Ted Ts'o on Linux File Systems

An Interview

RIK FARROW

Rik Farrow is the Editor of *;login:*.
rik@usenix.org

Theodore Ts'o is the first North American Linux Kernel Developer, having started working with Linux in September 1991. He also served as the tech lead for the MIT Kerberos V5 development team, and was the architect at IBM in charge of bringing real-time Linux in support of real-time Java to the US Navy. He previously served as CTO for the Linux Foundation and is currently employed at Google. Theodore is a Debian Developer and is the maintainer of the ext4 file system in the Linux kernel. He is the maintainer and original author of the e2fsprogs userspace utilities for the ext2, ext3, and ext4 file systems.
tytso@mit.edu

I ran into Ted Ts'o during a tutorial luncheon at LISA '12, and that later sparked an email discussion. I started by asking Ted questions that had puzzled me about the early history of ext2 having to do with the performance of ext2 compared to the BSD Fast File System (FFS).

I had met Rob Kolstad, then president of BSDi, because of my interest in the AT&T lawsuit against the University of California and BSDi. BSDi was being sued for, among other things, having a phone number that could be spelled 800-ITS-UNIX. I thought that it was important for the future of open source operating systems that AT&T lose that lawsuit.

That said, when I compared the performance of early versions of Linux to the current version of BSDi, I found that they were closely matched, with one glaring exception. Unpacking tar archives using Linux (likely .9) was blazingly fast compared to BSDi. I asked Rob, and he explained that the issue had to do with synchronous writes, finally clearing up a mystery for me.

Now I had a chance to ask Ted about the story from the Linux side, as well as other questions about file systems.

Rik: After graduating from MIT, you stayed on, working in the Kerberos project. But you also wrote the e2fsprogs, which include e2fsck for ext2, at about the same time. Can you tell me how you got involved with writing key file-system utilities for the newly created Linux operating system?

Ted: I originally got started with file systems because I got tired of how long it took for ext2's fsck to run. Originally, e2fsck was an adaption of the fsck for MINIX, which Linus Torvalds had written. It was very inefficient, and read inodes in from disks multiple times.

So I got started with file systems by deciding to create a new fsck for ext2 from scratch. I did this by creating a library called libext2fs that allowed programs to easily manipulate the file system data structures. This library was originally designed for use by e2fsck, but I anticipated that it would also be useful for other applications, including dump/restore, ext2fuse, etc.

I also made a point of creating a very robust set of regression tests for fsck, consisting of small file system images with specific file system corruptions, which I could use to make sure e2fsck would be able to handle those file system corruptions correctly. As far as I know, it is one of the only file system checkers (even today) that has a regression test suite.

For the design of how to check the file system, and the order of the file system scans, I took most of my inspirations from the Bina and Emrath paper "A Faster fsck for BSD Unix" [3]. This paper described improvements to the BSD fsck for the Fast File System. As far as I know, its ideas were never adopted into BSD 4.x's fsck, probably because the changes it suggested were too disruptive. Since I was doing a reimplementation from scratch, though, it was a lot easier for me to use their ideas in e2fsprogs.

So that's how I got involved with file-system development. I started by creating the e2fsprogs utilities, and then I gradually switched the focus of my kernel development efforts from the tty layer and serial driver to the ext2 and later the ext4 file system code.

Rik: The difference between sync and async had mystified me since I first read the Lions code [1]. Later, I read the source code to BSD fsck, as well as the paper, and still didn't get it. Both express the idea that certain types of damage to the file system will not happen, and when others do, assumptions can be made about how this damage occurred. I later learned that they were talking about what would be called "ordered writes." How did Linux avoid these issues in ext2?

Ted: So the concern behind asynchronous FFS, and what BSD folks constantly would say was inherently dangerous with ext2fs, is what would happen with a power failure. Various BSD folks swore up and down that after a power failure or a kernel crash, ext2 users would surely lose data. To avoid that, with the synchronous FFS, metadata updates were written in a very careful order so that fsck could always figure out how to either roll back or complete a metadata operation.

I was never all that persuaded by this argument, since if you didn't use fsync(), the data blocks weren't getting forced to disk, so who cares if the metadata blocks were written in a very careful order? Furthermore, in practice, all modern disks (especially those engineered to get the best WinBench scores) work with writeback caching enabled, and in fact are optimized assuming that writeback caching is enabled, and this defeats the careful write ordering of FFS synchronous mode (or soft updates). You can disable writeback caching, of course, but on modern disks, this will very badly trash write performance.

