EXT3, Journaling Filesystem

The ext3 filesystem is a journaling extension to the standard ext2 filesystem on Linux. Journaling results in massively reduced time spent recovering a filesystem after a crash, and is therefore in high demand in environments where high availability is important, not only to improve recovery times on single machines but also to allow a crashed machine's filesystem to be recovered on another machine when we have a cluster of nodes with a shared disk.

This talk will describe the ext3 filesystem, both its design goals and its implementation. It will explain some of the challenges involved in adding journaling in a way which is completely compatible with existing ext2 filesystems (it is possible to migrate existing ext2 filesystems to ext3 and back again), and will cover the architecture of the implementation, which involves a completely new, generic block device journaling layer in the kernel.

1. Notes

1.1. Original presentation

The original presentation of this talk occurred in room A of the Ottawa Linux Symposium, Ottawa Congress Centre, Ottawa, Ontario, Canada on the 20th of July, 2000 at 13:45 local time. This presentation was given by Dr. Stephen Tweedie.

1.2. Presenter bio

Stephen has been involved with the development of the Linux kernel since its early days. His work has been primarily on the filesystem and virtual memory code, with miscellaneous contributions all over the kernel. However he never goes near the network code. His recent and current projects include several high-end features such as raw I/O, fast zero-copy filesystem I/O and high availability.

Working for DEC for two years, Stephen worked on VMS kernel internals for high-availability clustered filesystems. He is now employed full-time by Red Hat, which lets him work on Linux exclusively.

1.3. Presentation recording details

This transcript was created using the OLS-supplied recording of the original live presentation. This recording is available from ftp://ftp.linuxsymposium.org/ols2000/2000-07-20_15-05-22_A_64.mp3

The recording has a 64 kb/s bitrate, 32KHz sample rate, mono audio (due to the style of single microphone recording used) and has a file size of 35657984 bytes. The MD5 sum of this file is: d1aac5c2d7d24123245b3a45956eeb1e

1.4. Creation of this transcript

1.4.1. Request for corrections

This transcript was not created by a professional transcriptionist; it was created by someone with technical skills and an interest in the presented content. There may be errors found within this transcript; we ask that you report them to using the bug tracking interface described at http://olstrans.sourceforge.net/bugs.php3

1.4.2. Tools used in transcript creation

This transcription was made from the MP3 recording of the original presentation, using XMMS for playback and lyx (with docbook template) for the transcription.

1.4.3. Format of transcript files

The transcribed data should be available in a number of formats so as to provide more ready access to this data to a larger audience. The transcripts will be available in at least HTML, SGML and plain ASCII text formats; other formats may be provided.

1.4.4. Names of people involved with this transcription

This transcription was created by Jacob Moorman of the Marble Horse Free Software Group (whose pages live at http://www.marblehorse.org). He may be reached at roguemtl@marblehorse.org

The primary quality assurance for this document was performed by Stephanie Donovan. She may be reached at sdonovan@achilles.net

1.4.5. Notes related to the use of this document

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. While quality assurance checks on this transcript were performed, it was not created nor checked by a professional transcriptionist; the technical accuracy of this transcript is neither guaranteed nor confirmed. Please refer to the original audio recording of this talk in the event confirmation of the speaker's actual statements are needed.

1.4.6. Ownership of the content within this transcript

These transcripts likely contain content owned, under copyright, by the original presentation speaker; please contact them for licensing requests, but do so in a polite manner, please. It may also be useful to contact the coordinator for the Ottawa Linux Symposium, the original venue for this presentation. All trademarks are property of their respective owners.

1.5. Markup used in this transcript

1.5.1. Time markers

At the end of each paragraph within the body of this transcript, a time offset is listed, corresponding to that point in the MP3 recording of the presentation. This time marker is emphasized (in document formats in which emphasis is supported) and is placed within brackets at the very end of each paragraph. For example, *[05m, 30s]* states that this paragraph ends at the five-minute, thirty-second mark in the MP3 recording.

1.5.2. Questions and comments from the audience

These recordings were created using a bud microphone attached to the speaker during their presentation. Due to the inherent range limitations of this type of microphone, some of the comments and questions from the audience are unintelligible. In cases where the speaker repeats the audience question, the question shall be omitted and a marker will be left in its place. Events which happen in the audience shall be bracketed, such as: [The audience applauds.]

Further, in cases where the audience comments or questions are not repeated by the speaker, they shall be included within this transcript and shall be enclosed within double quotes to delineate that the statements come from the audience, not from the speaker.

1.5.3. Editorial notes

The editor of this transcript, the transcriptionist (if you will), and the quality assurance resource who have examined this transcript may each include editorial notes within this transcript. These shall be placed within brackets and shall begin with ’ED:’. For example: [ED: The author is referring to sliced cheese, not grated cheese.]

1.5.4. Paragraph breaks

The paragraph breaks within this transcript are very much arbitrary; in many cases they represent pauses or breaks in the speech of the speaker. In other cases, they have been inserted to allow for enhanced clarity in the reading of this transcript.

1.5.5. Speech corrections by the speaker

During the course of the talk, the speaker may correct himself or herself. In these cases, the corrected speech will be placed in parenthesis. The reader of this transcript may usually ignore the parenthised sections as they represent corrected speech. For example: My aunt once had (a dog named Spot, sorry) a cat named Cleopatra.

1.5.6. Unintelligible speech

In sections where the speech of the author or audience has been deemed useful, but unintelligible by the transcriptionist or by the quality assurance resource, a marker will be inserted in their places, [unintelligible]. Several attempts will be made to correct words and phrases of this nature. In cases where the unintelligible words or phrases are clearly not of importance to the meaning and understanding of the sentence, they may be omitted without marker insertion.

2. Transcript

We had an overview this morning about all the different filesystems in Linux and roughly what different options there are about these days. I'm going to just be talking purely about the EXT3 filesystem in general. I'll also talk a little bit about some of the specific things that we've been working on in the virtual memory system that particularly affect journaling filesystems and the work that we've been doing. For example, things like allowing filesystems to reserve memory in clean, deadlock-safe ways. But really the bulk of the talk will be about this EXT3 filesystem and also about some of the (really about the) implementation details; how the internals of the filesystem abstract out the journaling from the actual filesystem operations. *[00m, 54s]*

And why have we done all of this EXT3 stuff anyway? Well, really there are several motivations. First of all, people still like EXT2; they still trust EXT2. EXT2 is a fairly well proven filesystem. It doesn't have all the bells and whistles of some of the newer filesystems. It doesn't have the small file efficiency of reiserfs. It doesn't have the directory scalability of XFS, but it is a proven workhorse for Linux. And most importantly, there are a rather lot of users out there who have got existing EXT2 filesystems. And more every day. *[01m, 40s]*

And some of these EXT2 filesystems are getting really rather big. Even 24 months ago, there were people building 500 gigabyte EXT2 filesystems. They take a long time to fsck. I mean, really. These are filesystems that can take three or four hours just to mkfs. Doing a consistency check on them is a serious down time. So the real objective in EXT3 was this simple thing: availability. When something goes down in EXT3, we don't want to have to go through a fsck. We want to be able to reboot the machine instantly and have everything nice and consistent. *[02m, 23s]*

And that's all it does. It's a minimal extension to the existing EXT2 filesystem to add journaling. And it's really important, EXT2 is the workhorse filesystem. It's the standard stable filesystem. We don't want to turn EXT2 into an experimental filesystem. For one thing, users expect to have EXT2 there as a demonstration of how to code filesystems for Linux. It's a small, easily understood filesystem which demonstrates how to do all of the talking to the page cache, which has changed in 2.4, all of the locking in the directory handling, which has changed in 2.4. All of these changes in the VFS interface and the VM interface that filesystems have to deal with are showcased in EXT2. So there are multiple reasons why we really do not want to start making EXT2 into an experimental filesystem, adding all sorts of new destabilizing features. And so the real goal for EXT3 was to provide the minimal changes necessary to provide a complete journaling solution. *[03m, 26s]*

