

From ext3 to ext4 An Interview with Theodore Ts'o

Jeffrey B. Layton

Jeff Layton talks with Theodore Ts'o about getting the best performance out of your file system, painless migration and the work still to do.

While you can read the on-line documentation and articles about ext4, you can gain some important perspective by going directly to the horse's mouth. Jeff Layton talks with Theodore Ts'o to talk about designing ext4, painless migration and the work still to do.

Jeff Layton: What original design goals did you have for ext4?

Theodore Ts'o: There were a number of features that we've wanted to add to ext3 — to improve performance, support larger number of blocks, etc. — that we couldn't without breaking backwards compatibility, and which would take a long enough time to stabilize that we couldn't make those changes to the existing ext3 code base. Some of those features included: extents, delayed allocation, the multiblock allocator, persistent preallocation, metadata checksums, and online defragmentation.

Along the way we added some other new features that were easy to add, and didn't require much extra work, such as NFSv4 compatible inode version numbers, nanosecond timestamps, and support for running without a journal — which was a feature that was important to Google, and which was contributed by a Google engineer. This wasn't something we were planning on adding, but the patch was relatively straightforward, and it meant Google would be using ext4 and providing more testing for ext4, so it was a no-brainer to add that feature to ext4.

JL: What design goals were not met in the current version of ext4? Why did they not make it into the current version?

TT: The biggest thing which is not yet done on the kernel side is online defragmentation. Unfortunately, that work was contributed by developers who were relatively new to ext2/3/4 filesystem development, and the patches had a number of problems. They've been rewritten, and they are a lot better, but the patches are still not quite mainline ready. Hopefully we'll be able to get those patches whipped into shape and merged in the near future, though.

The other piece which is still not quite finish is support for large block numbers. The kernel code is there, and it's been lightly tested, but the e2fsprogs support for 64-bit block numbers has not yet been merged into mainline. Patches exist, but I haven't had the time I've needed to do the necessary Q/A and to get those patches merged into e2fsprogs' mainline. Again, that should hopefully happen soon. [Click here to read about setting up and benchmarking an ext4 file system.](#)

JL: Ext4 can be used as an upgrade path for ext3. Was this one of the top design goals and was there any consideration given to something completely new and not interoperable with ext3?

TT: One of our primary design goals was that it should be painlessly easy to upgrade from ext3 to ext4. You might not get all of the benefits of ext4 unless you do a backup/reformat/restore of your filesystem, but you would get at least some of the benefits by simply remounting the filesystem using ext4 and enabling some of ext4's features.

We didn't really consider doing something competely new and totally incompatible with ext3 because part of the goal of ext4 was to have something that could be stablized fairly quickly. The reality is that it takes years before a completely new filesystem to be considered stable enough for use in an enterprise environment, and we wanted something that could be ready as quickly as possible.

From ext3 to ext4 An Interview with Theodore Ts'o Jeffrey B. Layton

Besides, there are other efforts, such as btrfs, which are starting from scratch, and btrfs will have new features, such as filesystem-level snapshots, and the equivalent of dynamic inode tables, that ext4 could never have because we wanted to stick to the tried-and-true ext3 design — with enhancements, to be sure, but in many critical ways ext4 doesn't really deviate from ext3 in that we still use ext3's physical block journaling, ext3's fixed inode table, and bitmaps for inode and block allocation.

You can think of ext4 as being an exercise to see how much a tried-and-true filesystem design could be stretched while still retaining the fundamental BSD-style FFS architecture.

JL: In some of the article I have read on the web, there is some mention about being able to modify ext4 in the future for 64-bits to go above the 1 EB range. While not committing yourself to any comments ala' Bill Gates and 640KB of memory, do you think it's possible we'll see a need for 64-bit block addressing? Would this become ext5?

TT: Ext4's kernel code supports 48 bit block numbers; using 4k block sizes, that gives a maximum filesystem size of 1 Exabyte. One of the reasons why we decided to stick with this was out of consideration of the Clusterfs folks, who contributed the extents and delayed allocation code. Since they have customers using Lustre that utilize this format, we decided to keep on-disk compatibility so that Lustre users could easily migrate their server filesystems to use ext4.

It would be relatively easy to add an alternate extent format to support 64-bit block numbers, and we may end up doing that at some point. The e2fsprogs code was written to easily support multiple extent formats; the kernel code is less flexible, but if this were to become an issue, we could add this support easily enough. I wouldn't really consider this "ext5"; it would probably just be an additional feature for ext4.

JL: Is there any thought being given to adapting ext4 to SSDs? If so, what concepts are being thrown around?

