Who wants another filesystem?

Neil Brown

February 6, 2003

Abstract

Q. With ext3, reiserfs, reiser4, xfs, jfs (just to name a few), who in their right mind would write another general-purpose filesystem?

A. Wrong question. They are all special purpose: Each has a particular focus.

The goal of lafs is to provide optimal service for a different special purpose: as a departmental file server. Specific sub-goals include being friendly to:

- NFS - low latency writes
- backups - snapshots for on-line incremental backups
- quotas - tree based quotas
- admins - easy resize/migration, crash recovery
- RAID - stripe-wide writes. always.
- users - fast, reliable, stable, efficient, doesn't crash, doesn't corrupt data, stable. Did I say reliable?

lafs (pronounced "laughs") is a log structured filesystem that is designed to meet all these goals.

Note: This paper will also be available after the conference at `http://www.cse.unsw.edu.au/~neilb/conf/lca2003/`.

1 Introduction

In early 1999 when I first started seriously "getting into" linux, there was no real choice in filesystems. ext2 was the only real option. Linux could of course access FAT based filesystems, and even had UMSDOS which provided a POSIX layer on-top of those filesystems. But in terms of a main filesystem for a Linux-only system, ext2 was it. ext3 was promised, and reiserfs was rumored but they weren't stable options yet.

Now, in early 2003, we have almost an embarrassment of riches. ext3 and reiserfs are here and reasonably mature. XFS from SGI and JFS from IBM have arrived and are integrated in the development kernels at least. We even have reiser4 on the horizon which promises all sorts of interesting things.

Against this back drop one might wonder why anyone would want to create yet another filesystem as this paper proposes. Is there not enough choice

already? and would not any effort in filesystem development better be used in supporting one of the existing projects?

My answer to this is "No". While it may seem that we have a wide selection of fairly general purpose filesystems, the fact is that they all have a particular focus and don't necessarily try to meet every need.

Ext3 is very significantly an extension of ext2. That fact that the on-disc structure is compatible with ext2, and that the wealth of experience with ext2 can equally apply to ext3, and that the very well war-hardened e2fsck works equally on ext3 are all very strong motivators towards using ext3, and also are chains that limit the enhancement of ext3 to take advantage of new ideas in filesystem technology.

Reiserfs appears to have a strong emphasis on efficient handling of small files and large directories. XFS has a particular emphasis on very high data throughput as needed by for multimedia applications, as well as compatibility with an established base of IRIX systems. JFS I am less familiar with, but while it, like the others, is a well rounded and very usable filesystem, there are certain features that I would like to see in a filesystem that are not present in these.

The particular focus for the filesystem described in this paper is to be a filesystem for a departmental file server, particularly a file server which serves the NFS file access protocol. The core technology used for this filesystem is the concept of a log-structured filesystem (LFS).

The remainder of this paper will give a brief overview of a log structured filesystem, and then present some particular needs that the departmental file-server has, and show how they can be met using the log structured filesystem. Not all needs require the specific characteristics of a LFS (though many do) but all are beneficial in the particular context and are not generally available elsewhere.

2 A primer on Log Structured Filesystems

The concept of a Log Structured File System was first developed by Mendel Rosenblum [1] and was extended by Margo Seltzer et.al. for the BSD operating system. logfs has been available on some flavours of BSD, but does not appear to be well supported.

A couple of effort have been reported which aimed to create a log structured filesystem for Linux, possibly the most notable being LinLogFS, previously known as dtfs[2].

LinLogFS was hampered somewhat by being developed on a 2.2 kernel which would have made much of the implementation much more awkward than in a 2.6 kernel. It also embodied some design decisions that do not fit with my goal. Thirdly, development seems to have stopped nearly 3 years ago. For these reasons, LinLogFS was not considered as a starting point for LaFS.

To understand how a log structured filesystem works two basic ideas are needed. The first involves storing the entire filesystem as a tree, providing a single starting point for all addressing. The second involves dividing the storage device into large segments to which new data is written.

[1] ftp://ftp.cs.berkeley.edu/ucb/sprite/papers/thesis-Rosenblum.ps
[2] http://www.complang.tuwien.ac.at/czezatke/lfs.html

Representing a filesystem as a single tree is not an idea unique to log structure filesystems. reiserfs works this way as does TUX2. Indeed any Unix filesystem is largely a tree already, as there is a 'root' directory and all files and directories are attached to that single root by way of one or more intermediate levels in the tree.