For more information on soft updates and why we aren't all that impressed with it as a technology, please see Val Henson's LWN piece [2]. The one amplification I'll add to it is that soft updates are so complicated, I very much doubt there are many BSD developers beyond Greg Ganger and Kirk McKusick who understand it well enough to add new file system features. As a result, UFS didn't get extended attributes or ACL support until Kirk personally added it himself, and as far as I know, it still doesn't have online file-system resizing, while ext3/4 have had online resizing since 2004.

Rik: Ext2 had been in use for years when the decision was made to create ext3. Can you talk about the decision-making process, about what changes were needed to justify a newer version of the venerable ext2?

Ted: As hard drives started getting bigger (which back in 2000 to 2001 meant drives with 40 GB to 80 GB), the time to run fsck got larger and larger. As a result, there was a strong desire to have file systems that did not require running fsck after a power failure. So there wasn't an explicit decision-making process, as much as there was a cry from the user community demanding this feature.

Between 2010 and 2011, there were multiple efforts launched to add a journaling file system to Linux: ReiserFS, ext3, and JFS. ReiserFS actually got merged into the Linux kernel first, in version 2.4.1; however, it wasn't fully stable when it hit the mainline. Stephen Tweedie didn't submit ext3 for merging until 2.4.15, in November 2011, and at that time ext3 and ReiserFS were roughly at the same level of stability. JFS was merged into the mainline a bit later, in version 2.4.18, although like ext3, it was available in preview releases before it was deemed ready to be submitted to Linus.

For a while, ReiserFS was favored by SUSE (it had helped to fund the development of ReiserFS by Namesys) while ext3 was favored by Red Hat (since Stephen Tweedie, the primary author of the journaling feature, worked for Red Hat). JFS was donated by IBM, and at least initially it actually had somewhat better performance and scalability than the other two file systems; however, the only people who understood it were IBM employees, whereas ext3 had a much wider developer pool. As a result, it evolved faster than its siblings, and eventually became the de facto default file system for Linux.

Rik: You were involved in the decision-making process to go ahead with the design and implementation of Btrfs [4]. Can you tell us a bit about that process and your part in it?

*Ted:*At the time, Sun's ZFS was receiving a lot of attention due to a large marketing push from Sun Microsystems. So a group of Linux file system engineers, representing multiple file systems and working at many different companies, got together in November 2007 to design the requirements for a "next generation file system" with the goal of convincing the management from multiple companies to work together to make a new file system possible.

The consensus of the engineers who attended this workshop was that adding simple extensions to ext3 (to create ext4) was the fastest way to improve the capabilities of Linux's file system; however, in the long term, it would be necessary to create a new file system to provide support for more advanced features such as file system-level snapshots, and metadata and data checksumming. A number of potential file system technologies were considered for the latter, including the Digital Equipment Corporation's Advanced File System (AdvFS, which HP/Compaq had offered to make available under an open source license). Ultimately, though, Chris Mason's Btrfs was thought to have the best long-term prospects.

I warned people at the workshop that from surveying the past experience from other file system development efforts, creating an enterprise-ready, production file system would probably require 50–200 person years of efforts, and five years of calendar time. For example, the development of ZFS started in 2001, but it was not shipped as part of Solaris 10 until 2006;

however, there was concern that companies would not fund it if we stated that it would take that long, so many folks tried to claim that Btrfs would be ready in only 2–3 years. As it turns out, distributions have only been willing to support Btrfs as ready for consumer use in fall of 2012, which was indeed about five years—and it will probably be at least another two years before most system administrators will be willing to start using it on mission-critical systems.

People tend to underestimate how long it takes to get a file system ready for production use; finding and fixing all of the potential problems, and then doing performance and scalability tuning, takes a long time.

Rik: I've heard that while Google uses ext4, the extent-based version of ext3, for cluster file systems like GFS, you disable journaling. Could you explain the reasoning behind that?

Ted: It's a question of costs versus benefits. Journaling requires extra disk writes, and worse, journal commits require atomic writes, which are extremely expensive. To give an extreme example, consider a benchmark where 10K files are written with an fsync() after each file is written. Regardless of whether ext3, ext4, XFS, or ReiserFS is used, only about 30 files per second can be written; that's because the bottleneck is the CACHE FLUSH command. With journaling disabled, 575 files can be written per second. I used the following fs_mark [5] line for testing:

```
fs_mark -s 10240 -n 1000 -d /mnt
```

Against these costs, what are the benefits? First, journaling allows a machine to reboot after a crash much more quickly, since the need to run fsck on all of the file systems can be avoided. Secondly, journaling provides a guarantee that any files written after an fsync() will not be lost after a crash. Mail servers rely on this guarantee quite heavily to ensure that email won't be lost; however, cluster file systems need to survive far more than a crash of a single machine. When there are hundreds or thousands of servers, the likelihood that a hard drive dies, or a power supply fails, or a network switch servicing a rack of servers quits, is quite high. To deal with this, Google File System (GFS) stores redundant copies of each 64 MB chunk across multiple servers, distributed in such a way so that a single failure—whether of a hard drive, an entire server, or the loss of a network switch—will not cause any interruption of service. The GFS chunkserver also maintains checksums of each 64K block; if any data corruption is found, GFS will automatically fetch the chunk from another chunkserver.