So it provides scaling of the disk filesystem size and it allows you to make larger and larger filesystems without the fsck penalty, but it's not designed to add things like very, very fast, scalable directories (to Linux) to EXT2. It's not designed to provide extremely efficient support for extremely large files and extent-mapped files and this type of thing that people have been talking about for EXT2. But really the goal is to provide that one piece of functionality alone. *[03m, 55s]*

And one of the most important goals for the whole project was to provide absolute, total, complete backwards and forwards compatibility between EXT2 and EXT3. You can take an existing EXT2 filesystem, throw a journal file onto it, and mount it as EXT3. There, you have a journalled filesystem. This laptop, I installed Red Hat 6.2 on it, it formats all of these partitions as EXT2. I've added a couple of journal files and now it's all running EXT3. *[04m, 28s]*

Better yet, if you unmount an EXT3 filesystem, then it marks that filesystem as having been cleanly unmounted. The journal doesn't need to be recovered after a clean remount, so all these extra data structures you get for journaling, for dealing with reboots, are not relevant if the filesystem has been unmounted cleanly. If you do a clean unmount of an EXT3 filesystem, you can then mount it again as EXT2 and EXT2 just doesn't care. It's completely compatible in both directions. *[05m, 02s]*

If you have an EXT3 filesystem that after a crash, for example, and there's a journal on there which is active and needs to be replayed onto the filesystem, then in that particular case you need to have recovery. You cannot mount it as EXT2, because to do so, you would get an inconsistent filesystem. You would corrupt all of the information that's in the journal. So EXT2 has for some time had a set of compatibility bits in the superblock which let you say, for example: what features are in the filesystem? So that versions of the kernel which don't understand a particular feature will not try to mount a filesystem that it can't understand. So if you've got an active EXT3 filesystem, the flag set in that superblock which says: don't you dare mount this as EXT2; it's not compatible. And EXT2 will cleanly, will be quite careful; I don't understand that bit, I'll refuse to mount it. But as soon as you've unmounted it, that bit gets cleared. *[05m, 56s]*

Question?

"You answered it."

Okay. And similarly the fsprogs, e2fsck can use those bits to work out exactly whether or not this perfectly normal-looking EXT2 filesystem actually has an EXT3 journal and whether or not that journal needs recovery. *[06m, 14s]*

So we have complete compatibility, both in terms of the on-disk format, so that an unmounted EXT3 filesystem does look exactly like EXT2, but also in terms of the functionality. So all of the existing things that we have in EXT2... things like the persistent inode attributes, you can set on an inode. For example in EXT2, you can set a directory inode to be synchronous, so that all of the updates on that filesystem have synchronous metadata updates. *[06m, 41s]*

And you can set that attribute on a mail spool to get DSD synchronous update compatible behavior for your mail files. Things like sendmail can make assumptions about consistency after a reboot. And if you need that kind of consistency, you can do that in EXT2. All those attributes are all exactly there in EXT3, because the EXT3 source code started off by me taking a copy of the entire EXT2 directory and copying it into the directory called ext3 and then doing a global search-and-replace for all occurrences of ext2 and replacing that string with ext3. *[07m, 13s]*

It's exactly the same source base it started off from. The only reason it was made into a separate source tree was so that I could have test boxes that run EXT3 development code on my test partition without having to run that same development code on my root filesystem, which kind of made me a little bit nervous. I didn't want to do that. *[07m, 31s]*

So we have EXT3 as a separate filesystem simply for that reason, to isolate the new code from the old, stable code. But apart from that, I mean it's exactly the same source base that's been used for the two. There's no loss of existing functionality. And in particular, the guarantee about journaling consistency covers all of the existing functionality. *[07m, 51s]*

So even things like quotas are guaranteed to be consistent after a reboot with journaling. So if you update a file, write to or extend a file, or truncate a file, the quota operations that go on along side that operation are guaranteed to be consistent with the contents of the quota file. You never have to run a quota check after a reboot on EXT3, just as you never have to run a fsck. *[08m, 16s]*

There are another couple of subtle issues around recovery which are really not at all handled in any way whatsoever by existing filesystems. The main one of these is orphaned files. Orphaned files is this concept that the GFS people were talking about earlier as being a nasty case. That you can have files which are deleted from the filesystem, but which are still open by processes. And the semantics are that when that process dies, you want to close and you want to delete the file, remove it from disk, return all the space to the filesystem. *[08m, 53s]*

Well, that's all well and good. The trouble is that, that filesystem state that has the file (closed, sorry) deleted but still open – that's a completely consistent state for the filesystem. So in the first versions of EXT3, if you have a filesystem in that state, it's perfectly consistent. The unlink of that file is a single transaction on the disk. That transaction did not reclaim the disk space because it didn't need to, the file's still open. So we've got this completely consistent on-disk state in which there is a file which exists on the disk but isn't in the directory structure anywhere. *[09m, 27s]*

Now obviously after a reboot, you can be pretty sure that the reboot also kills a process that had that file open and so we need to preserve the semantics that killing the process deletes the file. So we have to have some way of dealing with these orphaned files and that's new functionality that we just don't have to deal with in EXT2, because EXT2 assumes that there's always a filesystem check there to clean up these things after a reboot. So there are a few things like that in EXT3 that we need to deal with after a reboot to make sure that consistency is complete. And so, really, this is the goal: absolute consistency of the filesystem in every respect after a reboot, with no loss of existing functionality. *[10m, 08s]*

So how's it actually implemented internally? What does this source look like? Well you see the first thing we did is just take the EXT2 filesystem and turn it into EXT3. That's not the only new filesystem subdirectory that the EXT3 patches provide. EXT3 also provides a new (subdirectory and) filesystem directory under the Linux source tree called JFS. *[10m, 35s]*

That's the journaling layer for EXT3; and that's entirely independent of the EXT3 filesystem itself. It's a completely abstract journaling layer which allows you to make arbitrary block device modifications in the buffer cache and have those obey transactional semantics. So that you can make arbitrary transactions on arbitrary block devices and filesystem transactions are just one example of things you can do with that. *[10m, 59s]*

It was explicitly written so that, for example, if you had a logical volume manager which wants to make a number of changes over a number of different block devices and make those changes completely atomic, with respect to reboot. So that, for example, you are adding a whole pile of logical volumes or you're deleting a pile of logical volumes and you're updating it across multiple block devices for a stripe set. *[11m, 21s]*

The JFS layer that we added for this is perfectly capable of being used for something like that as well; it's not restricted to the EXT3 filesystem. And in particular the EXT3 filesystem does not know anything about journaling. Journaling is separate. EXT3 doesn't have the journaling; all it knows is transactions. It says: here is the beginning of a set of block device modifications; I'm going to modify this block device which contains my filesystem and I'm going to tell you that these five block device updates form a single transaction. And it tells that to the journaling layer and the journaling layer is responsible for making sure those five updates either all appear in the block device after a reboot, or none of them appear. And the journaling is done inside that. EXT3 does not know about journaling. *[12m, 13s]*

The only places where EXT3 has to interact with the journaling layer are to tell it where these transactions start and stop and which updates belong to which transaction. And also, to manage the disk space which is used by the inode which contains the journal. And the journal in the JFS layer can be on any inode in any filesystem or it can be on an arbitrary sub-range, set of contiguous blocks on any block device. It doesn't even have to be on the same device that your filesystem is on. And you can have multiple filesystems sharing the same journal, if you want. The JFS layer will cope with all of that. *[12m, 49s]*