In a traditional Unix filesystem, the parts of the filesystem that are not addressed as part of a tree are the inode table and the map of free blocks. Rather than having locations on the device described by higher levels in the tree, these have fixed locations on the device.

Taking such a filesystem, and storing the inode table and the free-block map in special files is a simple way to create a filesystem that is fully based on a tree and this is exactly what TUX2 does.

The result of having all the metadata in files is that the only datum that has a fixed location on the device is a pointer to the root of the tree. This root then points, possibly via indirect blocks, to the file containing all inodes. One of these inodes addresses a file that contains the free block bitmap. Another inode contains the root directory. Following this pattern, the entire filesystem can be found.

Once we have all addressing for the filesystem in a single tree we have a great deal of flexibility about what data gets store on which part of the device. As there are no fixed locations, anything can be stored anywhere on the device. This leads us to the second important aspect of a log structured filesystem — the segments.

The device is conceptually divided into a number of segments, each a few megabytes in size. When data needs to be written to the filesystem, an unused segment is found, and all dirty data is written to that segment together with any indexing information needed to find the data. This could involve:

- Some data blocks from a number of modified or created files.
- A few indirect block that contain the addresses of some of those blocks, where the file is too large of all addressing to fit in the inode.
- Possibly some blocks of directories where files were create or deleted.
- Inodes for all of the files that have been changed containing (at least) new addressing information.
- Possibly some indirect blocks which store the addresses of these inodes in the inode file.
- Finally the inode for the file that contains all other inodes is written

All this data is written contiguously into one or more segments, and finally the address of the root inode is written to some fixed location on the device. From that inode, the entire new state of the filesystem can be found.

This approach seems simple and elegant, until you run out of unused segments. At that point is suddenly seems horribly flawed. To get us out of this sticky situation, we have a service called a *cleaner*.

As files are created, updated, and deleted, some segments that at one time were full of useful data, will eventually become full of uninteresting, unreachable data, and maybe just a few blocks of live data. It is the task of the cleaner to

identify those segments which have just a few blocks of interesting data, and to gather the interesting blocks from a number of segments together and relocate them to a new, previously clean, segment. Once this is done, all those old segments can be marked as 'clean' and can be re-used.

To help the cleaner in its task we store some extra metadata in the filesystem. These include:

- A segment usage table that keeps a count of the number of live data blocks in each segment. This is kept up-to-date as new blocks are allocated and as old blocks become inaccessible.
- An 'age' for each segment giving an ordering to all segments indicating how old the data is. As newly written data is likely to be more volatile than data that has been stable for a while, this is used to help guide the cleaner. Cleaning a new segment is not as productive a cleaning an old segment as, if we wait a little while, the new segment my soon have much less live data in it that it does now and so can be cleaned more cheaply.
- Summary information in each segment. This summary information indicates, for each block in the segment, which file it is part of, and which block in that file it is. When the cleaner chooses to clean a particular segment, it reads this summary information and checks each file/block to see if it is still live in that segment, or if it has already been relocated by an update. Any block that has not already been relocated gets relocated. This will bring the usage count for the segment down to zero, at which point it can be re-used.

The summary information is not really per-segment, but is per-write-cluster. A write-cluster is a header containing the summary plus a number of data and metadata blocks that are all written at once. A write cluster may span an entire segment, but in the face of synchronous writes there could be many write clusters per segment.

From the above description, it would appear that to update a single block, and to force that change safely to disk, would require writing the block, possibly an indirect block, the inode, an indirect block to locate the inode, and the root inode, and then recording the new location of the root inode in its fixed home. Performing lots of synchronous writes, as an NFS server typically does, would cause a substantial waste in write throughput.

However because of the summary information in each segment we do not need to do this. To flush a single block to storage, all that is needed is to write out the block together with some summary information that describes it. Each write cluster contains this summary information and also the address of the next write cluster, and some information for internal validation. This allows recently written write clusters to be found at filesystem mount time, and to be incorporated into the filesystem through a process called roll-forward. Thus committing a single write cluster is all that is needed to commit a block of data.

Typically several blocks will be written together even when doing synchronous writes, so the overhead will only be one block for the summary information, rather than several blocks for all indexing and inode information.

