Next-generation Linux file systems: NiLFS(2) and exofs

Advancing Linux file systems with logs and objects

Skill Level: Intermediate

M. Tim Jones (mtj@mtjones.com)
Independent Author

03 Nov 2009

Linux® continues to innovate in the area of file systems. It supports the largest variety of file systems of any operating system. It also provides cutting-edge file system technology. Two new file systems that are making their way into Linux include the NiLFS(2) log-structured file system and the exofs object-based storage system. Discover the purpose behind these two new file systems and the advantages that they bring.

There's something both exciting and frightening about the announcement of a new Linux file system. It's exciting because file systems represent new territory for interesting advances. It's frightening because a file system in the early stages tends to be experimental and not quite ready for prime time. But sometimes these announcements are about investments in the future of Linux, and a recent announcement for 2.6.30-rc1 indicates a very interesting future, indeed. In the past few quarters, Linux has had three major file system announcements. Late 2008 brought in the B-Tree File System (Btrfs), and more recently, two other unique file systems were introduced: NiLFS(2) and exofs.

File system background

Let's start with a quick introduction to these non-traditional file system approaches, and then explore the specifics of NiLFS(2) and exofs.

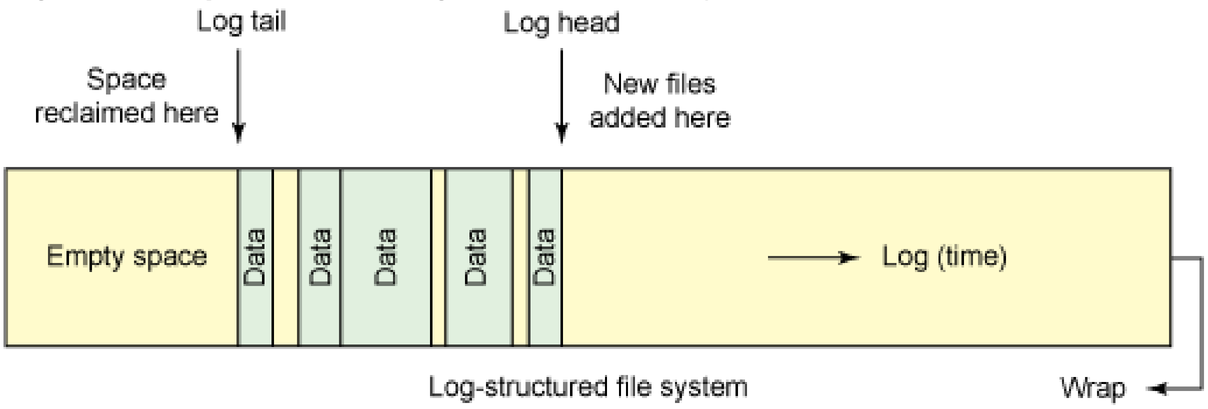
Log-structured file systems

Log-structured file systems and SSDs

Log-structured file systems are an ideal format for solid-state disks (SSDs), which are made up of NAND flash. The fundamental problem with flash is that it has a limited number of write and erase cycles. Because the log writes over the entire device, it levels the writes over the device and thus minimizes erase cycles. For this reason, log-structured file systems perform very well on SSDs (sequential writes) and also provide better wear-leveling.

Log-structured file systems have a rich history in modern computing systems. The first log-structured file system was proposed by John Ousterhout and Fred Douglis in 1988 and subsequently implemented in the Sprite operating system in 1992. As the name implies, a log-structured file system views the file system as a circular log in which new data and file system metadata are written to the head of the log, and free space is reclaimed from the tail (see Figure 1). This means that data may appear two or more times in the log, but as the log is chronologically advancing, the most recent data is viewed as the active data. Having multiple copies of data in the log introduces some interesting benefits, which will be covered in more detail below.

Figure 1. Simple view of a log-structured file system



Although the log-structured approach is an architectural detail more than a selling point, the approach does offer some distinct advantages. One key advantage is recovery from system crashes, which is simpler using the log-structured approach.

