==========
WHAT IS EXOFS?
==========

exofs is a file system that uses an OSD and exports the API of a normal Linux file system. Users access exofs like any other local file system, and exofs will in turn issue commands to the local OSD initiator.

OSD is a new T10 command set that views storage devices not as a large/flat array of sectors but as a container of objects, each having a length, quota, time attributes and more. Each object is addressed by a 64bit ID, and is contained in a 64bit ID partition. Each object has associated attributes attached to it, which are integral part of the object and provide metadata about the object. The standard defines some common obligatory attributes, but user attributes can be added as needed.

==========
ENVIRONMENT
==========

To use this file system, you need to have an object store to run it on. You may download a target from:
http://open-osd.org

See Documentation/scsi/osd.txt for how to setup a working osd environment.

==========
USAGE
==========

1. Download and compile exofs and open-osd initiator:
   You need an external Kernel source tree or kernel headers from your distribution. (anything based on 2.6.26 or later).

   a. download open-osd including exofs source using:
      [parent-directory]$ git clone git://git.open-osd.org/open-osd.git

   b. Build the library module like this:
      [parent-directory]$ make -C KSRC=$(KER_DIR) open-osd

      This will build both the open-osd initiator as well as the exofs kernel module. Use whatever parameters you compiled your Kernel with and $(KER_DIR) above pointing to the Kernel you compile against. See the file open-osd/top-level-Makefile for an example.

2. Get the OSD initiator and target set up properly, and login to the target. See Documentation/scsi/osd.txt for farther instructions. Also see ./do-osd for example script that does all these steps.

3. Insmod the exofs.ko module:
   [exofs]$ insmod exofs.ko

4. Make sure the directory where you want to mount exists. If not, create it. (For example, mkdir /mnt/exofs)

5. At first run you will need to invoke the mkfs.exofs application

   As an example, this will create the file system on:
   /dev/osd0 partition ID 65536

   mkfs.exofs --pid=65536 --format /dev/osd0

   The --format is optional. If not specified, no OSD_FORMAT will be
   performed and a clean file system will be created in the specified pid,
   in the available space of the target. (Use --format=size_in_meg to limit
   the total LUN space available)

   If pid already exists, it will be deleted and a new one will be created in
   its place. Be careful.

   An exofs lives inside a single OSD partition. You can create multiple exofs
   filesystems on the same device using multiple pids.

   (run mkfs.exofs without any parameters for usage help message)

6. Mount the file system.

   For example, to mount /dev/osd0, partition ID 0x10000 on /mnt/exofs:

        mount -t exofs -o pid=65536 /dev/osd0 /mnt/exofs/

7. For reference (See do-exofs example script):
        do-exofs start - an example of how to perform the above steps.
        do-exofs stop - an example of how to unmount the file system.
        do-exofs format - an example of how to format and mkfs a new exofs.

8. Extra compilation flags (uncomment in fs/exofs/Kbuild):
        CONFIG_EXOFS_DEBUG - for debug messages and extra checks.

==
exofs mount options
==
Similar to any mount command:
        mount -t exofs -o exofs_options /dev/osdX mount_exofs_directory

Where:
    -t exofs: specifies the exofs file system

    /dev/osdX: X is a decimal number. /dev/osdX was created after a successful
               login into an OSD target.

    mount_exofs_directory: The directory to mount the file system on

    exofs specific options: Options are separated by commas (,)
                pid=<integer> - The partition number to mount/create as
                                container of the filesystem.
                                This option is mandatory.
                to=<integer> - Timeout in ticks for a single command.
                                default is (60 * HZ) [for debugging only]

==
DESIGN

* The file system control block (AKA on-disk superblock) resides in an object with a special ID (defined in common.h).
  Information included in the file system control block is used to fill the in-memory superblock structure at mount time. This object is created before the file system is used by mkexofs.c. It contains information such as:
      - The file system's magic number
      - The next inode number to be allocated

* Each file resides in its own object and contains the data (and it will be possible to extend the file over multiple objects, though this has not been implemented yet).

* A directory is treated as a file, and essentially contains a list of <file name, inode #> pairs for files that are found in that directory. The object IDs correspond to the files' inode numbers and will be allocated according to a bitmap (stored in a separate object). Now they are allocated using a counter.

* Each file's control block (AKA on-disk inode) is stored in its object's attributes. This applies to both regular files and other types (directories, device files, symlinks, etc.).

* Credentials are generated per object (inode and superblock) when they are created in memory (read from disk or created). The credential works for all operations and is used as long as the object remains in memory.

* Async OSD operations are used whenever possible, but the target may execute them out of order. The operations that concern us are create, delete, readpage, writepage, update_inode, and truncate. The following pairs of operations should execute in the order written, and we need to prevent them from executing in reverse order:
      - The following are handled with the OBJ_CREATED and OBJ_2BCREATED flags. OBJ_CREATED is set when we know the object exists on the OSD - in create's callback function, and when we successfully do a read_inode.
        OBJ_2BCREATED is set in the beginning of the create function, so we know that we should wait.
            - create/delete: delete should wait until the object is created on the OSD.
            - create/readpage: readpage should be able to return a page full of zeroes in this case. If there was a write already en-route (i.e. create, writepage, readpage) then the page would be locked, and so it would really be the same as create/writepage.
            - create/writepage: if writepage is called for a sync write, it should wait until the object is created on the OSD. Otherwise, it should just return.
            - create/truncate: truncate should wait until the object is created on the OSD.
            - create/update_inode: update_inode should wait until the object is created on the OSD.
      - Handled by VFS locks:
            - readpage/delete: shouldn't happen because of page lock.
            - writepage/delete: shouldn't happen because of page lock.

- readpage/writepage: shouldn't happen because of page lock.

==========
LICENSE/COPYRIGHT
==========
The exofs file system is based on ext2 v0.5b (distributed with the Linux kernel version 2.6.10). All files include the original copyrights, and the license is GPL version 2 (only version 2, as is true for the Linux kernel). The Linux kernel can be downloaded from www.kernel.org.