Some useful characteristics of a log structured filesystem as outlined above are:

- Data is written in large contiguous blocks, so a very high write throughput can be achieved, at least in smallish bursts.
- Data is only ever written to unused portions of the device, so we never over-write live data. This means that an interrupted write can never corrupt live data.
- Recently written data can easily be found and examined, so an unclean shutdown need not cause any filesystem inconsistencies.
- Data is continuously being re-arranged by a background task — the cleaner. This means that file defragmentation can easily be incorporated.
- As new data does not over-write old data, it is possible to keep old images of the filesystem accessible, as long as we don't run out of storage space. These old images are referred to as 'snapshots'. In order to manage cleaning in the presence of snapshots, we keep a segment usage table for each active snapshot. Segments used by a snapshot typically do not get cleaned until the snapshot it released.

Some other aspects of lafs that are not general to all Log structure filesystem, are not specific to any of the needs which will be discussed in the following sections, but are none-the-less interesting are:

- Inodes are stored one per block. This is largely because an LFS gains no particular benefit, and some costs, from storing multiple inodes in a single block. Thus small files will be wholly contained within the inode, and many large files will have all index information in the inode.
- Indexing of large files is achieved with a B-tree that addresses extents rather than individual blocks (but that is pretty common these days).
- As noted, we need to keep an age for each segment, and we need to keep the table of ages in memory for easy access by the cleaner, so we don't want to use too many bits. To achieve this we note that we do not need a high level of granularity for old segments: very old and extremely old are much the same as far as the cleaner is concerned. So we store the age in a 16bit field and when we get close to running out of new values, we decay all values currently in the table: values less than 32768 are halved, values greater have 16384 subtracted. The means that we can still differentiate where we need to without wasting bits. The summary information in each segment stores an actual 64bit age (well, date-of-birth really) for the rare occasions when it is needed.
- Access time is a fairly frustrating aspect of filesystem design, as it needs to be updated often, accessed whenever an inode is accessed (whether the application will use it or not) but is very rarely used.

  lafs does not store the access time in the inode, but rather stores all access times in a special non-logged area of storage that will be described in more detail in the next section. Access times are thus not guaranteed to be uptodate and can more easily be corrupted, but are mostly close enough.

3 Supporting volume management

I should come right out and say it: I am no great fan of Logical Volume Management, or LVM.

It is not that I don't think there is a need for it: there clearly is. People want flexibility in arranging how their storage space is used. It's not that I think LVM shouldn't be allowed: clearly is should as it allows lots of people to do things that they otherwise couldn't. However I still don't like it. It isn't the right solution.

A typical logical volume manager takes one or more large devices and divides them up into chunks. It then assembles some of those chunks together into one or more 'volumes' or virtual devices.

A typical use of a volume is to store a filesystem in it. The filesystem sees the large contiguous piece of storage, and divides it up into little pieces, and then re-assembles those pieces into one or more files, which are in many ways similar to volumes or devices.

This double layering seems inappropriate. Why have two layers that present abstractions of variable sized continuous storage from fixed size storage objects, when only one will do.

LaFS addresses this issue by allowing a multitude of devices to be provided as storage for the filesystem data. It also allows for a multitude of filesystems to be stored in those devices, though usually one filesystem with a multitude of directories will be enough.

This is not a unique idea: the Advanced File System in Tru64 from HP/Compaq/Digital contains similar ideas. However it is not a widely used idea, at least in Linux.

Further, this idea does not particularly require log structuring to make it work. However some aspects of a log structured filesystem to work quite well with the multiple device concept.

With a traditional LVM, it is quite easy to make a volume grow larger, and then tell the filesystem in the volume to use the extra space (if the FS knows how to do that). However shrinking is not quite as easy. Making a filesystem shrink will invariably require relocating some data which many filesystems do not support.

However a LFS has a cleaner which is continuously relocating data. If we wish an LFS to stop using some component device, we can simply adjust the cleaning heuristic to prefer to clean segments from that device, even if they are full. If we also inhibit clean segments from that device from being allocated, then the device will eventually (depending on how fast we push the cleaner) become completely unused and so can easily be removed from the filesystem.

Another piece of functionality provided by some Logical Volume Managers is creating a snapshot. When this is enabled, write requests are trap and if the data to be over-written is from before the snapshot, it is relocated first. Thus an image of the filesystem can be kept stable (e.g. for backups) while other changes are still happening.