Another advantage is the use of the underlying storage system to exploit performance. You might recall that writing to disks sequentially is much faster than random I/O. As all writes occur sequentially, the seek tax is diminished, resulting in faster disk I/O and subsequently a faster file system.

Object-based storage systems

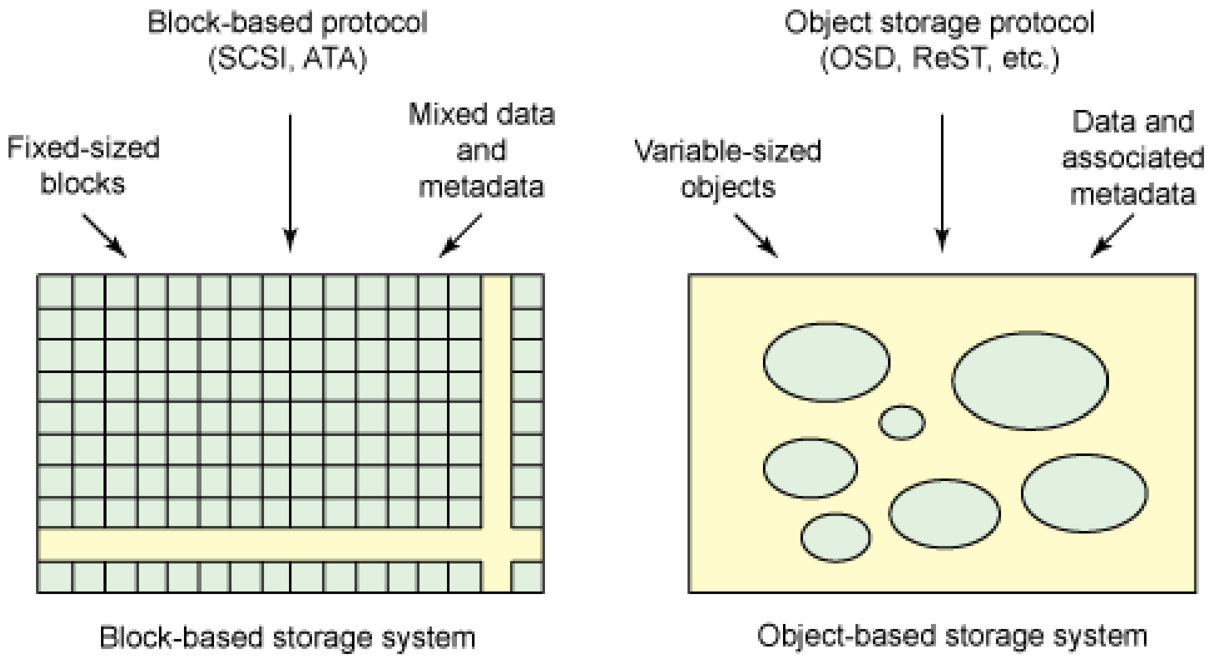
Traditional storage systems rely on disk drives and their native interfaces for persistently storing data. These interfaces rely on block storage semantics, where small, fixed-sized blocks of data are communicated along with their mappings (file system metadata). Object storage systems take a very different approach: instead of managing fixed-size blocks, they manage variable-sized objects and associated metadata (that provides system-level information about the object).

Object-storage devices and standards

Object storage systems are covered under the T-10 Object Storage Devices (OSD) standard. This specification details extensions to the standard SCSI command set to support object-level management. In addition to defining object-level access methods, the specification covers security and metadata management.

Object storage systems are a unique path to very scalable storage that incorporates multi-tenancy and security. OSD as a standard (see sidebar) can be built in a number of ways. You can use OSD-compliant components (such as OSD drives and initiators) or higher-level components (target systems that build OSD behavior over traditional drives). But the fundamental difference between block-based and object-based storage systems is that in block-based, you create objects from collections of blocks that contain both data and metadata using a protocol that communicates with blocks. In object-based, you communicate instead with objects and their associated metadata (see Figure 2). Object storage devices then become flat namespaces of objects, where hierarchy (if necessary) is built higher up in the storage system stack.

