN_CI_TESTS.sh [-h] -s {all|[writeread,[write|read],pc,stage]} -t {all|[posix,
↪mpiio]}

Any previously set UnifyFS environment variables will take precedence.

Options:
  -h, --help
         Print this help message

  -s, --suite {all|[writeread,[write|read],pc,stage]}
         Select the test suite(s) to be run
         Takes a comma-separated list of available suites

  -t, --type {all|[posix,mpiio]}
         Select the type(s) of each suite to be run
         Takes a comma-separated list of available types
         Required with --suite unless stage is the only suite selected
```

Note: Running Integration Tests from Spack Build

Running the integration tests from a Spack installation of UnifyFS requires telling Spack to use a different location for staging the build in order to have the source files available from inside a job allocation.

Open the Spack config file:

```
spack config edit config
```

Provide a staging path that is visible to all nodes from a job allocations:

```
config:
  build_stage:
  - /visible/path/from/all/allocated/nodes
  # or build directly inside Spack's install directory
  - $spack/var/spack/stage
```

Include the `--keep-stage` option when installing:

```
spack install --keep-stage unifyfs
```

Allocate compute nodes and spawn a new shell containing the package's build environment:

```
spack build-env unifyfs bash
```

Run the integration tests:

```
spack load unifyfs
```

```
spack cd unifyfs
```

```
cd t/ci
```

```
# Run tests using any of the following formats
```

Individual Suites

To run individual test suites, indicate the desired suite(s) and type(s) when running *RUN_CI_TESTS.sh*. E.g.:

```
$ ./RUN_CI_TESTS.sh -s writeread -t mpiio
```

or

```
$ prove -v RUN_CI_TESTS.sh :: -s writeread -t mpiio
```

The `-s|--suite` and `-t|--type` options flag which set(s) of tests to run. Each suite (aside from `stage`) requires a type to be selected as well. Note that if `all` is selected, the other arguments are redundant. If the `read` suite is selected, then the `write` argument is redundant.

Available suites: all|[writeread,[write,read],pc,stage]
: all: run all suites writeread: run writeread tests write: run write tests only (redundant if read also set) read: run write then read tests (all-hosts producer-consumer tests) pc: run producer-consumer tests (disjoint sets of hosts) stage: run stage tests (type not required)

Available types: all|[posix,mpiio]
: all: run all types posix: run posix versions of above suites mpiio: run mpiio versions of above suites

All Tests

> **Warning:** If running all or most tests within a single allocation, a large amount of time and storage space will be required. Even if enough of both are available, it is still possible the run may hit other limitations (e.g., `client_max_files`, `client_max_active_requests`, `server_max_app_clients`). To avoid this, run individual suites from separate job allocations.

To run all of the tests, run *RUN_CI_TESTS.sh* with the all suites and types options.

```
$ ./RUN_CI_TESTS.sh -s all -t all
```

or

```
$ prove -v RUN_CI_TESTS.sh :: -s all -t all
```

Subsets of Individual Suites

Subsets of individual test suites can be run manually. This can be useful when wanting more fine-grained control or for testing a specific configuration. To run manually, the testing functions and variables need to be set up first and then the UnifyFS servers need to be started.

First source the *t/ci/001-setup.sh* script whereafter sharness will change directories to the `$SHARNESS_TRASH_DIRECTORY`. To account for this, prefix each subsequent script with `$UNIFYFS_CI_DIR/` when sourcing. Start the servers next by sourcing *002-start-server.sh* followed by each desired test script. When finished, source *990-stop-server.sh* last to stop the servers, report the results, and clean up.

```
$ . ./001-setup.sh
$ . $UNIFYFS_CI_DIR/002-start-server.sh
$ . $UNIFYFS_CI_DIR/100-writeread-tests.sh --laminate --shuffle --mpiio
$ . $UNIFYFS_CI_DIR/990-stop-server.sh
```

The various CI test suites can be run multiple times with different behaviors. These behaviors are continually being extended. The *-h|–help* option for each script can show what alternate behaviors are currently implemented along with additional information for that particular suite.