The only other thing... so the only things we have to do is manage the transactions and manage the journal inode. And one of the things involved in that, is that the filesystem is responsible for asking the journal layer to do journal recovery after an unclean reboot. And as I said earlier, once you've done all that, once you've unmounted the thing, the EXT3 filesystem tells the journaling to close down, clean up the journal and mark everything as consistent. And once it's done that and the journaling layer has said: find all of your updates are consistent, and this, the journal has been emptied out, The filesystem can then just set a flag in the filesystem that says: okay, we no longer need to recover anything; don't worry about the journal and just feel free to mount it as EXT2. So that's basically what we have in the design. It's two completely different sets of code; one that's the abstract journaling layer and one, a simple set of modifications to EXT3 to add transactions. *[13m, 48s]*

So if we look at this layer that's adding journaling, what does it look like? What does it provide? Well what it does is, it exports an API which allows you add transactions onto any block device. So just as EXT3 doesn't understand the first thing about journaling, it doesn't need to, the JFS layer doesn't understand the first thing about filesystems, it doesn't need to. All of the filesystem properties are dealt with in EXT3. *[14m, 16s]*

All of the journaling is done within the JFS layer. It provides an abstract concept of a log; it allows you to register a journal with the JFS layer. And when you register that journal, you've got two choices, you can either say: here's some inode on some filesystem; please use the contents of this inode as a journal. That must be a block device filesystem because the JFS layer assumes that it can always do a mapping between an arbitrary block inside that inode and the block on disk. *[14m, 50s]*

So it says: create one of these journals, either in an inode on a block device. It doesn't care which you use and you don't even have to have the journal on the same device that you're going to be doing the updates on. So you can have a filesystem over here and journal it to a separate spool disk over there, if you want to. The JFS layer will be quite happy with that. In fact, you can even have more than one block device journaling to the same disk, if you want. The JFS layer is quite happy about that. *[15m, 20s]*

It provides write ordering guarantees. All the way through the I/O layers in the kernel. So it makes all these guarantees that, if you have a transaction which is in progress, but has not yet been committed, the JFS layer provides a guarantee that not one update of that particular transaction will hit the main disk until you've done the commit. It provides all that guarantee, but it doesn't necessarily tell you when the updates will hit the disk. The JFS' updates are still write-behind. *[15m, 57s]*

So in other words, you're not doing transactions synchronously; you're not saying: well I'll do all of these updates and that results in the creation of a new directory on the disk. And then all of the updates I've made to the disk are committed into the journal and I return. *[16m, 12s]*

The JFS layer maintains, in cache, a list of all of the updates which form any particular transaction and it will do, in its own sweet time, the normal write-behind that the buffer layer is already doing on this. It'll just write behind in some future time; we'll make sure all of those future updates hit the disk. But it will make guarantees about the ordering, so that when they do hit disk, the transaction is either all there, or not there at all. *[16m, 37s]*

Now journaling in many cases is very similar to databases. *[16m, 45s]*

[The speaker calls on an audience member for a question.]

"Does the journaling layer provide an ordering guarantee between transactions? Can transaction 47 be... when you recover, can transaction 47 exist in the recovered state when transaction 46 does not?" *[17m, 08s]*

The question is: is there an ordering guarantees in the JFS layer? Yes and no. This is actually a reasonably complicated design issue. And it's something which in particular makes a huge amount of difference whether you're running on a single node filesystem or a shared disk. It's one of the reasons that GFS implements things very differently, internally, to EXT3. The JFS layer in EXT3... its API does not make any guarantees about transaction ordering. If you have a transaction which updates blocks one, two and three on the disk and another one which updates blocks five, six and seven, the API doesn't give you any ordering guarantee between those. *[18m, 00s]*

Now on something like GFS, if it's doing journaling, that's really important that there's no ordering guarantee because in GFS, you've got to be able to release a disk block back to disk as quickly as possible in order to relinquish a lock, which is required by some other node. On this local disk filesystem, you don't have that. So on a local disk filesystem, it's quite legitimate to batch all these updates off into very large transactions and just send them all out at once. That works really efficiently. *[18:36]*

The only place it breaks down on a local disk is if you've got synchronous updates. If you do, for example, an fsync() on a file, or if you open a file as O_SYNC, and in that case if you have absolute write ordering guarantees, then in order to flush this one little file out to disk, say it's a mail spool, and you're doing a fsync() for some file that's just arrived off the network; in order to sync that to disk, if you've got write ordering guarantees, then you've got to sync all previous transactions and commit them to disk as well. And that's expensive, because that hurts the latency of those syncs. It does not change your bandwidth; it's actually more efficient in terms of disk bandwidth and throughput to batch your transactions off in large chunks. *[19m, 18s]*

So the API that JFS exports does not make any guarantees about write ordering consistency. Internally, it batches (all of the transactions) all of the updates which are made by the filesystem... it just batches them up sequentially into big, compound transactions and puts them out to disk as a single unit. So the implementation makes... does actually happen to preserve write ordering in all cases. *[19m, 45s]*

That is not guaranteed by the API, and one of the things that we need to do once the core is out there and being used... once everything is up and running, including all of the performance stuff that's still a work in progress, is to do profiling to find out whether or not there are applications which really do need to have fine-grained committing of transactions, so that when you fsync(), you just... you know you've got transactions three, four, five, six and seven, and transaction six suddenly becomes related to an fsync(), so you have to commit that one. Does it actually make a difference to performance to be able to commit transaction six without having to commit transactions three to five? If that turns out to be the case, then I will be able to do that in the future without changing the API. The API doesn't make that guarantee; it's just an optimization internally. *[20m, 36s]*

But apart from that whole, the right ordering guarantees are made and how it does this is fairly simple. Journaling filesystems are, in most cases, very closely related to database journaling. But with some very, very special cases. In databases, typically a locking database is characterized by the steps it has to go through to recover the state of the database after a crash. And it's characterized by whether you have to do undo's or redo's. *[21m, 19s]*

So in the case of an undo logging, that means that what you do is that you put into the journal all of the old state of the buffers that you're modifying and then you can write the new state to the disk. And if you crash before the transaction is committed, then you can undo the modifications you made on disk by copying the old contents from the log. So that the log contains the information necessary to undo incomplete transactions. *[21m, 46s]*

Or you can do redo logging, which is that the new data is written into the log and you leave the original data on the disk, in its main location. And then after a crash, any transactions which are incomplete, the original copy is still on disk, so you don't have to do anything. And it's only complete transactions which exist in the log and you have to replay those into the main filesystem. *[22m, 08s]*

In almost all cases, journaling filesystems just use simple case of doing redo logging. So basically every single modification that's made to the filesystem will be written to the log first. And only once it's committed to the log, not just written in the log, but committed to the log, are we allowed to update the main copy on disk. And that's (what EXT2 does) what EXT3 does, so all of those write ordering guarantees are provided by the JFS layer. *[22m, 36s]*

So that JFS layer controls these various, different things. It controls transaction commit and the commit of a transaction involves writing all of the things which that transaction modified to the journal, and then writing a commit record. It's not sufficient just to write the thing to the journal, because there's got to be some mark in the journal which says: well, (has this journal record actually) does this journal record actually represent a complete consistency to the disk? And the way you do that is by having some atomic operation which marks that transaction as being complete on disk. *[23m, 14s]*

Now, disks these days actually make these guarantees. If you start a write operation to a disk, then even if the power fails in the middle of that sector write, the disk has enough power available, and it can actually steal power from the rotational energy of the spindle; it has enough power to complete the write of the sector that's being written right now. In all cases, the disks make that guarantee. *[23m, 41s]*