Again, this functionality can better be provided by a filesystem than by introducing an intermediate layer. When the filesystem is asked to take a snapshot, it can simply choose not to over-write any data that was live at the time of the snapshot, so no data relocation is needed. Lafs provides this functionality quite naturally.

Lafs goes a little way beyond just making an underlying LVM irrelevant. It also provides some LVM functionality itself. Lafs can mark some segments to be non-logged so that they do not take part in the normal logging and cleaning process. A file can then be created so that all the data blocks of this file (but not the inode/index blocks) reside in these non-logged segments.

Such a file will not benefit from the various data-protection features of an LFS, but also will not suffer terrible fragmentation in the face of lots of random updates. Such a file would have limited uses, but would be very valuable in those limited cases. They include:

- A swap file. Swap does not require integrity protection on unclean shutdown, but does receive lots of random updates, and so would be very suitable for a non-logged file. Using a non-logged file for your swap partition means that your whole device can be in one lafs.

- A file storing all access times. As mentioned earlier, access time stamps of files are problematic. They are occasionally wanted, but do not need to be guaranteed always accurate after a system failure. Having all of them in one big non-logged file means that access is fairly fast as there is a good chance you will find what you want in cache, and updates do not cause otherwise stable files to be continually relocated in the filesystem.

Another offshoot of having lafs know about multiple devices is that it can have a concept of devices with different performance. For example, lafs can have an NVRAM device which is of limited size but has very low write latency, and a RAID5 array which is much larger, but slower.

Any new data would be written to the NVRAM device using fairly small segment sizes, and the oldest segments on the NVRAM device would be cleaned off onto the RAID5 array as needed to make more room. This would cause the NVRAM to effectively work as a low-latency write cache in front of the RAID5 array.

In general, having the filesystem talk directly to the device rather than through an intermediate LVM layer provides more opportunities for the filesystem to make informed decisions on how to use the device.

4 Supporting RAID

There are three important ideas used in RAID - Redundant Arrays of Independent Devices: data duplication, data striping, and redundancy through parity. These are combined in different way to form different RAID levels.

Data duplication is of little interest to a filesystem. It does not effect the performance characteristics of a storage device in a way that a filesystem can particularly make use of.

Data striping can usefully be recognised by the filesystem. If the filesystem knows which data blocks are on different real devices, then it can make decisions about data layout to try to avoid imbalance (don't put all the meta-data on the same device, or it will be a bottle neck) and to reduce 'head contention' - e.g. by trying to make small files live entirely on just one device, so that multiple files can be accessed independently in parallel.

Redundancy through parity can be a substantial performance hit for random updates as a write request will often need old data blocks to be read first so that parity calculations can be performed. Any assistance that the filesystem can give in patterning it's access to decrease pre-reading is a good thing.

The best write access pattern for parity based redundancy RAID levels, such as RAID-4 and RAID-5, is to write whole stripes at a time. If this pattern is followed, then old data or old parity will never need to be read first, and the full device bandwidth can be given to actually writing the data.

Further there is an interesting complication with these raid levels. If a system suffers an unclean shutdown (e.g. power failure) while writing out some data and matching parity, it will not know on restart whether the parity blocks are correct, and will have to regenerate them. If after the unclean shutdown and restart, one of the devices in the array is found to be missing or faulty, then the parity regeneration will not be possible and some unknown amount of data will be undetectably corrupted.

In particular, any data block on a failed drive on a stripe that was being written at the time of failure must be considered to be lost. In a traditional filesystem, this could potentially be a block in some file that hasn't been touched for months. The probability of corruption is fairly low, but it is a very real possibility and so is not acceptable.

So the ideal access pattern for a filesystem to use on a RAID array that uses parity, such as RAID-4 or RAID-5 is:

1. To always write whole stripes. i.e. when it writes a block on any device, it writes to the same block on every device (except the parity device).

2. To only write to stripes which have no live data, so that live data cannot become corrupted by an unclean shutdown followed by a restart with a missing device.

3. To be able to confirm after the fact if data was written successfully, so that corruption of just-written data can be detected.

4. Layout policies that understand the arrangement of real devices and places data appropriately.

LaFS addresses all of these issues. Issue 1 is easy to achieve by padding, where necessary, all write clusters to be a multiple of the stripe size.

Issue 2 is fundamental to the nature of log structuring providing segments are aligned with stripes, and write clusters are padded as above.