```
[prompt]$ ./100-writeread-tests.sh --help
Usage: 100-writeread-tests.sh [options...]

  options:
    -h, --help        print help message
    -l, --laminate    laminate between writing and reading
    -M, --mpiio       use MPI-IO instead of POSIX I/O
    -x, --shuffle     read different data than written
```

18.2.3 Adding New Tests

In order to add additional tests for different workflows, create a script after the fashion of t/ci/100-writeread-tests.sh where the prefixed number indicates the desired order for running the tests. Then source that script in t/ci/RUN_CI_TESTS.sh in the desired order. The different test suite scripts themselves can also be edited to add/change the number, types, and various behaviors each suite will execute.

Just like the helpers functions found in sharness.d, there are continuous integration helper functions (see *below* for more details) available in t/ci/ci-functions.sh. These exist to help make adding new tests as simple as possible.

One particularly useful function is `unify_run_test()`. Currently, this function is set up to work for the *write*, *read*, *writeread*, and *checkpoint-restart* examples. This function sets up the MPI job run command and default options as well as any default arguments wanted by all examples. See *below* for details.

Testing Helper Functions

There are helper functions available in t/ci/ci-functions.sh that can make running and testing the examples much easier. These may get adjusted over time to accommodate other examples, or additional functions may need to be written. Some of the main helper functions that might be useful for running examples are:

`unify_run_test()`

USAGE: `unify_run_test app_name "app_args" [output_variable_name]`

Given a example application name and application args, this function runs the example with the appropriate MPI runner and args. This function is meant to make running the cr, write, read, and writeread examples as easy as possible.

The `build_test_command()` function is called by this function which automatically sets any options that are always wanted (-vkfo as well as -U and the appropriate -m if posix test or not). The stderr output file is also created (based on the filename that is autogenerated) and the appropriate option is set for the MPI job run command.

Args that can be passed in are ([-cblnpx][-A|-L|-M|-N|-P|-S|-V]). All other args (see *Running the Examples*) are set automatically, including the outfile and filename (which are generated based on the input `$app_name` and `$app_args`).

The third parameter is an optional "pass-by-reference" parameter that can contain the variable name for the resulting output to be stored in, allowing this function to be used in one of two ways:

Listing 1: Using command substitution

```
app_output=$(unify_run_test $app_name "$app_args")
```

or

Listing 2: Using a "pass-by-reference" variable

```
unifyfs_run_test $app_name "$app_args" app_output
```

This function returns the return code of the executed example as well as the output produced by running the example.

Note: If `unify_run_test()` is simply called with only two arguments and without using command substitution, the resulting output will be sent to the standard output.

The results can then be tested with sharness:

```
basetest=writeread
runmode=static

app_name=${basetest}-${runmode}
app_args="-p n1 -n32 -c $((16 * $KB)) -b $MB

unify_run_test $app_name "$app_args" app_output
rc=$?
line_count=$(echo "$app_output" | wc -l)

test_expect_success "$app_name $app_args: (line_count=$line_count, rc=$rc)" '
    test $rc = 0 &&
    test $line_count = 8
'
```

get_filename()

USAGE: `get_filename app_name app_args [app_suffix]`

Builds and returns the filename with the provided suffix based on the input app_name and app_args.

The filename in `$UNIFYFS_MOUNTPOINT` will be given a `.app` suffix.

This allows tests to get what the filename will be in advance if called from a test suite. This can be used for posix tests to ensure the file showed up in the mount point, as well as for read, cp, stat tests that potentially need the filename from a previous test prior to running.

Error logs and outfiles are also created with this filename, with a `.err` or `.out` suffix respectively, and placed in the logs directory.

Returns a string with the spaces removed and hyphens replaced by underscores.