So the fundamental thing about transactions are: at the end of writing the new contents of the transaction to the log, we write a single 512-byte sector to the disk, which contains whatever magic numbers to identify as it as a particular type of block; so say this is a commit block and it will contain a sequence number that matches all of the transactions that have gone previously, so that it doesn't get confused between that, what you've been writing there, and the old contents of the log, previously. *[24m, 14s]*

That single write of that one sector on disk marks the entire transaction as being complete. And so all of that write ordering, the write ordering within the journal, that says that the journal has to be written and complete on disk before you have to write the commit record... that's all handled by JFS. The write ordering that says: you have to write the commit record before you write any of the blocks back to the main filesystem. That's all handled by the JFS. *[24m, 38s]*

Obviously there's only a limited amount of space in that log. So the old transaction checkpointing... Checkpointing is the process of flushing all the contents of the log out to the main disk. That's handled by the JFS layer and that's actually really important, because if you think about it, the transaction is committed on disk as soon as we've written that commit record in the log. But once that commit record has been written to the log, the only copy of the data that the transaction has just written is in the log; the main copy on disk still has the old version, so we cannot throw away that data in the log until we have written it, copied it back, onto disk. And that's called checkpointing. *[25m, 19s]*

That's what allows us to re-use bits of the log. We take the contents of the log, make sure the copies on disk are all up-to-date, and at that point, we can trim the tail of the log. All of that is handled by the JFS layer. All of the write-behind is handled by the JFS layer. It has its own set of timeouts and links itself into the buffer cache write-behind layers and so all of that is handled completely transparently to the filesystem. And also, the JFS layer, for performance reasons, tries to make everything go as asynchronously as possible. It never stalls things; it never tries to do more copies than it needs to. *[25m, 55s]*

So, for example, when we're doing journaling, the filesystem may say: take some buffer that's in memory and I'm going to write this to disk for that application there and journal it. And the journaling code has to make sure that block gets written to the journal first, and then, after the commit, it goes to its main location on disk. And the journaling layer will do zero-copy from that; it will actually create a new I/O request that points to the old disk buffer location and use that to journal the data to the journal file, without copying the data block. Now all that kind of thing is handled by the JFS layer. *[26m, 33s]*

[The speaker calls on an audience member for a question.]

The question is: does it make sense under extreme memory pressure to throw away data that's been written to the journal, but hasn't yet been written to disk. Absolutely not, because if it's written into the journal and you want to throw it away, it's much, much more efficient to just write it to disk than doing anything else. Actually on the next slides, I'll talk about reservations, which are the way that you limit the amount of outstanding data that's being used by the transaction layer at any point in time. And that reservation system is sufficient to reduce the memory pressure. *[27m, 21s]*

[The speaker calls on an audience member for a question.]

So the two questions: first of all, the sector size that the disks guarantee to be atomic is smaller than (the sector size) the block size the filesystem uses. Yes, but for the commit block, I only ever care about the first 512 bytes of that block. If the commit block's first 512 bytes are up-to-date on disk, then that's assumed to mean (the sector, sorry) the entire transaction is committed. And you have to be very, very careful not to have critical commit information that spans a 512-byte sector boundary. As for write ordering, absolutely. We need to make sure that even when we submit multiple async I/O requests to the disk, the disk doesn't allow us the write to reorder things in such a way as that the commit block hits the disk before the other transaction blocks. And I will come back to that point, because it's actually a really nasty point for performance. (I'll come back to that, yes.) *[28m, 36s]*

One other thing that I'll say about this in terms of asynchronous behavior is that the journaling layer is really careful... (Ah, come back.) The journaling layer is really careful to make sure that things don't stall unnecessarily. So that means that when we start committing a transaction, (we don't stop) we don't stall the filesystem itself. And committing a transaction means that we take, okay, I'm going to take a snapshot of the entire filesystem state at this point in time and I'll start committing that state to disk. But the filesystem is still allowed to make new copies of the data. The filesystem is still allowed to modify the virtual block device in the buffer cache. While it does that, we have to keep the old contents of that snapshot present in memory, so that we can commit it to disk. *[29m, 41s]*

And so in that particular case, the JFS layer provides a copy-on-write mechanism so that if a new filesystem request comes along, that wants to modify a block that we are in the process of committing but haven't finished committing, then we make a copy of that before the filesystem is allowed to modify the buffer. And that means that the disk I/O for committing a previous transaction can go on in parallel with the filesystem operations for the new transaction. That is one of the things we do in the JFS layer to make sure that the concurrency of the system is as high as possible. There are no synchronous transactions in the JFS layer, at all. The only way you can get something synchronous is if you say: well I actually want the application to wait until this thing is on disk because the application has done something like an fsync(). *[30m, 28s]*

So JFS provides all this functionality and it provides it to the user, where in this case the user is something like the LVM layer or in this case, the EXT3 layer. And it has a nice abstract API for exporting this functionality. Everything is expressed in terms, not of transactions, but of handles and to make this distinction clear, a handle represents one single operation that marks a consistent set of updates on the disk. *[30m, 59s]*

So a handle might be something like a create and the create has to go through a directory, add a directory entry to that directory, modify the timestamp on that directory, modify the size of the directory; it has to allocate a new inode, and it then has to modify the inode table for that; it has to modify the superblock to change the number of inodes in that group in the superblock; and it has to mark the inode bitmap as being changed. And all of these operations for a single, consistent operation update in the filesystem are done with a single handle. *[31m, 37s]*

But a handle is not necessarily the same thing as the journaling transaction on disk, because the journaling layer will allow multiple updates like this to be batched into a single transaction. So, to make this distinction quite clear, the transaction on the disk is not necessarily the same as (the handles) the updates the filesystem is doing. And that means that because we are doing write-behind, we may be making only one filesystem commit every five seconds, and you can have hundreds and hundreds of filesystem operations proceeding in that time scale. So everything can be batched up very efficiently using these handles. *[32m, 16s]*

Now the API has a journal start and a journal stop pair. A journal start gives you a handle. A journal stop tells the system that handle is finished with. It provides nested transactions, so if you do a journal start and then another journal start, that all gets batched into the same transaction. And the handle is not marked as complete until you've gone through two journal stops in that case. *[32m, 42s]*

When you do that journal start, you have to tell the JFS layer how many blocks you expect might be modified by this update. That's really, really important. This is absolutely critical, to avoid deadlocking in the journal. The JFS layer has to know (before it starts committing your transaction, sorry) before it starts processing your transaction that there is enough space left in the journal to write out all of the blocks which might become part of your transaction. *[33m, 14s]*

And if it turns out there's not enough log space left, well it might be that your transaction includes blocks from previous transactions in the journal. And because these blocks are now being pinned as part of a new transaction, we can't flush them to disk. And because we can't flush them to disk, we can't checkpoint the log to remove one of the old transactions from the log. And because we've run out of space, we're just deadlocked completely. *[33m, 36s]*

So we have to make these kind of reservations to make sure the transaction does not start until all of the space that it might use is guaranteed present in the log. So the journal start/journal stop provide boundaries to make that sort of reservation guarantees. *[33m, 50s]*

There's a question at the back.

[The speaker calls on an audience member for a question.]