Figure 2. Block-based vs object-based storage systems


Block-based protocol (SCSI, ATA)
Fixed-sized blocks
Mixed data and metadata
Block-based storage system
Object storage protocol (OSD, ReST, etc.)
Variable-sized objects
Data and associated metadata
Object-based storage system

This article explores one implementation of a file system over an object-based storage system.

New implementation of a log-structured file system: NiLFS(2)

NiLFS(2) is the second iteration of a log-structured file system developed in Japan by Nippon Telegraph and Telephone (NTT). The file system is under very active development, having recently entered the mainline Linux kernel (in addition to the NetBSD kernel). The first version of NILFS (version 1) appeared in 2005 but lacked any form of garbage collection. In mid-2007, version 2 was first released, which included a garbage collector and the ability to create and maintain multiple snapshots. This year (2009), the NiLFS(2) file system entered the mainline kernel and can be enabled simply by installing its loadable module.

An interesting aspects of NiLFS(2) is its technique of continuous snap-shotting. As NILFS is log structured, new data is written to the head of the log while old data still exists (until it's necessary to garbage-collect it). Because the old data is there, you can step back in time to inspect *epochs* of the file system. These epochs are called *checkpoints* in NiLFS(2) and are an integral part of the file system. NiLFS(2) creates these checkpoints as changes are made, but you can also force a checkpoint to occur.

File systems with snapshots

NiLFS(2) is one of numerous file systems that incorporate snapshot behavior. Other file systems that include snapshots are ZFS, LFS, and Ext3cow.

As I show further down, checkpoints (recovery points) can be viewed as well as changed into a snapshot. Snapshots can be mounted into the Linux file system space just like other file systems, but currently they can be read-only. This is quite useful, as you can mount a snapshot and recover files that were previously deleted or check previous versions of files.

In addition to continuous snapshots, NiLFS(2) provides a number of other benefits. One of the most important from an availability perspective is quick restart. If the current checkpoint was invalid, the file system need only step back to the last good checkpoint for a valid file system. That certainly beats the fsck process.

Challenges of NiLFS(2)

NiLFS(2) version and kernel

This demonstration of NiLFS(2) was done in the 2.6.27 Linux kernel. The 2.6.30-rc1 kernel includes NiLFS(2) in the mainline, but in this case, the NILFS file system module and tools were installed from source. See Resources for information on how to install NiLFS(2) into your kernel.

Although continuous snapshot is a great feature, a cost is associated with it. The upside, as I mentioned, is that it's log structured, so writes are sequential in nature (minimizing seek behavior of the physical disk) and thus very fast. The downside is that it's log structured, and garbage collection is needed to clean up old data and metadata. Normally, the file system operates very quickly, but when garbage collection is required, performance slows down.

Exploring NiLFS(2)

Let's look at NiLFS(2) in action. This demonstration shows how to create an NiLFS(2) file system in a loop device (a simple method to test file systems), then looks at some of the NiLFS(2) features. Start by installing the NiLFS(2) kernel module:

```
$ sudo modprobe nilfs2
$
```

Next, create a file that will contain the file system (an area on the host operating system that you mount as its own file system through the loop device), then build the NiLFS(2) file system within it using `mkfs` (see Listing 1).

Listing 1. Preparing the NiLFS(2) file system

```
$ dd if=/dev/zero of=/tmp/disk.img bs=384M count=1
1+0 records in
1+0 records out
402653184 bytes (403 MB) copied, 60.7253 s, 6.6 MB/s
$ mkfs.nilfs2 /tmp/disk.img
mkfs.nilfs2 ver 2.0
Start writing file system initial data to the device
       Blocksize:4096  Device:/tmp/disk.img  Device Size:402653184
File system initialization succeeded !!
$
```

You now have your disk image initialized with the NiLFS(2) file system format. Next, mount the file system onto a mount point using the loop device (see Listing 2). Note that when the file system is mounted, a user-space program called `nilfs_cleanerd` is also started to provide garbage collection services.

