ORANGEFS

OrangeFS is an LGPL userspace scale-out parallel storage system. It is ideal for large storage problems faced by HPC, BigData, Streaming Video, Genomics, Bioinformatics.

Orangefs, originally called PVFS, was first developed in 1993 by Walt Ligon and Eric Blumer as a parallel file system for Parallel Virtual Machine (PVM) as part of a NASA grant to study the I/O patterns of parallel programs.

Orangefs features include:

- Distributes file data among multiple file servers
- Supports simultaneous access by multiple clients
- Stores file data and metadata on servers using local file system and access methods
- Userspace implementation is easy to install and maintain
- Direct MPI support
- Stateless

Mailing List Archives

http://lists.orangefs.org/pipermail/devel_lists.orangefs.org/

Mailing List Submissions

devel@lists.orangefs.org

Documentation

http://www.orangefs.org/documentation/

Running ORANGEFS On a Single Server

OrangeFS is usually run in large installations with multiple servers and clients, but a complete filesystem can be run on a single machine for development and testing.

On Fedora, install orangefs and orangefs-server:

```
dnf -y install orangefs orangefs-server
```

There is an example server configuration file in /etc/orangefs/orangefs.conf. Change localhost to your hostname if necessary.

To generate a filesystem to run xfstests against, see below.

There is an example client configuration file in /etc/pvfs2tab. It is a single line. Uncomment it and change the hostname if necessary. This controls clients which use libpvfs2. This does not control the pvfs2-client-core.

Create the filesystem:

```
pvfs2-server -f /etc/orangefs/orangefs.conf
```

Start the server:

```
systemctl start orangefs-server
```

Test the server:

```
pvfs2-ping -m /pvfsmnt
```

Start the client. The module must be compiled in or loaded before this point:

```
systemctl start orangefs-client
```

Mount the filesystem:

```
mount -t pvfs2 tcp://localhost:3334/orangefs /pvfsmnt
```

Userspace Filesystem Source

http://www.orangefs.org/download

Orangefs versions prior to 2.9.3 would not be compatible with the upstream version of the kernel client.

Building ORANGEFS on a Single Server

Where OrangeFS cannot be installed from distribution packages, it may be built from source.

You can omit --prefix if you don't care that things are sprinkled around in /usr/local. As of version 2.9.6, OrangeFS uses Berkeley DB by default, we will probably be changing the default to LMDB soon.

```
./configure --prefix=/opt/ofs --with-db-backend=lmdb --disable-usrint

make

make install
```

Create an orangefs config file by running pvfs2-genconfig and specifying a target config file. Pvfs2-genconfig will prompt you through. Generally it works fine to take the defaults, but you should use your server's hostname, rather than "localhost" when it comes to that question:

```
/opt/ofs/bin/pvfs2-genconfig /etc/pvfs2.conf
```

Create an /etc/pvfs2tab file (localhost is fine):

```
echo tcp://localhost:3334/orangefs /pvfsmnt pvfs2 defaults,noauto 0 0 > \
    /etc/pvfs2tab
```

Create the mount point you specified in the tab file if needed:

```
mkdir /pvfsmnt
```

Bootstrap the server:

```
/opt/ofs/sbin/pvfs2-server -f /etc/pvfs2.conf
```

Start the server:

```
/opt/ofs/sbin/pvfs2-server /etc/pvfs2.conf
```

Now the server should be running. Pvfs2-ls is a simple test to verify that the server is running:

```
/opt/ofs/bin/pvfs2-ls /pvfsmnt
```

If stuff seems to be working, load the kernel module and turn on the client core:

```
/opt/ofs/sbin/pvfs2-client -p /opt/ofs/sbin/pvfs2-client-core
```

Mount your filesystem:

```
mount -t pvfs2 tcp://`hostname`:3334/orangefs /pvfsmnt
```

Running xfstests

It is useful to use a scratch filesystem with xfstests. This can be done with only one server.

Make a second copy of the FileSystem section in the server configuration file, which is /etc/orangefs/orangefs.conf. Change the Name to scratch. Change the ID to something other than the ID of the first FileSystem section (2 is usually a good choice).

Then there are two FileSystem sections: orangefs and scratch.