The question is: what happens if there's not enough space physically in the journal for the transaction? Transactions are very limited in size; they're never more than a few dozen blocks. The only two cases where a transaction can grow without bounds are for write system calls, because writes can actually... an application can quite legitimately write a half gigabyte of memory to a file in one syscall. *[34m, 22s]*

That's okay because EXT3 does not guarantee that write is atomic. That'll be broken up into multiple, smaller transactions. And the only other case is truncate, because you might have a ten gigabyte file that you're deleting and you really want that delete to be an atomic operation, but that delete can touch arbitrary amounts of disk space. Potentially you have one separate bitmap block; in the most fragmented case in the filesystem, you'd have a separate bitmap block being updated on the disk for every single block that you free from that file. So truncates are special case which I'll come to later, because we have to deal with that in a nasty manner. *[35m, 00s]*

Actually, ask any of the journaling filesystem people in the audience: what's the hideous part of the entire system. And it's deleting files. Everything to do with delete is hairy. Everything to do with delete... you have nightmares around what happens if blocks get deleted and then reallocated. *[35m, 22s]*

What happens if they get reallocated with a different type of data? What happens if you reallocate them and then take a crash and have to undo the reallocation and undo the delete operation? And they really get nasty. *[35m, 33s]*

So most of the problems in EXT3, most of the hairy parts of the design, as in most journaling filesystems, in fact, come from deletes. All of the block updates that the filesystem makes after this time go through a pair of (system calls, sorry) journaling calls exported by the JFS. You can ask the journal to give you write access to a block and remember I said earlier that to ensure efficiency in the system, we do copy-on-write. If there's a block (being journalled, sorry) being scheduled for commit, then we don't mind the filesystem continuing to modify that block as long as we have a chance to copy it out and make sure that the old snapshot that we're committing is still consistent. *[36m, 14s]*

And to achieve that copy-on-write, we have to know what the filesystem is going to modify before it modifies it. So you have to go through a process of getting permission to write a block, just to make sure that copy-on-write can happen. And then at the end of it, we can say: okay, that block has now been finished, it's now been dirtied; it can now be written to a journal or whatever. *[36m, 33s]*

This API provides the reservation, it provides... the handles you get back from this journaling, you can mark individual handles as being synchronous before the journal stop. And when the final journal stop happens and that transaction (is committed to the disk, sorry) is bundled up for the current transaction on disk, it will immediately submit that commit to the disk, and will synchronously wait for that commit to finish on disk before returning. So you can get synchronous operation on a per-handle basis. *[37m, 03s]*

And it also provides various... the JFS layer provides various kind of management functions for creating a journal, doing recovery of a journal, marking a journal as being complete, flushing a journal out to disk, things like that. You need that inside EXT3. For example, if an EXT3 filesystem that's read-write gets mounted read-only. If you remount it read-only, you want to make sure that, beyond that point, no more updates happen to the filesystem. You want to make sure that nothing else happens in the journal. You also want to make sure that there's no recovery necessary on that filesystem. If you take a reboot while the filesystem is marked read-only. So all of the consistency functions necessary to flush everything out and stop further writes when you get one of these remounts. All of these kinds of functions are exported by the JFS layer. *[37m, 52s]*

The important thing about this API is that all of the updates that we're making are expressed in terms of: here is a block, here is the new contents of that buffer on disk. We're doing physical journaling. Now there's some filesystems that do journaling which do logical journaling and that means that, for example, if they allocate a particular disk block to a particular file, they'll write an entry in the journal that says: this disk block (is marked free, sorry) is marked in use. And this file has a mapping pointer pointing to that disk block. And that description for that allocation might only be a few bytes long. EXT3 will journal the entire copy of the blocks that have been modified, all 1K, 4K, whatever, of the blocks. *[38m, 43s]*

But it doesn't do any copy when it's doing that, so the CPU overhead is minimal and in particular, if you have got a lot of operations that touch the same blocks, that block only gets journalled once. And that means that this is a very, very simple mechanism which gives you completely free compression of multiple events. So, for example, you almost never get a disk block allocated in a single commit operation. If the commits are going every five seconds, then if you only had one block allocated within that five second boundary, then it's no big deal to write out 4K; to mark that block in use. But, as is more likely, you've got hundreds and hundreds of allocations going in rapid succession. *[39m, 24s]*

All of the bitmap blocks, all of the bits in the bitmaps which are being updated only result in one copy of the new block being sent to disk. So it's actually a fairly efficient way of compressing these kind of multiple operations... multiple directory operations or multiple allocations within a single block group; they can all be compressed fairly efficiently down to the disk. There are different types of buffer that you can pass through into the JFS layer. Now this is getting into the issue about: are we journaling every operation on the disk, or are we only journaling metadata? Right now, the current version of EXT3 that's in public release: EXT3 is (journaling all metadata, sorry) journaling metadata and data. So every file that you write is being written twice; it's being written once to the log and once to the main disk. The design goal is obviously not to do that. *[40m, 19s]*

So the design is that you can have metadata being journalled, but the data itself just gets written back to the disk any old way. Now if you do that, a whole pile of new ordering constraints come in. And guess what... they all have to do with delete. Except for one. *[40m, 43s]*

The only one that doesn't have to do with delete is: what happens if you allocate a whole pile of data, write it to disk, and then take a crash? Well, if you take a crash and then recover the state of that journal and replay all of the allocations to disk, but the data has not yet been written to disk, then the user can read those disk locks and get old, stale contents of what was previously there on the disk. And that might be an old copy of /etc/passwd or some other file you don't really want the user poking around in. *[41m, 11s]*

So for security, you really want to make sure that newly-allocated data blocks (will get committed, sorry) will get flushed to the disk before the transaction which allocates them is allowed to commit. So we have this concept of data blocks and that write ordering guarantee for data blocks is preserved by the JFS layer. *[41m, 30s]*

Then there are a whole pile of other things that can go wrong. Well, I can have a directory which is deleted; I can delete a directory and that is a perfectly legitimate operation in EXT2. And I can do that and I can commit that delete to disk and then I can have a new transaction which reallocates those same data blocks, which are now free, and puts them into a file. And I can commit that. And everything is now consistent. *[41m, 58s]*

Except what happens if I take a reboot? Well what happens is that we do a reboot and the log gets replayed and we go through the journal, replay all the metadata blocks that are in that log. Well, actually, this directory that we deleted a few transactions back, we made a new entry in that directory and there's a metadata block for that directory entry on the journal, but we don't journal data. So that the new contents of that data block, which are on the main disk, aren't in the journal. And so we take a crash and we replay and we overwrite that data block with the old contents of the directory block, because that's all that's in the journal. This is generally considered bad behavior. [The audience laughs.] *[42m, 40s]*

So we need to have ways of dealing with that, so we need to be able to have revoke records. There are various different ways of dealing with this; you can deal with it by making certain delete operations synchronous, you can do it by making sure that you don't reuse disk blocks until you've checkpointed that old record out of the journal. The way we're doing that in EXT3 is that deleting metadata can cause a revoke record to be written into the journal. And when you do the replay of the journal, the very first pass of the journal recovery, we look for all of the revoke records and make sure that any data that's been revoked is never, ever replayed. And so that deals with that particular case. *[43m, 20s]*

There are much worse things that could happen in journaling with deletes. For example, what happens if you have a piece of metadata that's in your directory and you've deleted that and you've reallocated that as data? And because you're reallocating it as data, well we have this property that we flush all the data to the disk before we commit the transaction. So we flush this data to the disk and then before we've committed this transaction, we take a power failure. And we have to reboot. And we do log recovery. And guess what? The operation which deleted that directory wasn't committed, so we've revoked that delete, so we've basically undeleted that directory. *[44:06]*

Unfortunately, actually we had thought we were going to reallocate that directory as data, we've already flushed the data to disk and overwritten the directory that we've actually just had to undelete. Whoops. So we have to make sure that we avoid that kind of thing. And there are, again, various ways of doing that. The way that this works in EXT3, is that the journaling layer provides the filesystem with the ability to record the last committed state of the bitmap block. And the filesystem can then use that to make sure that, when it is allocating data, it never, ever reuses the data block that has been freed in the bitmaps but that freeing hasn't yet been committed. *[44m, 49s]*