Listing 2. Mounting NiLFS(2) using the loop device

```
$ sudo losetup /dev/loop0 /tmp/disk.img
$ sudo mkdir /mnt/nilfs
$ sudo mount -t nilfs2 /dev/loop0 /mnt/nilfs/
mount.nilfs2: WARNING! - The NILFS on-disk format may change at any time.
mount.nilfs2: WARNING! - Do not place critical data on a NILFS filesystem.
```

```
$ ls /mnt/nilfs
$
```

Now, add a few files to the file system, and then use the `lscp` command to list the current checkpoints available (see Listing 3). You define a snapshot using the `mkcp` command, and then look at the checkpoints again. At the second `lscp`, you can see your newly created snapshot (with all checkpoints and snapshots having a CNO, or checkpoint number).

Listing 3. Listing checkpoints and creating a snapshot

```
$ cd /mnt/nilfs
$ sudo touch file1.txt file2.txt
$ lscp
                 CNO        DATE     TIME  MODE  FLG   NBLKINC       ICNT
                   1  2009-08-21 22:29:31   cp    -         11          3
                   2  2009-08-21 22:36:44   cp    -         11          5
$ sudo mkcp -s
$ lscp
                 CNO        DATE     TIME  MODE  FLG   NBLKINC       ICNT
                   1  2009-08-21 22:29:31   cp    -         11          3
                   2  2009-08-21 22:36:44   ss    -         11          5
                   3  2009-08-21 22:39:47   cp    i          7          5
$
```

Now that you have a snapshot, add a few more files to your current file system, again with the `touch` command (see Listing 4).

Listing 4. Adding a few more files to your NiLFS(2) file system

```
$ sudo touch file3.txt file4.txt
$ ls
file1.txt  file2.txt  file3.txt  file4.txt
$
```

Now, mount your snapshot as a read-only file system. You do this similarly to your previous mount, but in this case, you need to specify the snapshot to mount. You do this with the `cp` option. Note from your prior `lscp` that your snapshot was `CNO=2`. Use this CNO for the `mount` command to mount the read-only file system. Once mounted, you first `ls` your mounted read/write file system and see all files. In the read-only snapshot, you see only the two files that existed at the time of the snapshot (see Listing 5).

Listing 5. Mounting the read-only NiLFS(2) snapshot

```
$ sudo mkdir /mnt/nilfs-ss2
$ sudo mount.nilfs2 -r /dev/loop0 /mnt/nilfs-ss2/ -o cp=2
$ ls /mnt/nilfs
file1.txt  file2.txt  file3.txt  file4.txt
$ ls /mnt/nilfs-ss2/
file1.txt  file2.txt
$
```

Note that these snapshots persist once converted from checkpoints. Checkpoints can be reclaimed from the file system when space is needed, but snapshots are persistent.

This demonstration showed two of the command-line utilities for NiLFS(2): `lscp` (list checkpoints and snapshots) and `mkcp` (make checkpoint or snapshot). There's also a utility called `chcp` for converting a checkpoint to a snapshot or vice-versa, and `rmcp`to invalidate a checkpoint or snapshot.

Given the temporal nature of the file system, NTT has considered some other very innovative tools for the future—for example, `tls` (temporal `ls`), `tdiff` (temporal `diff`), and `tgrep` (temporal `grep`). Introducing time-based functionality seems to be the logical next step.