Issue 3 is needed as part of the roll-forward mechanism in mounting a LFS anyway.

Issue 4 can be addresses as follows.

The preferred parity based raid level for LaFS is RAID-4. The advantage that the more common RAID-5 has is that parity blocks are distributed among all devices so parity updates are also distributed. However the practice of writing whole stripes at a time makes that advantage irrelevant, and the more simple rectangular layout of RAID-4 gives it an advantage.

Further the preferred style of addressing within a RAID-4 array is to have the first device as the parity device, and to address blocks in remaining devices in a linear fashion. All blocks in the first data device come first, followed by all blocks in the next device and so on.

LaFS is able to handle the non-contiguous segments with some simple arithmetic, and we gain the advantage of being able to add devices to the array quite easily. A device can be zeroed and then added, and parity will remain correct. LaFS can then be told about the extra space. This extra space will be realised as a little bit of extra space in each segment. It will not become available instantly, but as the cleaning process starts filling up, and freeing up, these larger segments, more space will become available.

Whether the RAID-4 uses a traditional chunk-based addressing or an expandable linear addressing, it will 'know' which blocks are on the same device and will be able to lay data out to make best use of the devices.

Another advantage of RAID-4 is that, to LaFS, a RAID-4 looks exactly like a RAID-0, so the knowledge of the rectangular addressing can be equally useful for both.

Thus with some careful thought, the fact that a LFS always writes in largish contiguous blocks, means that it can behave optimally for RAID arrays.

5 Supporting NFS

The NFS file access protocol (prior to version 4 at least) placed two primary requirements on a filesystem (beyond obvious things required by the POSIX interface). These relate to filehandles and synchronous writes.

The usage of 'filehandles' in NFS requires that the filesystem be able to present an small fixed length identifier for every file that is universally unique over all time. While the file exists, it must always have the same file handle, no other file may ever have the same filehandle, and the filehandle must be enough to find the file.

Combining the inode number with a per-inode 'generation' number that is different each time the inode number is reused is a typical and adequate approach to this need, and is shared by many filesystems, including LaFS.

The other need, for synchronous writes, is more interesting. The specification for NFS requires that every change is stored safely on stable storage before the response is sent. This means that writing to stable storage must have very low latency. NFS version 3 relaxes this requirement for file data, but it still holds for directory updates (create, unlink, etc) so good behavior in the face of synchronous writes is still needed.

The log-structured filesystem helps here by allowing 1 or several updates to be committed safely in a single write to a contiguous section of the device. This greatly reduces the number of costly seeks that are required

While this feature is helped by the log structuring, it is not unique to a LFS. Filesystems that can do full data journaling, such as ext3 and more recently reiserfs, can achieve similar low latencies.

When an NVRAM device is included in a LaFS as described in the section on volume management, we get even better write latency and can provide even better NFS performance.

6 Supporting backups

Maintaining backups is possibly one of the more frustrating aspects of system administration. It consumes substantial resources such as storage media and storage devices, puts a noticeable burden on the fileservers while the backup is happening, requires regular monitoring, and occasional attention, but is comparatively rarely used to actually restore anything. Of course when a calamity does hit and you loose some data, you are very thankful for the backups.

If the filesystem can make backups less burdensome and more effective in any way, then it will make life easier for system administrators.

Three sorts of backups that we make at UNSW/CSE can be described as Calamity backups, Archival backups, and Carelessness backups. We will look at these in turn and see how the filesystem can help.

6.1 Calamity Backups

Calamity backups are targeted at restoring a complete filesystem in the event of major failure. These backups need to be able to restore a complete filesystem to a recent state, quite possibly on different hardware. We currently take a complete CPIO archive of each filesystem every 8 days or so, and rely on incremental backups to fill in the gaps.

The main problems with this are that we repeatedly (every 8 days) backup lots of data that we already have backed up, and that traversing the filesystem in logical order (by directory, and then by file) is unlikely to involve accessing the disk in physical order, so there will be lots of seeking to gather the various parts of directories together.

In a past life we used to take calamity backups by simply dumping the underlying block device to tape. This meant better disc throughput (which, given the speed of disc drives at the time, was a good thing), but it meant that we needed a spare partition at least the size of the largest active partition for restoring from these backups (sometimes we do need to restore selected files from calamity backups). This also meant that we were dumping a lot of blocks that did not contain live data.