```
get_filename write-static "-p n1 -n 32 -c 1024 -b 1048576" ".app"
write-static_pn1_n32_c1KB_b1MB.app
```

Some uses cases may be:

- posix tests where the file existence is checked for after a test was run

- read, cp, or stat tests where an already existing filename from a prior test might be needed

For example:

```
basetest=writeread
runmode=posix

app_name=${basetest}-${runmode}
app_args="-p nn -n32 -c $((16 * $KB)) -b $MB

unify_run_test $app_name "$app_args" app_output
rc=$?
line_count=$(echo "$app_output" | wc -l)
filename=$(get_filename $app_name "$app_args" ".app")

test_expect_success POSIX "$app_name $app_args: (line_count=$line_count, rc=$rc)" '
    test $rc = 0 &&
    test $line_count = 8 &&
    test_path_has_file_per_process $UNIFYFS_CI_POSIX_MP $filename
'
```

Additional Functions

There are other convenience functions used bythat my be helpful in writing/adding tests are also found in t/ci/ci-functions.sh:

find_executable()
: USAGE: `find_executable abs_path *file_name|*path/file_name [prune_path]`

  Locate the desired executable file when provided an absolute path of where to start searching, the name of the file with an optional preceding path, and an optional prune_path, or path to omit from the search.

  Returns the path of the first executable found with the given name and optional prefix.

elapsed_time()
: USAGE: `elapsed_time start_time_in_seconds end_time_in_seconds`

  Calculates the elapsed time between two given times.

  Returns the elapsed time formatted as HH:MM:SS.

format_bytes()
: USAGE: `format_bytes int`

  Returns the input bytes formatted as KB, MB, or GB (1024 becomes 1KB).

Sharness Helper Functions

There are also additional sharness functions for testing the examples available when t/ci/ci-functions.sh is sourced. These are to be used with sharness for testing the results of running the examples with or without using the *Example Helper Functions*.

process_is_running()

USAGE: `process_is_running process_name seconds_before_giving_up`

Checks if a process with the given name is running on every host, retrying up to a given number of seconds before giving up. This function overrides the `process_is_running()` function used by the UnifyFS unit tests. The primary difference being that this function checks for the process on every host.

Expects two arguments:

- $1 - Name of a process to check for
- $2 - Number of seconds to wait before giving up

```
test_expect_success "unifyfsd is running" '
    process_is_running unifyfsd 5
'
```

process_is_not_running()

USAGE: `process_is_not_running process_name seconds_before_giving_up`

Checks if a process with the given name is not running on every host, retrying up to a given number of seconds before giving up. This function overrides the `process_is_not_running()` function used by the UnifyFS unit tests. The primary difference being that this function checks that the process is not running on every host.

Expects two arguments:

- $1 - Name of a process to check for
- $2 - Number of seconds to wait before giving up

```
test_expect_success "unifyfsd is not running" '
    process_is_not_running unifyfsd 5
'
```

test_path_is_dir()

USAGE: `test_path_is_dir dir_name [optional]`

Checks that a directory with the given name exists and is accessible from each host. Does NOT need to be a shared directory. This function overrides the `test_path_is_dir()` function in sharness.sh, the primary difference being that this function checks for the dir on every host in the allocation.

Takes once argument with an optional second:

- $1 - Path of the directory to check for
- $2 - Can be given to provide a more precise diagnosis

```
test_expect_success "$dir_name is an existing directory" '
    test_path_is_dir $dir_name
'
```

test_path_is_shared_dir()

USAGE: `test_path_is_shared_dir dir_name [optional]`

Check if same directory (actual directory, not just name) exists and is accessible from each host.

Takes once argument with an optional second:

- $1 - Path of the directory to check for
- $2 - Can be given to provide a more precise diagnosis

```
test_expect_success "$dir_name is a shared directory" '
    test_path_is_shared_dir $dir_name
'
```

test_path_has_file_per_process()

USAGE: `test_path_has_file_per_process dir_path file_name [optional]`

Check if the provided directory path contains a file-per-process of the provided file name. Assumes the directory is a shared directory.

Takes two arguments with an optional third:

- $1 - Path of the shared directory to check for the files
- $2 - File name without the appended process number
- $3 - Can be given to provided a more precise diagnosis