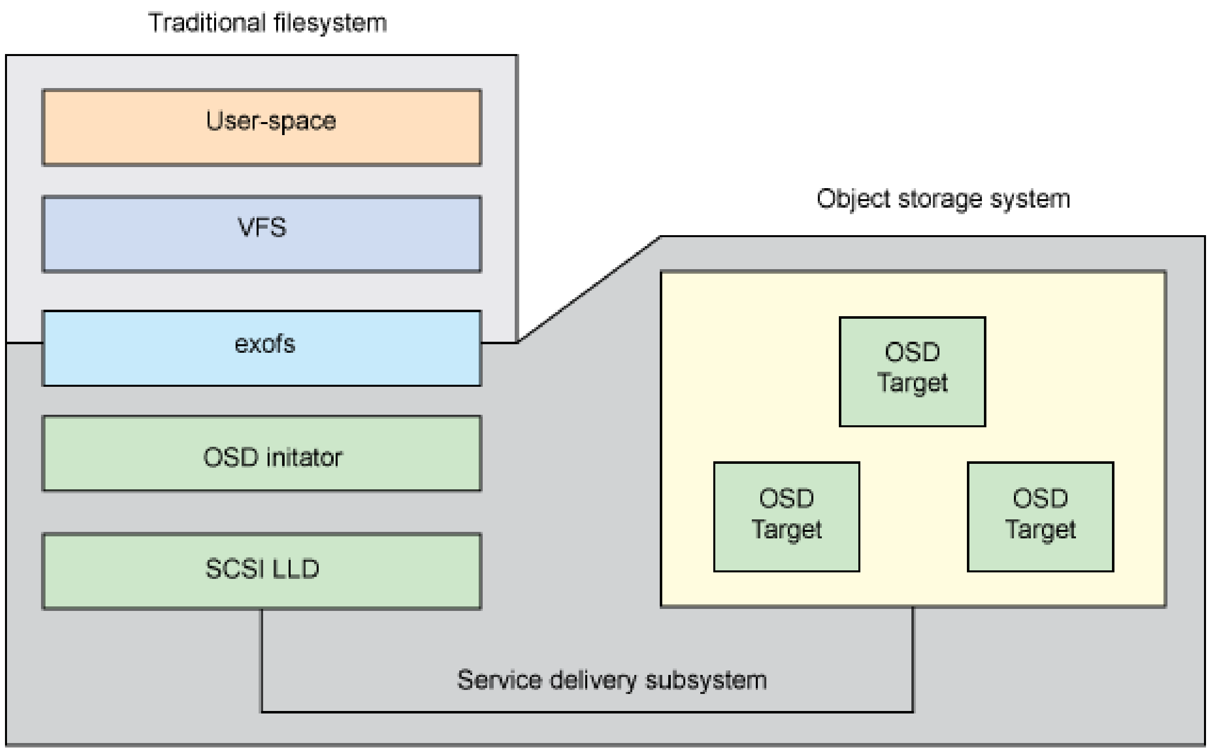
The Extended Object File System (exofs)

The Extended Object File System (exofs) is a traditional Linux file system built over an object storage system. Exofs was initially developed by Avnishay Traeger of IBM and at that time was called the *OSD file system,* or osdfs. Panasas (a company that builds object storage systems) has since taken over the project and renamed it exofs (as its ancestry is from the ext2 file system).

A file system over an object storage system

Conceptually, an object storage system can be viewed as a flat namespace of objects and their associated metadata. Compare this to traditional storage systems based on blocks, with metadata occupying some blocks to provide the semantic glue. At a high level, exofs is built as shown in Figure 3. The Virtual File System Switch (VFS) provides the path to exofs, where exofs communicates with the object storage system through a local OSD initiator. The OSD initiator implements the OSD T-10 standard SCSI command set.