This change should be made before creating the filesystem.

```
pvfs2-server -f /etc/orangefs/orangefs.conf
```

To run xfstests, create /etc/xfsqa.config:

```
TEST_DIR=/orangefs
TEST_DEV=tcp://localhost:3334/orangefs
SCRATCH_MNT=/scratch
SCRATCH_DEV=tcp://localhost:3334/scratch
```

Then xfstests can be run:

```
./check -pvfs2
```

Options

The following mount options are accepted:

acl
: Allow the use of Access Control Lists on files and directories.

intr
: Some operations between the kernel client and the user space filesystem can be interruptible, such as changes in

debug levels and the setting of tunable parameters.

local_lock
: Enable posix locking from the perspective of "this" kernel. The default file_operations lock action is to return ENOSYS. Posix locking kicks in if the filesystem is mounted with -o local_lock. Distributed locking is being worked on for the future.

Debugging

If you want the debug (GOSSIP) statements in a particular source file (inode.c for example) go to syslog:

```
echo inode > /sys/kernel/debug/orangefs/kernel-debug
```

No debugging (the default):

```
echo none > /sys/kernel/debug/orangefs/kernel-debug
```

Debugging from several source files:

```
echo inode,dir > /sys/kernel/debug/orangefs/kernel-debug
```

All debugging:

```
echo all > /sys/kernel/debug/orangefs/kernel-debug
```

Get a list of all debugging keywords:

```
cat /sys/kernel/debug/orangefs/debug-help
```

Protocol between Kernel Module and Userspace

Orangefs is a user space filesystem and an associated kernel module. We'll just refer to the user space part of Orangefs as "userspace" from here on out. Orangefs descends from PVFS, and userspace code still uses PVFS for function and variable names. Userspace typedefs many of the important structures. Function and variable names in the kernel module have been transitioned to "orangefs", and The Linux Coding Style avoids typedefs, so kernel module structures that correspond to userspace structures are not typedefed.

The kernel module implements a pseudo device that userspace can read from and write to. Userspace can also manipulate the kernel module through the pseudo device with ioctl.

The Bufmap

At startup userspace allocates two page-size-aligned (posix_memalign) mlocked memory buffers, one is used for IO and one is used for readdir operations. The IO buffer is 41943040 bytes and the readdir buffer is 4194304 bytes. Each buffer contains logical chunks, or partitions, and a pointer to each buffer is added to its own PVFS_dev_map_desc structure which also describes its total size, as well as the size and number of the partitions.

A pointer to the IO buffer's PVFS_dev_map_desc structure is sent to a mapping routine in the kernel module with an ioctl. The structure is copied from user space to kernel space with copy_from_user and is used to initialize the kernel module's "bufmap" (struct orangefs_bufmap), which then contains:

- refcnt - a reference counter
- desc_size - PVFS2_BUFMAP_DEFAULT_DESC_SIZE (4194304) - the IO buffer's partition size, which represents the filesystem's block size and is used for s_blocksize in super blocks.
- desc_count - PVFS2_BUFMAP_DEFAULT_DESC_COUNT (10) - the number of partitions in the IO buffer.
- desc_shift - log2(desc_size), used for s_blocksize_bits in super blocks.
- total_size - the total size of the IO buffer.
- page_count - the number of 4096 byte pages in the IO buffer.
- page_array - a pointer to `page_count * (sizeof(struct page*))` bytes of kcalloced memory. This memory is used as an array of pointers to each of the pages in the IO buffer through a call to get_user_pages.
- desc_array - a pointer to `desc_count * (sizeof(struct orangefs_bufmap_desc))` bytes of kcalloced memory. This memory is further intialized:

    user_desc is the kernel's copy of the IO buffer's ORANGEFS_dev_map_desc structure. user_desc->ptr points to the IO buffer.

    ```
    pages_per_desc = bufmap->desc_size / PAGE_SIZE
    offset = 0

    bufmap->desc_array[0].page_array = &bufmap->page_array[offset]
    ```

```
bufmap->desc_array[0].array_count = pages_per_desc = 1024
bufmap->desc_array[0].uaddr = (user_desc->ptr) + (0 * 1024 * 4096)
offset += 1024
                    .
                    .
                    .
bufmap->desc_array[9].page_array = &bufmap->page_array[offset]
bufmap->desc_array[9].array_count = pages_per_desc = 1024
bufmap->desc_array[9].uaddr = (user_desc->ptr) +
                                        (9 * 1024 * 4096)
offset += 1024
```

- buffer_index_array - a desc_count sized array of ints, used to indicate which of the IO buffer's partitions are available to use.
- buffer_index_lock - a spinlock to protect buffer_index_array during update.
- readdir_index_array - a five (ORANGEFS_READDIR_DEFAULT_DESC_COUNT) element int array used to indicate which of the readdir buffer's partitions are available to use.
- readdir_index_lock - a spinlock to protect readdir_index_array during update.