TT: I've actually written a whole series of blog posts on this subject, which you can see [here](#). Part of the problem right now is that SSD's are still under going major changes. For example, if you are using Intel's new SSD's, the X25-M and X25-E, pretty much no changes seem to be necessary. Ext4 has support for the ATA TRIM command, which allows filesystems to inform SSD's that blocks have been deleted and do not need to be taken into account by the SSD's garbage collection and wear-leveling algorithms. Unfortunately the ATA TRIM command hasn't been finalized yet, and so (as of today) there are no drives, including Intel's SSD's that actually support the ATA TRIM command; and for this reason Linux's block device layer does not currently issue the ATA TRIM command, since there haven't been any devices to test the command. So at the moment, ext4 informs the block layer that blocks that belong to deleted files can be discard, so once TRIM-capable SSD's become available, and the Linux block layer actually sends the TRIM command to the hard drives, everything will be all set to go.

However, even without TRIM support, the X25-M SSD works very well on ext4 today. I have one installed in my laptop, and it works just fine. Unfortunately, older SSD's do not work so well on ext2/3/4. It will be interesting to see how well the next generation of SSD's work on ext4. For example, I expect SanDisk and OCZ to both release new SSD's fairly soon. Both of these SSD manufacturers haven't stated how their new SSD's will compare to Intel's SSD offerings, but hopefully they will have comparable features. If so, it may not be worth it to try to optimize ext4 for "legacy" SSD's. Time will tell....

JL: What's left to be done with ext4 and the supporting utilities?

From ext3 to ext4 An Interview with Theodore Ts'o

Jeffrey B. Layton

TT: The big thing that's left to be done is the online resize and 48-bit block number support in e2fsprogs.

JL: **What options do you recommend to get the best performance from ext3 and ext4? (understanding that there are some articles around the web that discuss performance tuning).**

TT: The big one that I always recommend is the noatime mount option. It disables POSIX-required functionality, but it makes a huge difference on many workloads, especially desktop workloads.

If you compare the number of megabytes written for the “make” and “make clean” steps with and without noatime, you'll find that “make” requires 10% less disk writes (with all of the attendant seeks to the inode table) and “make clean” requires 50% less disk writes.

Other than that, it really depends on the workload. We try to make the defaults work well for most users. Speaking generally, ext4 will perform better if you create a fresh ext4 file system image compared to converting an existing ext3 file system. (On the other hand, if you have a very large pre-existing ext3 file system, you may not have the space or can afford the downtime to do a backup/reformat/restore operation.)

If you don't need the reliability guarantees of what happens on a crash, you can run without a journal, and disable barriers (ext4 enables barriers by default, for safety; for historical reasons, ext3 does not enable barriers by default) via the mount option “barriers=0”. If you don't need the security guarantees of what happens after a crash that are provided by “data=ordered”, try using the “data=writeback” mount option. “data=ordered” prevents files which were created right before a crash, from containing blocks that contain uninitialized data, which might reveal private information from another's mail or p0rn directory, for example. This is much more important on timesharing systems than it is on single-user systems. “data=ordered” also has some implied data safety issues for badly written application which don't bother to call fsync() that has been the subject of recent controversy

Most of the ways to get best performance out of the file system isn't in the tuning, but rather in making minor changes to your application programs.

If you are using ext4, and you are writing a large file, particularly in a random order (for example, as a bittorrent client might do, or an HPC program which is filling in a results file in random order), preallocate the output file to expected final size, using fallocate() or posix_fallocate(). Using fallocate() is also good idea if the file will take a long time to write out — for example, if you are writing out a large video file in real-time, as you might in a DVR, and you know that you are recording a one hour show at a compression/quality rate that requires 1GB/hour, then fallocate()ing the 1GB in advance will allow the file to be allocated contiguously on disk.

[n.b. The fallocate() system call is not in most glibc's as of this writing, but posix_fallocate() is; the problem with posix_fallocate is that if you use it on ext3, it will attempt to emulate fallocate() by writing all zeros to the file. This emulation step can be very slow, and may come as a surprise to the application that was expecting posix_fallocate() to be quick; the fallocate() system call has the advantage that if it is not present, it will fail, and the application can then decide on its own what it wants to do.]

For both ext3 and ext4, if you are using readdir() and then accessing all of the files in a directory, is a very good idea to sort the files returned by readdir() in inode order. For why, see [here](#) and [here](#).

From ext3 to ext4 An Interview with Theodore Ts'o

Jeffrey B. Layton

(Ext4 has a inode table readahead performance algorithm that helps avoid this problem somewhat, but it's still a good idea to do sort-after-readdir.)

For both ext3 and ext4, try to avoid small writes; large writes which are block aligned will always be faster. If the application must do many small writes, it may be worthwhile to use `mmap()`; however, if the application is only going to be making a single sequential read or write pass over the file, `mmap()` is unlikely to be helpful.

Hope this helps!