So there are all these little tricky things that happen around delete. And all of the write ordering constraints the JFS layer has enough infrastructure in there to support to the filesystem for getting this thing right. But the JFS layer doesn't understand anything about the difference between these different types of data, it just has ordering guarantees which the filesystem can use to make sure that metadata-only journaling does actually work right. That is actually all implemented, but the current EXT3 doesn't use it. It's not enabled for the simple reason that these write ordering constraints are really tricky to get right. And the priority has been to make sure that the core, rest of the EXT3 code is rock solid, before we start introducing all of these weird and wonderful new bits and pieces. *[45m, 28s]*

A few other things that we have in the JFS layer, there. It supports nested transactions. Now this is what we were saying earlier, that if you have journal start and you get another operation which does a journal start, well you can do that quite happily. The nested transactions allow you... This was really implemented for quota files. *[45m, 50s]*

Now if anyone's familiar with the Linux VFS there, they'll realize the quota system that EXT2 uses is not inside the EXT2 filesystem, it's actually a generic quota utility layer in the Linux VFS. And the Linux VFS (has) exports functions that the filesystems can use, so the filesystem can say: okay, I've allocated a disk block; please update this quota record accordingly. And the Linux VFS will say: okay, fine, there's enough quota left for you to do that; that's quite legal; please go ahead with that and I'll make sure that the appropriate quota record on some arbitrary file is updated. And as far as the VFS layer's concerned, the quota is just a regular file. And it makes write calls to the filesystem to update the quota file. *[46m, 38s]*

Now if you want to make quota updates consistent with the rest of the transactions, you actually need those write calls that VFS is making to be part of the same transaction that the allocation on disk was part of. You have to make sure they're consistent; you have to make sure that the quota file update is always part of the same transaction as the allocation that's (doing that) modifying it. And by using nested transactions, this comes as free. *[47m, 03s]*

The write system call starts the transaction. It calls the quota layer; the quota layer does another write. That new write for the quota file starts a transaction, but it gets a nested transaction inside the first one, modifies the quota file, completes the transaction. And only when the whole operation is finished do we mark that as being a complete transaction on the disk. So this quota file stuff, it just came for free. And it turns out that this is actually quite useful. There are applications out there that want to be able to use nested transactions in the filesystem. *[47m, 31s]*

And one in particular that's already using this is the InterMezzo filesystem that Ted talked about earlier. It's a distributed filesystem that has a local disk cache for caching files on the hard disk locally. And there's a coherency protocol that the various different nodes in this InterMezzo network use to communicate with each other to say: okay, I have an updated... I've got a newer copy of this particular file; I'll propagate it to all the other servers; I'm about to modify this copy of the file, so I'm going to invalidate the servers' copies; and all that type of thing. InterMezzo supports disconnected operation. *[48m, 07s]*

If I have populated my local disk cache with copies of the main server's in InterMezzo, then I can put that on my laptop and take it away, go to this conference, have all of my e-mail on that; have all of my other stuff on that; and then I get back home, I can plug it into the network and everything gets replayed. And the disconnected operation is really powerful. *[48m, 29s]*

But that means that the local disk cache... there has to be a way of recording what has changed in the local disk cache so that when we do reconnect to the rest of the network, we can replay what's changed. And InterMezzo actually uses EXT3 for this, to get very, very high performance in the local disk cache. Because it creates a nested transaction and say I'm running a disconnected operation and I'm making a new file on my filesystem. InterMezzo will create that new file in its disk cache and will update a journal file that InterMezzo maintains to say: this inode over here has been created in the InterMezzo filesystem with such-and-such a filename and it belongs to this particular place in the cache. *[49m, 13s]*

And InterMezzo really wants to make sure that the contents of its log, of its replay log, match what's actually on the filesystem cache. And it can use EXT3 for that; it can use a nested transaction. It will start a nested transaction, create the file, write an update record to its replay log stating that file has been created. And it gets normal write-behind so there's no synchronous updates on disk. It's fully asynchronous write-behind performance to that local disk cache but after a crash, it guarantees that that replay log is exactly consistent with the actual state of the disk cache. So InterMezzo is using these nested transactions; it's proven very, very useful for them. *[49m, 52s]*

Now we get to some of these other awkward little things. Orphaned files, as I said earlier, if we have a file which has been unlinked on disk, but is still open, then on the reboot, we need to make sure that file is deleted. EXT3 adds a new data structure on the disk. It has an entry in the superblock which points to a linked list of inodes on disk which need to be deleted on reboot. And whenever you unlink an open file, it gets added on to that list. And when you finally close that file, the delete operation which happens as a result of that close will remove the inode from that list. *[50m, 37s]*

I said earlier that truncate operations also can have unbounded transaction sizes. Well that's okay; if you get a truncate operation which exhausts the size of the journal, then that truncate will be split up into more than one transaction. But we still guarantee that that's atomic over a reboot, because if we have to split up one of these truncates over multiple transactions, we put it on exactly the same orphaned file list. And so in recovery, you can do all the cleanup. It basically means that when you do recovery, we look at the number of links, the number of hard links for that file in the inode list, and if the number of links is zero, we know it's a file that's just been deleted and we delete it. We finish doing the delete. if the number of links is greater than zero, we know it must have been in the middle of a truncate, and so we complete the truncate operation. So all of that kind of gets done at the EXT3 level; that cannot be done inside the journaling level. *[51m, 33s]*

VM reservations... journaling has this unfortunate property that if you have a transaction that's in progress, you cannot free the memory that transaction is occupying without first allowing that transaction to complete. Because unlike databases, most filesystems do not implement transaction aborts. It's just not something which is normally needed in a filesystem. And if you're not going to be able to abort a transaction, then you have to let the transactions run to completion. And if we're going to let the transactions run to completion, you have to have enough memory to do that. And if the VM system is saying: well I can't give you any more memory right now until you give me back some; then you can deadlock very, very rapidly. *[52m, 18s]*

So we need to have a way of doing VM, virtual memory, (transaction reservations, sorry) page reservations, so that the filesystem doesn't use more memory than the VM layer is able to give back independently. And it turns out that this is a relatively trivial thing to add to the EXT3 filesystem, because journaling filesystems already do the same kind of reservation in the log. So one of the things that the VM developers and the journaling filesystem developers have been talking about recently, is how to add an API to the virtual memory layer which allows the filesystems to tell the VM about the reservations it's making and to make sure that we never run into these deadlock situations. *[52m, 57s]*

There are also some tricky corners about write pressure. For example, if you've filled too much memory with dirty data, and it's proving impossible to clear stuff out of memory, because all of the pages in memory are dirty and need to be written to disk first, then we need to, at that point, stop making more dirty data. We also need to start cleaning the existing dirty pages back to disk. Now the filesystem has its own write ordering constraints. It cannot let the VM arbitrarily decide to write these things back to disk. *[53m, 33s]*

But the VM is the only part of the system that knows when this write pressure is getting excessive. So we have to have callbacks into the journaling filesystem which let the VM say: well, hey, you've got all these dirty pages, I want to start writing them back. But the filesystem has got to be able to say: well, actually these dirty pages are pinned in memory because of transactions, but I've got pages over here that I can free. So we want to have callbacks from the virtual memory system into the filesystem. Purely advisory callbacks, so that the VM can say: I have found all these dirty pages which I want you to get rid of, but if you can't do it, I don't care as long as you find something to get rid of. *[54m, 10s]*

And once you've got that kind of advisory callback, dumb naive filesystems like FAT or EXT2 can say to the VM: okay, you told me to write this page back and I'll do it. Where advanced filesystems that do journaling can use that as an indication that we need to do some write pressure and can choose the most appropriate pages to get rid of, because they know what the write ordering constraints are. So all of these little tricks that we need to do between the VM and the VFS... hopefully we'll get a lot of that in before 2.4 is released. And that will allow the clean merging of a whole class of different journalled filesystems that have exactly the same ordering constraints that the VM isn't aware of right now. *[54m, 43s]*

And then finally, we've got this write ordering constraint at the SCSI level. Which our man in front was talking about. Right now, the only way that we can do commits safely, is by waiting for the disk to tell us that the entire transaction has hit disk. And only then will you give the commit block to the disk. *[55m, 08s]*

[There is a comment from the audience.]