What would be nice is something in between the two: To be able to dump largely in the order that data is on disc, but still to use information from the filesystem to avoid dumping too much dead data, and to arrange the dumped data so that piecemeal restoration is possible.

The structure of a LFS makes such an in-between position possible by focusing of segments.

- If we dump a whole segment at at time, then we will be accessing the device largely in device order so throughput should be high.
- If we access the segment usage information and only backup the segments that have live data then we will avoid backing up a lot of dead data.
- If we examine the segment age information and backup the segments in reverse chronological order, then we will always have indexing information on tape before the data it indexes, so a single pass through the tape will be enough to extract an particular subset of the stored files. As the age information does not have complete granularity (old segments are grouped

together with a single age, so that age can be stored in 16bits) we will need to read the segment headers to get the final ordering, but this should not increase the overall access time too much.

- Finally, given the age information stored about segments, it is very easy to determine which old segments have been dumped in a previous Calamity dump and so do not need to be dumped every week there after.

Combining these ideas we can produce a Calamity backup scheme that is reasonably efficient at creating backups, but exposes some complexity when it comes to restoring backups. This is probably a reasonable trade off as restorations happen much less often than backups are made.

6.2 Archival backups

Archival backups are targeted at restoring some or all of an individual user's home directory, possibly after an account has been closed and deleted.

Due to the long term nature of archival backups it is not appropriate to tie them to a particular filesystem format, and so a scheme similar to the Calamity backups would not be appropriate even if it were practical.

A cpio or tar archive of the home directory is likely the best option for these backups. The main burden that this imposed on a filesystem is the fact that files are normally collected sequentially and there is no opportunity for read-ahead between files. A better approach would involve submitting read requests for multiple files and directories in such a way that throughput is valued over latency, and the order of the returned data is not important. Then the filesystem could submit large number of requests to a device before waiting for any of the data, and so provide lots of opportunity for re-ordering and grouping requests and thereby improve throughput.

It may be possible to use the new asynchronous IO primitives in 2.6 to achieve this improved thoughput. If not a special purpose interface to the filesystem will be provided to provide optimal access for backups.

6.3 Carelessness backups

The third type of backups that we support are Carelessness backups which allow for restoring of individual files that have accidentally be lost or corrupted. The use of snapshots as mentioned earlier provide this for the short term of a day or maybe a week, depending on available storage space. However this is well short of the 12 months that we currently provide.

We provide this long term for backups without excessive space usage by ignoring many classes of files that are expected to be very uninteresting, such as very large file, automatically created files (dot-o files) and other classes. This is not possible with snapshots which are very much an all or nothing approach.

Exactly what scheme we will use when this filesystem goes live is not yet clear, but I suspect a heuristically limited incremental backup scheme similar to our current scheme, but using the high-bandwidth interface that was created for archival backups will meet our needs quite effectively.

7 Supporting quotas

Imposing storage quotas in a Unix filesystem is an interesting problem. It is interesting because there is an obvious approach that has been used quite successfully for twenty years or more, and which is wrong.

The obvious approach is to impose a quota for each user, and to charge the space used by each file to the user who owns the file, and then to prevent further allocations if the user's charged usage exceeds the user's quota. This works quite well quite often, but fails occasionally and so is imperfect.

One situation where it fails if when you have a group project that is being worked on by a number of members of a group. Files shared by a group could be owned by any member of the group, and the ownership could easily change as files are edited (if the editor creates a new file to replace an old one). In this case, one really wants a quota for the group, not for the members.

An apparently obvious solution is to allow quotas to be imposed on groups rather than users, and to charge file usage based on the group id of each file. This would solve the above problem, but creates others. Some people want to use group ownership purely for access control. In these cases it would not make sense for the group to have any quota as it isn't an entity that owns files so much as one that owns access rights.

Even here, it seems the we can work around the problem. We simply have some filesystems where the quota is based on group ownership, and some where the quota is based on user ownership. This can solve each problem, but introduces administrative complications. It is not always easy to know ahead of time what usage patterns are expected, and so whether to put certain files on a "group" filesystem or a "user" filesystem.

At UNSW/CSE we have tried all of these solutions and none of them are satisfactory.

On reflection we can see that there is something fundamentally inelegant about this whole approach. When faced with the possibility of imposing a quota on usage by individual users, one must face the question: Does the quota apply just on a single filesystem, or across all filesystems? When dealing with individual computers, it is possible to implement both alternatives, and at UNSW/CSE we have tried both, and each have their problems.