Operations

The kernel module builds an "op" (struct orangefs_kernel_op_s) when it needs to communicate with userspace. Part of the op contains the "upcall" which expresses the request to userspace. Part of the op eventually contains the "downcall" which expresses the results of the request.

The slab allocator is used to keep a cache of op structures handy.

At init time the kernel module defines and initializes a request list and an in_progress hash table to keep track of all the ops that are in flight at any given time.

Ops are stateful:

- unknown
    - op was just initialized
- waiting
    - op is on request_list (upward bound)
- inprogr
    - op is in progress (waiting for downcall)
- serviced
    - op has matching downcall; ok
- purged
    - op has to start a timer since client-core exited uncleanly before servicing op
- given up
    - submitter has given up waiting for it

When some arbitrary userspace program needs to perform a filesystem operation on Orangefs (readdir, I/O, create, whatever) an op structure is initialized and tagged with a distinguishing ID number. The upcall part of the op is filled out, and the op is passed to the "service_operation" function.

Service_operation changes the op's state to "waiting", puts it on the request list, and signals the Orangefs file_operations.poll function through a wait queue. Userspace is polling the pseudo-device and thus becomes aware of the upcall request that needs to be read.

When the Orangefs file_operations.read function is triggered, the request list is searched for an op that seems ready-to-process. The op is removed from the request list. The tag from the op and the filled-out upcall struct are copy_to_user'ed back to userspace.

If any of these (and some additional protocol) copy_to_users fail, the op's state is set to "waiting" and the op is added back to the request list. Otherwise, the op's state is changed to "in progress", and the op is hashed on its tag and put onto the end of a list in the in_progress hash table at the index the tag hashed to.

When userspace has assembled the response to the upcall, it writes the response, which includes the distinguishing tag, back to the pseudo device in a series of io_vecs. This triggers the Orangefs file_operations.write_iter function to find the op with the associated tag and remove it from the in_progress hash table. As long as the op's state is not "canceled" or "given up", its state is set to "serviced". The file_operations.write_iter function returns to the waiting vfs, and back to service_operation through wait_for_matching_downcall.

Service operation returns to its caller with the op's downcall part (the response to the upcall) filled out.

The "client-core" is the bridge between the kernel module and userspace. The client-core is a daemon. The client-core has an associated watchdog daemon. If the client-core is ever signaled to die, the watchdog daemon restarts the client-core. Even though the client-core is restarted "right away", there is a period of time during such an event that the client-core is dead. A dead client-core

can't be triggered by the Orangefs file_operations.poll function. Ops that pass through service_operation during a "dead spell" can timeout on the wait queue and one attempt is made to recycle them. Obviously, if the client-core stays dead too long, the arbitrary userspace processes trying to use Orangefs will be negatively affected. Waiting ops that can't be serviced will be removed from the request list and have their states set to "given up". In-progress ops that can't be serviced will be removed from the in_progress hash table and have their states set to "given up".

Readdir and I/O ops are atypical with respect to their payloads.

- readdir ops use the smaller of the two pre-allocated pre-partitioned memory buffers. The readdir buffer is only available to userspace. The kernel module obtains an index to a free partition before launching a readdir op. Userspace deposits the results into the indexed partition and then writes them to back to the pvfs device.
- io (read and write) ops use the larger of the two pre-allocated pre-partitioned memory buffers. The IO buffer is accessible from both userspace and the kernel module. The kernel module obtains an index to a free partition before launching an io op. The kernel module deposits write data into the indexed partition, to be consumed directly by userspace. Userspace deposits the results of read requests into the indexed partition, to be consumed directly by the kernel module.

Responses to kernel requests are all packaged in pvfs2_downcall_t structs. Besides a few other members, pvfs2_downcall_t contains a union of structs, each of which is associated with a particular response type.

The several members outside of the union are:

`int32_t type`
- type of operation.

`int32_t status`
- return code for the operation.

`int64_t trailer_size`
- 0 unless readdir operation.

`char *trailer_buf`
- initialized to NULL, used during readdir operations.

The appropriate member inside the union is filled out for any particular response.

PVFS2_VFS_OP_FILE_IO
: fill a pvfs2_io_response_t

PVFS2_VFS_OP_LOOKUP
: fill a PVFS_object_kref

PVFS2_VFS_OP_CREATE
: fill a PVFS_object_kref

PVFS2_VFS_OP_SYMLINK
: fill a PVFS_object_kref

PVFS2_VFS_OP_GETATTR
: fill in a PVFS_sys_attr_s (tons of stuff the kernel doesn't need) fill in a string with the link target when the object is a symlink.