Figure 3. High-level view of the exofs/OSD ecosystem



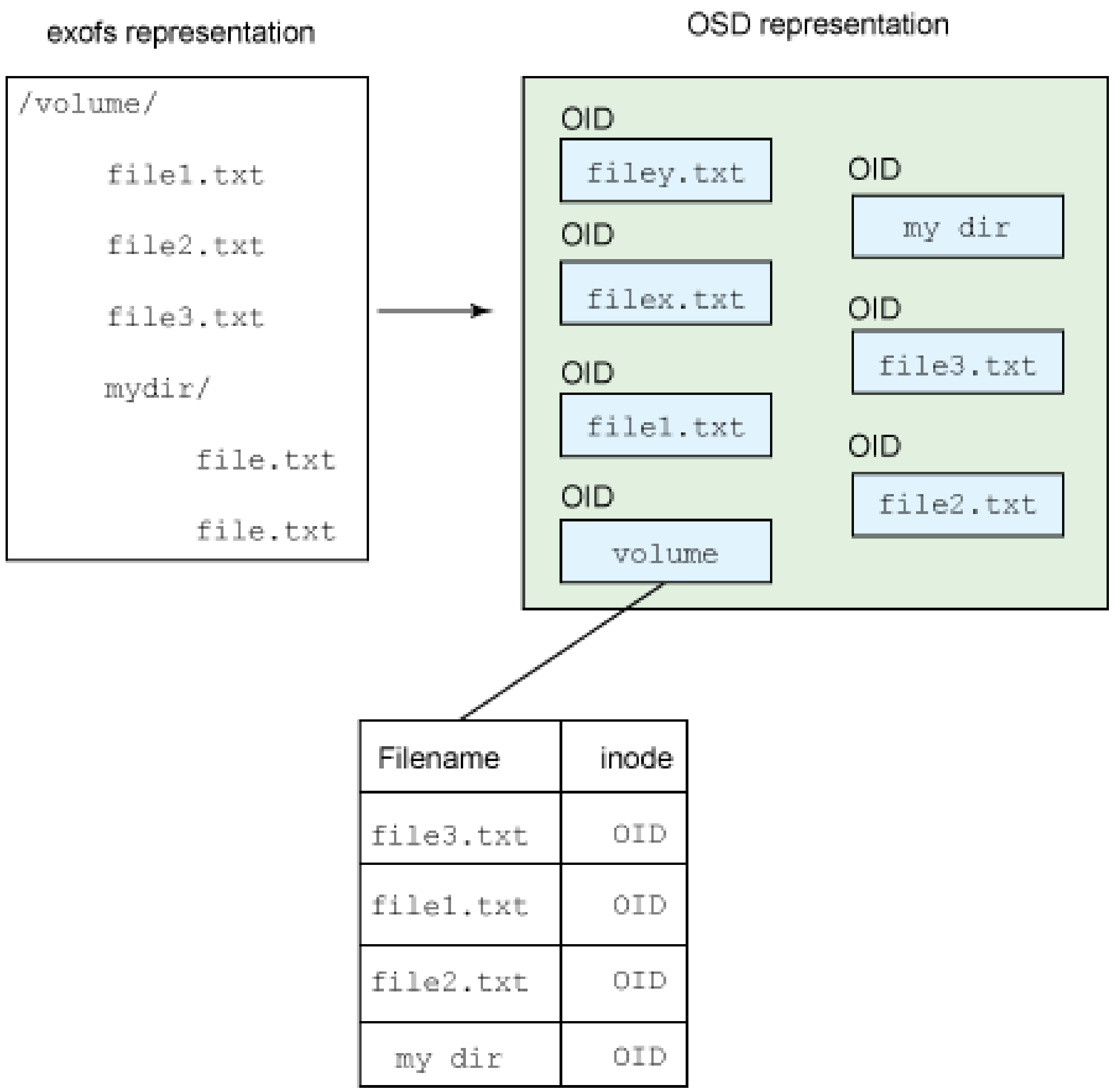
The idea behind exofs is to provide a traditional file system over an OSD backstore. In this way, it's easier to migrate to object-level storage, because the file system presented is a traditional file system.

File system mapping

Each object in an OSD is represented by a 64-bit identifier in a flat namespace. To overlay the standard POSIX interface onto an object storage system, a mapping is required. Exofs provides a simple mapping that is also scalable and extensible.