```
test_expect_success "$dir_name has file-per-process of $file_name" '
    test_path_has_file_per_process $dir_name $file_name
'
```

There are other helper functions available as well, most of which are being used by the test suite itself. Details on these functions can be found in their comments in t/ci/ci-functions.sh.

CHAPTER

NINETEEN

WRAPPER GUIDE

Warning: This document is out-of-date as the process for generating *unifyfs_list.txt* has bugs which causes the generation of *gotcha_map_unifyfs_list.h* to have bugs as well. More information on this can be found in issue #172.

An updated guide and scripts needs to be created for writing and adding new wrappers to UnifyFS.

The files in client/check_fns/ folder help manage the set of wrappers that are implemented. In particular, they are used to enable a tool that detects I/O routines used by an application that are not yet supported in UnifyFS. They are also used to generate the code required for GOTCHA.

- fakechroot_list.txt - lists I/O routines from fakechroot
- gnulibc_list.txt - I/O routines from libc
- cstdio_list.txt - I/O routines from stdio
- posix_list.txt - I/O routines in POSIX
- unifyfs_list.txt - list of wrappers in UnifyFS
- unifyfs_unsupported_list.txt - list of wrappers in UnifyFS that are implemented, but not supported

19.1 unifyfs_check_fns Tool

This tool identifies the set of I/O calls used in an application by running nm on the executable. It reports any I/O routines used by the app, which are not supported by UnifyFS. If an application uses an I/O routine that is not supported, it likely cannot use UnifyFS. If the tool does not report unsupported wrappers, the app may work with UnifyFS but it is not guaranteed to work.

```
unifyfs_check_fns <executable>
```

19.2 Building the GOTCHA List

The gotcha_map_unifyfs_list.h file contains the code necessary to wrap I/O functions with GOTCHA. This is generated from the *unifyfs_list.txt* file by running the following command:

```
python unifyfs_translate.py unifyfs_list
```

19.3 Commands to Build Files

19.3.1 fakechroot_list.txt

The fakechroot_list.txt file lists I/O routines implemented in fakechroot. This list was generated using the following commands:

```
git clone https://github.com/fakechroot/fakechroot.git fakechroot.git
cd fakechroot.git/src
ls *.c > fakechroot_list.txt
```

19.3.2 gnulibc_list.txt

The gnulibc_list.txt file lists I/O routines available in libc. This list was written by hand using information from http://www.gnu.org/software/libc/manual/html_node/I_002fO-Overview.html#I_002fO-Overview.

19.3.3 cstdio_list.txt

The cstdio_list.txt file lists I/O routines available in libstdio. This list was written by hand using information from http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

19.3.4 unifyfs_list.txt

The unifyfs_list.txt file specifies the set of wrappers in UnifyFS. Most but not all such wrappers are supported. The command to build unifyfs list:

```
grep UNIFYFS_WRAP ../src/\*.c > unifyfs_list.txt
```

19.3.5 unifyfs_unsupported_list.txt

The unifyfs_unsupported_list.txt file specifies wrappers that are in UnifyFS, but are known to not actually be supported. This list is written by hand.

CHAPTER

TWENTY

ADDING RPC FUNCTIONS WITH MARGO LIBRARY

In this section, we describe how to add an RPC function using the Margo library API.

Note: The following documentation uses `unifyfs_mount_rpc()` as an example client-server RPC function to demonstrate the required code modifications.

20.1 Common

1. Define structs for the input and output parameters of your RPC handler.

   The struct definition macro `MERCURY_GEN_PROC()` is used to define both input and output parameters. For client-server RPCs, the definitions should be placed in `common/src/unifyfs_client_rpcs.h`, while server-server RPC structs are defined in `common/src/unifyfs_server_rpcs.h`.

   The input parameters struct should contain all values the client needs to pass to the server handler function. The output parameters struct should contain all values the server needs to pass back to the client upon completion of the handler function. The following shows the input and output structs used by `unifyfs_mount_rpc()`.