When dealing with a network of computers, having a single quota is impractical and so defining per-filesystem quotas is the norm. However neither answer is really good, and the problem seems to be that the question needs to be asked at all. A model for quotas that requires that sort of question to be asked seems to be fundamentally inelegant. It would be nice to have a model where that question simply didn't exist.

For the past year at UNSW/CSE we have been using a new model for quotas and all of the problems have simply disappeared. This new model is referred to a "Tree quotas". The idea is that usage of a file is charged to the 'tree' that it is in, and each tree has a quota. The ownership or group ownership of files within the tree is irrelevant. It is only the location in a particular tree that is important.

A 'tree', for the purposes of storage accounting, is considered to be any subtree of a filesystem, rooted at a directory not owned by root, and for which all ancestors in the filesystem are owned by root. Thus in a typical installation, every user's home directory would form an individual tree. This allows us to

impose a quota on every home directory, and to not care about how the home directory is used, and what the ownership and group ownership of the files within that directory are.

We currently implement tree-quotas using a patch which makes use of a reserved-but-unused field in the ext2 and ext3 inode to store the tree-id of each file. This has proven very useful. However it seems unlikely that the maintainers of ext2 and ext3 will want to give up a field in the inode (a scarce resource) for treequotas.

So LaFS will have full integrated support for tree-quotas. This does not depend on the log-structuring of LaFS at all and could be implemented in any filesystem. However, except for our ext2/3 patches, it currently isn't, and we need it.

8 Supporting reliability

Reliability is a very different issue to all of the other issues addresses above. While they require consideration in the design of the filesystem layout, and to a lesser extent the design of the code, reliability needs to be addressed in the coding process.

To some extent, nothing can replace a meticulous attention to detail and a thorough understanding of the task, the implementation, and the interfaces. However some techniques can be used to ease the task.

In my experience, two techniques that are very helpful in improving reliability are documentation and prototypes.

Possibly one of the best examples of Documentation aiding reliability is the TeX system by Donald Knuth. While it is certain that much of the credit for the success of TeX and the stability and reliability of the code can be attributed to Donald's brilliance with programming, some must certainly go to his substantial efforts to document what he was doing. The TeX program is a wonderful example of the Literate Programming style that Knuth pioneered and shows that writing technical documentation hand-in-hand with the code has very positive results.

A little closer to home, Andrew Morton - kernel hacker and ext3 developer has said more than once that the excellent documentation that Stephen Tweedie (original ext3 developer) put in the ext3 code was big part of why Andrew was able to pick up the 2.2 code base and port it to 2.4 with great success.

It recognition of this, LaFS is being documented in detail while the code is developed. While this is not a complete panacea, it will make the project much more manageable.

The other technique I suggested is prototypes. To build a good system, it is invaluable to build a good prototype first. It is also dangerous to decide that the prototype is good enough and to put it into production.

Developing a prototype is, of course, a good way to test a design. While building it you discover all the little fiddly details that you didn't think about while you were concentrating on the big picture. Some of these you fix in the prototype. Some you wont be able to because the fundamentally affect the design, and you only work around.

If you try to use the prototype in production, it will still have those warts that you couldn't completely fix and they will eventually bite you. If you throw

away the prototype once finished — keeping all the lessoned learned (in your documentation) but discarding all the code — and start again, you will inevitably build a much better system. It can be a lot more work, but it can be worth it in the long run.

So I recommend building a prototype, preferably in a way that you cannot possibly use it in production. One approach is to write the prototype in a different language. Use a language that is suitable for prototyping but which you won't want your final product to be written in, such as Perl.

The approach used for LaFS is to build a prototype that runs as a user-space program. It can do all the file manipulations and data management that the final code will have to do, but as it runs in user-space and not kernel-space, many consideration such as locking, and memory management will be very different and so a complete re-implementation will be necessary.

Hopefully, these two techniques, together with constant attention to detail will help LaFS to be reliable and correct.

9 Current Status

As of the writing of this paper, comprehensive design documentation is well underway with most of the on-device layout established and many of the algorithms described. A prototype which can read files by inode number and can perform fairly simplistic layout of new files has been written. It is currently undergoing a re-write to provide more general block layout. Kernel code has not been started and will not be until the documentation nears completion and the prototype is managing directories and cleaning well.