"That's on media; not just on disk, because there's lots of media."

Yes, on media. Well if a disk has got write-behind battery-backed cache, then that's fine; we don't care what kind of media... as long as it's made persistent. As long as the log updates can be made persistent, then we can allow the commit block to go to disk.

It turns out that SCSI has a very nice feature called tagged command queueing. You can have a whole pile of disk operations, disk writes, outstanding on the disk at once, and the disk can do them in any order it cares. And it will just tell you that it's finished in whatever order it happens to choose. *[55m, 47s]*

There is a bit in the SCSI command called ordered tag. You can actually set an ordered tag bit in the tagged command queue and that is a write barrier to the SCSI layer. The SCSI layer will guarantee, the SCSI disk will guarantee, that no writes that you submitted before that ordered operation will overtake it. And that no writes submitted after it will be sent to disk before that operation. And if we have, in the block device I/O layer, if we have a way of specifying that barrier operation, saying: this is a commit block, don't reorder this block; reorder anything inside of it, but don't reorder this one... if we can set that in the block device layer so that the Linux internal device reordering queues and the disk's reordering queues all observe and honor that barrier operation, then we can keep the pipeline going to the disk, streaming the data to the disk at full speed, without waiting synchronously for the completion of these transactions. *[56m, 59s]*

And that becomes really important for a few special cases. That becomes really important if you're synchronously committing a lot of fast transactions to disk. And there are cases where that really happens. Mail spools and NFS servers... the two canonical examples. Mail spools are constantly updating lots of small files and they want to do an fsync() after each one to make sure they don't tell the sender that the mail has been received until the disk has recorded the fact that it's safe on disk. *[57m, 25s]*

And in the case of NFS, NFSv2 or NFSv1 servers are expected not to acknowledge the write to disk until it's safe on disk. Because the whole point of NFS recovery is that, if the server crashes, the client will replay any of the commands that weren't acknowledged by the server. But anything that was acknowledged by the server is assumed to be safe on disk. And therefore, the server... if that's to work over a server crash, the server can't acknowledge the NFS commands until it's safe on disk, which means that the NFS server is typically doing large numbers of very small data writes, synchronously. *[58m, 00s]*

And in EXT3, we want to be able to have those writes spooling sequentially to a separate disk, which is the journal disk. Remember that the journal will not necessarily have to be in the same disk as the filesystem. So we want to just be able to spool this stuff sequentially onto the journal disk at full speed and having to wait for all of the log to be written before you submit the commit request, the commit I/O, typically means that you're wasting a whole rotational latency in the disk, whenever you're doing a commit. If we can get these write barriers right through the Linux I/O layers, that really, really improves the performance of these streaming synchronous I/Os. That's one thing that's on the cards, it will probably not get into 2.4, but the I/O layers in Linux will hopefully have this in 2.5. *[58m, 46s]*

[The speaker calls on an audience member]. Another question here?

"Well actually, it's a comment. You're assuming that the write ordering implementation in the firmware on the disk actually is functional."

I'm assuming that the write ordering on the firmware of the disk is functional, correct. I assume that when the disk says that it's written something... [Someone comments in background.] Oh yes, I know... We have got, we will only enable this with either a white list or a black listed set of drives. We've already got drive blacklists in the kernel for things that lie hideously... Absolutely, that will not be a default behavior; we have to be very careful about that. *[59m, 27s]*

So the status on this. The core of EXT3 is... the one I'm running on the laptop here is absolutely robust; there are no known problems in that. Some of the kind of user interface issues are not quite 100%. If you delete the journal or you give it an invalid journal inode or things like that, then the actual setup of the journaling can sometimes get a bit confused, but those are management issues surrounding the user level tools which are being used to manage that. *[59m, 56s]*

The user level tools is the main thing that we're working on right now. And the metadata-only journaling. The actual core filesystem is rock solid. It's being used in production web/FTP servers for multiple tens-of-gigabyte filesystems; I trust it absolutely with the laptop here and given the status of some of the device drivers that have been running on this laptop recently, it's rather convenient to have the ability to reboot very, very quickly. [The audience laughs.] *[60m, 25s]*

The e2fsprogs has at the minute... it has minimal support for EXT3, but it is there. It will understand the presence of the recovery bits in the journal bits and will do appropriate things, at least in as much as it will not touch a filesystem that it doesn't know how to touch. And it will not complain about the existence of a journal if the journal happens to be there, but does not need recovery. *[60m, 51s]*

So there is ongoing work for things like the metadata-only journaling. The infrastructure in the JFS for that is all implemented except for the revoke records; the current superblock format, the journal format, does not have revoke records. That's the only thing that needs to be added before I can enable all of that support. *[61m, 13s]*

The quota stuff is in there. It has not been fully tested. I know there are people using it, but it's not tested as much as the rest of the code, so I need to have more testing done on that before I recommend it as being stable. *[61m, 27s]*

e2fsprogs, e2fsck, we already have prototype code in e2fsprogs, e2fsck, in the development branch for doing log replay. The one thing that's missing from that right now is the replay of the orphan lists. Once that's all done, we will have a fully functioning e2fsck, which won't require any kernel support. *[61m, 48s]*

And there are a couple other things which are being worked on. For example, I'm going to be moving the journal from being a regular file to being a reserved file and have a tune2fs function which will allow you to add an arbitrarily sized journal inode to an existing EXT2 filesystem, without it appearing in the filesystem namespace and therefore (without it polluting, sorry) without it tempting people to delete it and nasty things like that. *[62m, 14s]*

So that's basically where we're at right now. The current core EXT3 is stable; it does journal data and therefore it has poor write performance for write-intensive operations. But it is reliable, and it will run... it can be added transparently to any existing EXT2 filesystem. That will be maintained as a stable branch while I'm merging in the metadata-only journaling. So there will be a development branch. This is all 2.2; the 2.4 port will only happen once all this is stable because it's not possible to achieve a sufficiently high level of reliability if you're changing too many things at one time. It just makes it so much harder to maintain and debug. *[63m, 00s]*

There are some things which I'm leaving until after the first complete stable release. And that is, in particular, although the JFS layer understands off-disk journaling and understands multiple block devices sharing the same journal, there are a number of really nasty management issues in terms of administration of that kind of environment. *[63m, 23s]*

Like, what happens if you've got ten filesystems sharing the same off-disk journal and on reboot, one of those filesystems dies? Well you can't actually start reusing the contents of your journal until you've done all of the recovery. And you can't do all of the recovery until you've got all of the filesystems mounted or at least finished... at least until you've found all of the filesystems that use that journal. And if one of those filesystems has disappeared, you can't do recovery onto that filesystem, so therefore you can't start re-using the journal. *[63m, 52s]*

Therefore all of the other filesystems which are sharing the same journal go south. So it's a little problem. We have to do things like making sure that the recovery code has the ability to untangle the different bits of a journal into a separate file and store that in the temp directory somewhere, so that when that missing filesystem gets found later on, we can do the recovery then. *[64m, 17s]*

There are all sorts of little things like that that we have to deal with when you've got off-disk journals and shared journals; which just don't ever come into the picture if you're journaling on the same device the filesystem is on. So that will probably be a post 1.0 issue and one other thing that I want to do is to actually export the nested transaction API into userspace. You have to be very, very careful about that because it's not possible to guarantee proper database semantics. You can't have unbounded, large transactions. You have to have some way in which the user application can get in advance some idea of how many disk blocks it's going to need to modify for the operation, because it's going to call various things like that which are not entirely straight forward; it's not quite as simple as people would hope. But it's sufficiently useful that that will be exported to userspace at some point. *[65m, 07s]*