```
MERCURY_GEN_PROC(unifyfs_mount_in_t,
                 ((int32_t)(dbg_rank))
                 ((hg_const_string_t)(mount_prefix))
                 ((hg_const_string_t)(client_addr_str)))
MERCURY_GEN_PROC(unifyfs_mount_out_t,
                 ((int32_t)(app_id))
                 ((int32_t)(client_id))
                 ((int32_t)(ret)))
```

Note: Passing some types can be an issue. Refer to the Mercury documentation for supported types: https://mercury-hpc.github.io/documentation/ (look under *Predefined Types*). If your type is not predefined, you will need to either convert it to a supported type or write code to serialize/deserialize the input/output parameters. Phil Carns said he has starter code for this, since much of the code is similar.

20.2 Server

1. Implement the RPC handler function for the server.

   This is the function that will be invoked on the client and executed on the server. Client-server RPC handler functions are implemented in `server/src/unifyfs_client_rpc.c`, while server-server RPC handlers go in `server/src/unifyfs_p2p_rpc.c` or `server/src/unifyfs_group_rpc.c`.

   All the RPC handler functions follow the same protoype, which is passed a Mercury handle as the only argument. The handler function should use `margo_get_input()` to retrieve the input parameters struct provided by the client. After the RPC handler finishes its intended action, it replies using `margo_respond()`, which takes the handle and output parameters struct as arguments. Finally, the handler function should release the input struct using `margo_free_input()`, and the handle using `margo_destroy()`. See the existing RPC handler functions for more info.

   After implementing the handler function, place the Margo RPC handler definition macro immediately following the function.

```
static void unifyfs_mount_rpc(hg_handle_t handle)
{
    ...
}
DEFINE_MARGO_RPC_HANDLER(unifyfs_mount_rpc)
```

2. Register the server RPC handler with margo.

   In `server/src/margo_server.c`, update the client-server RPC registration function `register_client_server_rpcs()` to include a registration macro for the new RPC handler. As shown below, the last argument to `MARGO_REGISTER()` is the handler function address. The prior two arguments are the input and output parameters structs.

```
MARGO_REGISTER(unifyfsd_rpc_context->mid,
               "unifyfs_mount_rpc",
               unifyfs_mount_in_t,
               unifyfs_mount_out_t,
               unifyfs_mount_rpc);
```

20.3 Client

1. Add a Mercury id for the RPC handler to the client RPC context.

   In `client/src/margo_client.h`, update the `ClientRpcIds` structure to add a new `hg_id_t <name>_id` variable to hold the RPC handler id.

```
typedef struct ClientRpcIds {
    ...
    hg_id_t mount_id;
}
```

2. Register the RPC handler with Margo.

   In `client/src/margo_client.c`, update `register_client_rpcs()` to register the new RPC handler by its name using `CLIENT_REGISTER_RPC(<name>)`, which will store its Mercury id in the `<name>_id` structure variable defined in the first step. For example:

```
CLIENT_REGISTER_RPC(mount);
```

3. Define and implement an invocation function that will execute the RPC.

   The declaration should be placed in `client/src/margo_client.h`, and the definition should go in `client/src/margo_client.c`.

   ```
   int invoke_client_mount_rpc(unifyfs_client* client, ...);
   ```

   A handle for the RPC is obtained using `create_handle()`, which takes the the id of the RPC as its only parameter. The RPC is actually initiated using `forward_to_server()`, where the RPC handle, input struct address, and RPC timeout are given as parameters. Use `margo_get_output()` to obtain the returned output parameters struct, and release it with `margo_free_output()`. Finally, `margo_destroy()` is used to release the RPC handle. See the existing invocation functions for more info.

Note: The general workflow for creating new RPC functions is the same if you want to invoke an RPC on the server, and execute it on the client. One difference is that you will have to pass *NULL* to the last parameter of *MARGO_REGISTER()* on the server, and on the client the last parameter to *MARGO_REGISTER()* will be the name of the RPC handler function. To execute RPCs on the client it needs to be started in Margo as a *SERVER*, and the server needs to know the address of the client where the RPC will be executed. The client has already been configured to do those two things, so the only change going forward is how *MARGO_REGISTER()* is called depending on where the RPC is being executed (client or server).

CHAPTER

TWENTYONE

INDICES AND TABLES

- genindex
- modindex
- search