Files within a file system are represented uniquely as inodes. Exofs maps inodes to the object identifiers (OIDs) in the object system. From there, objects are used to represent all the elements of the file system. Files are mapped directly to objects, and directories are simply files that reference the files contained within the directory (as file name and inode-OID pairs). This is illustrated in simple form in Figure 4. Other objects exist to support things like inode bitmaps (for inode allocation).

Figure 4. High-level view of OSD representations



The OIDs used to represent objects within the object space are 64 bits in size, thereby supporting a large space of objects.

Why object storage?

Object storage is an interesting idea and makes for a much more scalable system. It removes portions of the file system from the host and pushes them into the storage subsystem. There are trade-offs here, but by distributing portions of the file system to multiple endpoints, you distribute the workload, making the object-based method simpler to scale to much larger storage systems. Rather than the host operating system needing to worry about block-to-file mapping, the storage device itself provides this mapping, allowing the host to operate at the file level.

Object storage systems also provide the ability to query the available metadata. This provides some additional advantages, because the search capability can be distributed to the endpoint object systems.

Object storage has made a comeback recently in the area of cloud storage. Cloud storage providers (which sell storage as a service) represent their storage as objects instead of the traditional block API. These providers implement APIs for object transfer, management, and metadata management.

What's ahead?

Although NiLFS(2) and exofs will be interesting and useful additions to the Linux file system inventory, there's more on the way. We've seen Btrfs introduced recently (from Oracle), which offers a Linux alternative to Sun Microsystems' Zettabyte File System (ZFS). Another recent file system is Ceph, which provides a reliable POSIX-based distributed file system with no single point of failure. Today, we find a new log-structured file system and the introduction of a file system over an object store. Linux continues to prove that it's the research platform of choice as well as an enterprise-class operating system.

Resources

Learn

- Visit the NiLFS(2) file system project Web site for the latest information. At this site, you'll find an FAQ, links, and a sample demonstration of the file system, including checkpoints and snapshots.
- Exofs was initially developed by IBM, with continuing development provided by Panasas. Visit the Exofs project Web site at open-osd.org for details.
- Learn more about Btrfs in the Btrfs wiki and also in "Linux kernel advances" (developerWorks, March 2009).
- Read "The Design and Implementation of a Log-Structured File System" (PDF), the seminal paper by the authors of the Sprite LFS, for a great historical perspective on log-structured file systems. This paper introduces the ideas and challenges behind log-structured file systems.
- See Wikipedia for intros to both Log-structured file systems and object storage devices. Along with the current state of the art, these references provide some historical perspective on the technologies.
- Check out the NILFS Web site for instructions on how to install NiLFS(2) and user-land tools into a pre-2.6.30 kernel. You'll find instructions for a variety of Linux distributions.
- Relevant notes for both NiLFS(2) and exofs are in the Documentation/filesystems subdirectory in the Linux kernel tree. It provides a wealth of information for the specific kernel version.
- The presentation "About SSD" (PDF) from Samsung Electronics provides an interesting review of SSD performance using SSD drives for a variety of file systems, including NILFS. Phoronix also provides a more recent analysis of file system performance, comparing EXT4, Btrfs, EXT3, XFS, and NiLFS(2).
- This white paper from Seagate, "The Advantages of Object-Based Storage—Secure, Scalable, Dynamic Storage Devices," provides a great introduction to object storage devices. You can also learn more about object storage from this Storage Networking Industry Association (SNIA) presentation on "Object Storage and Applications" (PDF) and this presentation on "OSD Architecture and Systems" (PDF).
- Seagate developed a prototype Fibre Channel drive that implemented the OSD command set. This drive was demonstrated with IBM and Emulex in the context of an object storage system. IBM provided metadata servers, while Seagate provided OSD-enabled drives and Emulex provided FC host-bus adapters enabled with OSD support.
- Browse all of Tim's articles on developerWorks.

- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the My developerWorks community; with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

About the author

M. Tim Jones

M. Tim Jones is an embedded firmware architect and the author of *Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective.* His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Senior Architect for Emulex Corp. in Longmont, Colorado.