Given that GFS has all of this redundancy to protect against higher level failures, the benefits of journaling at the local disk file system level are redundant. And so, if we don't need the benefits of journaling, why pay the costs of journaling? It is for this reason that Google used ext2 and never bothered switching to ext3. The ext4 file system has extent-mapped files, which are more efficient than the files mapped using indirect blocks. This is more efficient both for reading and writing files, as well as when running the file system checker (fsck). An ext2 or ext3 file system that requires 45 minutes to fsck might only require 4 or 5 minutes if the same set of files is stored on the same disk using ext4 instead.

Rik: HDD vendors are preparing to launch Shingled Magnetic Recording (SMR) drives, which have increased capacities but may perform best when the file system understands the issue of working with SMR. Do you know of any plans to support these drives in host- or coop-managed mode in Linux?

Ted: There is a saying: "Those who try to use flash as a fast disk, generally tend to be very happy. Those who try to use flash as slow memory, tend to be very frustrated." I suspect the same will be true with shingled drives. Specifically, people who use SMR drives as very fast tape drives will be very happy; however, people who try to use shingled drives as slow disk drives will be very frustrated.

For many applications, we are no longer constrained by hard drive capacity, but by seek speeds. Essentially, a 7200 RPM hard drive is capable of delivering approximately 100 seeks per second, and this has not changed in more than 10 years, even as disk capacity has been doubling every 18–24 months. In fact, if you have a big data application which required a half a petabyte of storage, what had previously required 1024 disk drives when we were using 500 GB drives, now that 3 TB disks are available, only 171 disk drives are needed. So a storage array capable of storing half a petabyte is now capable of 80% fewer seeks.

SMR drives make this problem far, far worse. As a result, so far, I'm not aware of a lot of work with shingled drives in the Linux development community. There are some research groups that are experimenting with SMR drives, but at the moment, there has been much more interest in trying to use flash devices—either very high speed, PCIe-attached flash, or dealing with the limitations of extremely inexpensive MMC flash found in mobile or consumer electronics devices.

Rik: Finally, why are there so many file systems?

Ted: There are lots of different reasons. Sometimes we have file systems for interoperability / data exchange. That explains file systems such as FAT/VFAT/MS-DOS, iso9660, HFS, NTFS, MINIX FS, FreeVXFS, BFS, QNX4, QNX6, etc. When you take a look at the list of file systems, there are more of those than most people might first suspect.

Then there are all of the various network file systems, and the reason why we have so many is because of interoperability requirements. So that explains NFS, NFS4, CIFS, AFS, 9P, etc. And the various cluster file systems: GFS2, OCFS2, and Ceph. There probably isn't much excuse for the existence of both GFS2

and OCFS2 since they fill essentially the same niche, but that's more about an accident of timing—the same reason why we have both Git and Mercurial.

There are also a number of pseudo file systems where the file system abstraction is really useful: hugetlbfs, ramfs, debugfs, sysfs, dlm, etc.

And, of course, there are the file systems used essentially for specialized bootup applications: cramfs, SquashFS, romfs.

Finally, we have file systems that are highly specialized for a specific use case, such as file systems that are designed to work on raw flash interfaces (MTD), or a file system designed to work on object-based storage devices (EXOFS).

Basically, "it's complicated," and there are a lot of different reasons.

References

[1] Lions' Commentary on UNIX 6th Edition, with Source Code: http://en.wikipedia.org/wiki/Lions%27_Commentary_on_UNIX_6th_Edition,_with_Source_Code.

[2] Val Aurora (Henson), "Soft Updates, Hard Problems": http://lwn.net/Articles/339337/.

[3] E. Bina and P. Emrath, "A Faster fsck for BSD Unix," Proceedings of the USENIX Winter Conference, January 1989.

[4] Josef Bacik, "Btrfs: The Swiss Army Knife of Storage," *;login:*, February 2012: https://www.usenix.org/publications/login/february-2012/btrfs-swiss-army-knife-storage.

[5] fs_mark: http://sourceforge.net/projects/fsmark/.