That's all to solve for EXT3. EXT3 is not the only thing that's going on with EXT2. There are other things that are happening in the EXT2 filesystem space as separate development branches; much like EXT3 is a development branch off of EXT2. And it's likely that some number of these will be merged into a single, new EXT2 variant sometime in the future. There are people increasingly hammering for security support. Access control lists, mandatory access control labels, capabilities... all that type of thing. There are people who really want that in Linux. *[65m, 40s]*

There are proposals from the US Department of Defense that will forbid them from purchasing any operating system which does not have (these facilities) these capabilities. And they have been trying to persuade the US Government to adopt the same rules. Although they are resisting that. *[66m, 00s]*

There's B-Tree support. B-Trees are fairly complicated on-disk structures. There are people who want to have B-Tree support (in Linux) in EXT2, for scalable directory performance. But B-Trees can go really horribly haywire if you interrupt them in the middle of a tree balancing operation. And so making them consistent over a reboot either requires that you're very, very careful and do lots of extra I/O in the disk structure to make sure that it can be recovered sanely, or you do journaling. *[66m, 26s]*

In fact, there's a whole section of code inside reiserfs which was dealing with exactly this issue. Which has basically been eliminated from the filesystem now that they've got journaling, because journaling deals with all that for them. Putting B-Trees into EXT3 really requires JFS as a prerequisite. *[66m, 47s]*

Online resize is being done for EXT2. And there are a number of other minimally intrusive extensions that... EXT2 has this advantage of being a very nice, simple filesystem. And you can do B-Tree extent maps inside your files as a very efficient way of encoding large files on your disk. Now you don't actually have to use B-Trees for that. If you don't use B-Trees, you can just have a very simple extent map structure on the disk, which maps entire contiguous extents of on-disk blocks for a single file in just a few bytes in the directory structures in the inode. *[67m, 23s]*

That is all well and good until you start doing things like having holes in the files and then wanting to write into the middle of those holes. Once you do that, then you're having to shuffle around all of your extent maps and so on. And that gets really complicated; that's normally why people want B-Trees for extent maps. If you're willing to forego with the ability to write into holes into your filesystem, you don't need B-Trees for your extent maps. *[67m, 48s]*

So that's something that we could do in EXT2 as an experiment to say: what's the minimal necessary modification to this filesystem to provide all of the benefits of extent maps, except for this facility with holes; which are going to be used by a vanishingly small fraction of users. We don't need to do B-Tree directories; we can maybe do hashing of the directories; a very much simpler extension to the filesystem, which can give us many of the performance improvements of B-Tree directories, but without the implementation costs. So really there are lots of these things that people are looking at with EXT2. *[68m, 24s]*

So EXT3 will be supported as a fully functional filesystem; it's not clear whether it will be a set ever merged into the official EXT2 source tree. I guess not, actually. But it's certainly not the only extension to EXT2 that's being talked about right now. *[68m, 40s]*

Any questions? You've got two-and-a-half minutes... two minutes. [The audience laughs.] One at the back.

[There is an unintelligible question from an audience member.]

...In progress, user reserved inodes for the journal. *[69m, 13s]*

[The audience member continues.]

"Second part is, usually when you create a journal, we do a dd..."

No, I've told you there's going to be a tune2fs options which does all of that for you and puts it into a reserved inode; that'll all be invisible. *[69m, 31s]*

[There is an additional question about having to disk seek to the journal every time you do a write; a problem similar to those with FAT filesystems that need to write to the FAT for every disk write.]

[There is a comment from another audience member on the subject.]

"You can get around it by putting it on a different disk."

Yeah. There are actually a number of different issues there. *[70m, 06s]*

The first issue is that the journal writes are batched up into commits every five or ten seconds. And they're sequential, so there's only one seek to the journal and one seek from the journal for that operation. And the second thing is that, because you are doing those updates into the journal, things like hot data... things like inodes which are being constantly updated or directories which are being constantly updated, are always hot in the journal and they never have to be written back to disk. So you can actually eliminate the random seeks all over your main filesystem data, because all of the data is currently in the journal. So you're reducing the number of seeks in that case. And you can also put your journal on a separate disk entirely, if you want to. So our experience with journaling filesystems suggests that this is not the problem that you think it will be. Obviously, it depends on the workload. *[71m, 04s]*

[The speaker calls on a different audience member for a question.]

“Beyond the write boundary, do you anticipate other changes to the block driver interface?”

Do I anticipate other changes to the block driver interface? Not right now. There are other changes going on, but they’re not related to EXT3. In particular, there’s work going on to replace the buffer head interface to the block device layer with a kiobuf-based implementation. And that’s going to be enormously more efficient; it will allow us to cleanly access high memory pages on large memory Intel boxes; it will allow us, if we get it right, to finally make the break and allow addressing of block devices larger than two terabytes. So yes, all of that’s going on, but it’s not related to EXT3 work. *[71m, 54s]*

[The speaker calls on another audience member for a question.]

What about crash during recovery? It doesn’t matter. Recovery just consists of going through the log and writing what’s in the log back to disk. And writing it back to disk twice is just as good as writing it once. So if you crash during the recovery and you’ve written half of the log, well the next recovery goes along and just does all the same set of writes, plus a few more. And only when the recovery is completed, do we make the modification to the journal which marks recovery as being done. There are no modifications to the journal while recovery is in progress. So it just works. There’s no problems there. *[72m, 32s]*

[There is another question from the audience.]

Does this at the moment work with software RAID? It will work with hardware RAID; it will work with software RAID once the 2.4 port is done, because a bug in software RAID has been fixed in 2.4. It will not work with software RAID in 2.2. The reason for that is that software RAID when it does RAID recovery after a crash works by doing a buffer cache read, stripe-by-stripe, through the whole disk, writing those stripes back. And when it writes those stripes back, it updates the disks which were not consistent at the time of the crash. Unfortunately, when it’s doing that, it’s causing the contents of the buffer cache to be written to disk without the filesystem’s say-so. And therefore is violating the write ordering requirements of the journaling filesystem. And EXT3 adds a whole pile of debugging code to the block device layer to detect violations in the write ordering requirements, because that's really good for debugging. You don't want these things to happen silently. *[73m, 32s]*

Write ordering violations are really, really hard to detect, because the only way you'll detect them normally is by crashing and finding that something doesn't work right when you recover the filesystem. You've got absolutely no idea what went wrong. So having those debugging entries in the EXT3 code is really, really important, to make sure that we detect write ordering violations when they happen. And the RAID devices violate ordering and recovery... not in 2.4. So it'll be fine in 2.4.

There's a question over here somewhere? ... Nope... Thank you very much, then.

[The audience applauds.] [The presentation ends.] *[74m, 17s]*

3. Additional resources

3.1. EXT3 distribution

The EXT3 filesystem patch distributions and design papers are available from ftp://ftp.kernel.org/pub/linux/kernel/people/sct/ext3

Alternately, these materials are available from ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/

The EXT3 author and maintainer, Stephen Tweedie, may be reached at sct@redhat.com (mailto:sct@redhat.com)

3.2. e2fsprogs

e2fsprogs is available from http://e2fsprogs.sourceforge.net/

Theodore Ts'o is the current maintainer for e2fsprogs.

3.3. Other materials mentioned in this talk

The InterMezzo filesystem distribution materials and information are available from http://inter-mezzo.org/

The Linux Memory Management (MM) team home page is located at http://www.linux.eu.org/Linux-MM/

The Ottawa Linux Symposium pages are located at http://www.ottawalinuxsymposium.org

Finally, an excellent set of kernel-related materials (highly recommended!) and the home page for the #kernelnewbies channel is located at http://www.kernelnewbies.org/