PVFS2_VFS_OP_MKDIR
: fill a PVFS_object_kref

PVFS2_VFS_OP_STATFS
: fill a pvfs2_statfs_response_t with useless info <g>. It is hard for us to know, in a timely fashion, these statistics about our distributed network filesystem.

PVFS2_VFS_OP_FS_MOUNT
: fill a pvfs2_fs_mount_response_t which is just like a PVFS_object_kref except its members are in a different order and "__pad1" is replaced with "id".

PVFS2_VFS_OP_GETXATTR
: fill a pvfs2_getxattr_response_t

PVFS2_VFS_OP_LISTXATTR
: fill a pvfs2_listxattr_response_t

PVFS2_VFS_OP_PARAM
: fill a pvfs2_param_response_t

PVFS2_VFS_OP_PERF_COUNT
: fill a pvfs2_perf_count_response_t

PVFS2_VFS_OP_FSKEY
: file a pvfs2_fs_key_response_t

PVFS2_VFS_OP_READDIR
: jamb everything needed to represent a pvfs2_readdir_response_t into the readdir buffer descriptor specified in the upcall.

Userspace uses writev() on /dev/pvfs2-req to pass responses to the requests made by the kernel side.

A buffer_list containing:

- a pointer to the prepared response to the request from the kernel (struct pvfs2_downcall_t).
- and also, in the case of a readdir request, a pointer to a buffer containing descriptors for the objects in the target directory.

... is sent to the function (PINT_dev_write_list) which performs the writev.

PINT_dev_write_list has a local iovec array: struct iovec io_array[10];

The first four elements of io_array are initialized like this for all responses:

```
io_array[0].iov_base = address of local variable "proto_ver" (int32_t)
io_array[0].iov_len = sizeof(int32_t)

io_array[1].iov_base = address of global variable "pdev_magic" (int32_t)
io_array[1].iov_len = sizeof(int32_t)

io_array[2].iov_base = address of parameter "tag" (PVFS_id_gen_t)
io_array[2].iov_len = sizeof(int64_t)

io_array[3].iov_base = address of out_downcall member (pvfs2_downcall_t)
                       of global variable vfs_request (vfs_request_t)
io_array[3].iov_len = sizeof(pvfs2_downcall_t)
```

Readdir responses initialize the fifth element io_array like this:

```
io_array[4].iov_base = contents of member trailer_buf (char *)
                       from out_downcall member of global variable
                       vfs_request
io_array[4].iov_len = contents of member trailer_size (PVFS_size)
                      from out_downcall member of global variable
                      vfs_request
```

Orangefs exploits the dcache in order to avoid sending redundant requests to userspace. We keep object inode attributes up-to-date with orangefs_inode_getattr. Orangefs_inode_getattr uses two arguments to help it decide whether or not to update an inode: "new" and "bypass". Orangefs keeps private data in an object's inode that includes a short timeout value, getattr_time, which allows any iteration of orangefs_inode_getattr to know how long it has been since the inode was updated. When the object is not new (new == 0) and the bypass flag is not set (bypass == 0) orangefs_inode_getattr returns without updating the inode if getattr_time has not timed out. Getattr_time is updated each time the inode is updated.

Creation of a new object (file, dir, sym-link) includes the evaluation of its pathname, resulting in a negative directory entry for the object. A new inode is allocated and associated with the dentry, turning it from a negative dentry into a "productive full member of society". Orangefs obtains the new inode from Linux with new_inode() and associates the inode with the dentry by sending the pair back to Linux with d_instantiate().

The evaluation of a pathname for an object resolves to its corresponding dentry. If there is no corresponding dentry, one is created for it in the dcache. Whenever a dentry is modified or verified Orangefs stores a short timeout value in the dentry's d_time, and the dentry will be trusted for that amount of time. Orangefs is a network filesystem, and objects can potentially change out-of-band with any particular Orangefs kernel module instance, so trusting a dentry is risky. The alternative to trusting dentries is to always obtain the needed information from userspace - at least a trip to the client-core, maybe to the servers. Obtaining information from a dentry is cheap, obtaining it from userspace is relatively expensive, hence the motivation to use the dentry when possible.

The timeout values d_time and getattr_time are jiffy based, and the code is designed to avoid the jiffy-wrap problem:

```
"In general, if the clock may have wrapped around more than once, there
is no way to tell how much time has elapsed. However, if the times t1
and t2 are known to be fairly close, we can reliably compute the
difference in a way that takes into account the possibility that the
clock may have wrapped between times."
```

from course notes by instructor Andy Wang