The Linux Implementation of a Log-structured File System

Ryusuke Konishi
ryusuke@osrg.net

Yoshiji Amagai
amagai@osrg.net

Koji Sato
koji@osrg.net

Hisashi Hifumi
hifumi@osrg.net

Seiji Kihara
kihara@osrg.net

Satoshi Moriai
moriai@osrg.net

NTT Cyber Space Laboratories, NTT Corporation
1-1 Hikari-no-oka, Yokosuka-shi,
Kanagawa, 239-0847, Japan

ABSTRACT

Toward enhancing the reliability of the Linux file system, we are developing a new log-structured file system (NILFS) for the Linux operating system. Instead of overwriting existing blocks, NILFS appends consistent sets of modified or newly created blocks continuously into segmented disk regions. This writing method allows NILFS to achieve faster recovery time and higher write performance. The address of the block that is written to changes for each write, which makes it difficult to apply modern file system technologies such as B-tree structures. To permit such writing on the Linux kernel basis, NILFS has its own write mechanism that handles data and meta data as one unit and allows them to be relocated. This paper presents the design and implementation of NILFS focussing on the write mechanism.

1. INTRODUCTION

As use of open-source operating systems advances not only for stationery PCs but also for backend servers, their reliability and availability are becoming more and more important. One important issue in these operating systems is file system reliability. For instance, applying Linux to such fields was difficult several years ago because of the unreliablity of its standard file system. This problem has been significantly eased by the adoption of journaling file systems such as ext3[9], XFS[2], JFS[1] and ReiserFS[3].

These journaling file systems enable fast and consistent recovery of the file system after unexpected system freezes or power failures. However, they still allow the fatal destruction of the file system due to the characteristic that recovery is realized by overwriting meta data with their copies saved in a journal file. This recovery is guaranteed to work properly only if the write order of on-disk data blocks and meta data blocks is physically conserved on the disk platters. Unfortunately, this constraint is often violated by the write optimizations performed by the block I/O subsystem and disk controllers. Careful implementation of write barriers and their correct application, which partially restrict the elevator seek optimizations, is indispensable for avoiding this problem. However, strict write ordering degrades storage performance because it increases disk seeks. Journaling file systems degrade performance due to the seeks between the journal file and original meta data blocks. At present, a few Linux journaling file systems support the write barrier, but it has not yet taken hold. Only XFS has recently made it effective by default. We seem to be faced with a trade-off between performance and reliability.

One interesting alternative approach is called log-structured file system (LFS)[6, 7]. LFS assures reliability by avoiding the overwriting of on-disk blocks; the modified blocks are appended to existing data blocks as is done for log data. This write method also improves performance since the sequential block writes minimize the number of disk head hops.

With regard to data recovery, we see the need for realizing more advanced features through the use of the technique called “snapshot”. Snapshots reduce the possibility of data loss including those caused by human error. Some commercial storage systems support this feature, but they are costly and are far from common. LFS suits data salvage and snapshots because past data are kept in disk. It can improve the restorability of data, and can compensate for operational errors.

Some LFS implementations appeared in the ’90s, but most of them are now obsolete. As for 4.4BSD and NetBSD, an implementation called BSD-LFS [8] is available. As for Linux, there was an experimental implementation called LinLogFS[5], but its development was abandoned and it’s not available for recent Linux kernels. This is primarily due to the difficulty of implementing LFS; LFS basically stores blocks at different positions at each write. Combining LFS with a modern block management technique such as the B-tree[4] is a significant challenge. To overcome this shortfall, we are developing NILFS, which is an abbreviation of the New Implementation of a Log-structured File System, for

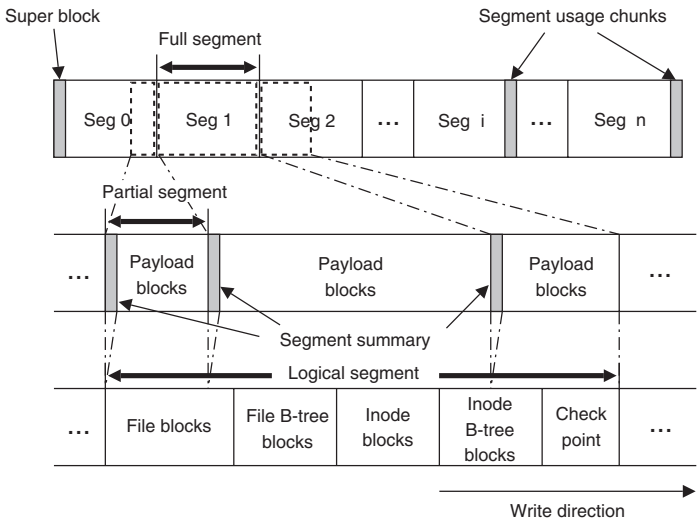

Figure 1: Disk layout of NILFS

the Linux operating system.

The rest of the paper is organized as follows. Section 2 overviews NILFS design. Section 3 focuses on the writing mechanism of NILFS, the key to our LFS implementation for Linux. After showing some evaluation results in Section 4, we conclude with a brief summary.

2. OVERVIEW OF THE NILFS DESIGN

2.1 Development goals

The development goals of NILFS are as follows:

1. High availability
2. Adequate reliability and performance
3. Support of online snapshots
4. High scalability
5. Compliance with Linux semantics
6. Improved operability with user-friendly tools

High availability means that the recovery time should match those of existing journaling file systems. For this purpose, a DB-like technique called checkpointing is applied. However, NILFS recovery is safer because it prevents the overwriting of meta data. In contrast, journaling file systems restore important blocks from the journal file during recovery. This can cause fatal collapse if the journal file is not written perfectly.

NILFS offers online snapshot support and adequate reliability and performance by taking advantage of LFS. Online snapshot support allows users to clip the consistent file system state without stopping services and then helps with online backup. The requirement of high scalability includes support of many files, and large files and disks with minimum performance degradation. We would like to implement NILFS without losing compliance with Linux file system semantics and to minimize changes in the kernel code. We note that the development of user tools is a future task.

2.2 On-disk representation

On-disk representation of inode number, block number and other sizes are designed around 64-bits in order to eliminate scalability limits. Data blocks and inode blocks are managed using B-trees to enable fast lookup. The B-tree structures of NILFS also have the role of translating logical block offset addresses into disk block numbers. We call the former file block number and the latter disk block number.

Figure 1 depicts the disk layout of NILFS. Disk blocks of NILFS are divided into units of segments. The term segment is used in multiple ways. To avoid ambiguity, we introduce three terms:

- Full segment:
  Division and allocation units. The full segment is basically divided equally and is addressable by its index value.
- Partial segment:
  Write units. Each partial segment consists of segment summary blocks and payload blocks. It cannot cross over the full segment boundary. The segment summary has information about how the partial segment is organized. It includes a breakdown of each block, length of the segment, a pointer to the summary block of the previous partial segment, and so on.
- Logical segment:
  Recovery units. Each logical segment consists of one or more partial segments. It represents the difference between two consistent states of a file system. The partial segments composing a logical segment are regarded, logically, as one segment.

A logical segment consists of file blocks, file B-tree node

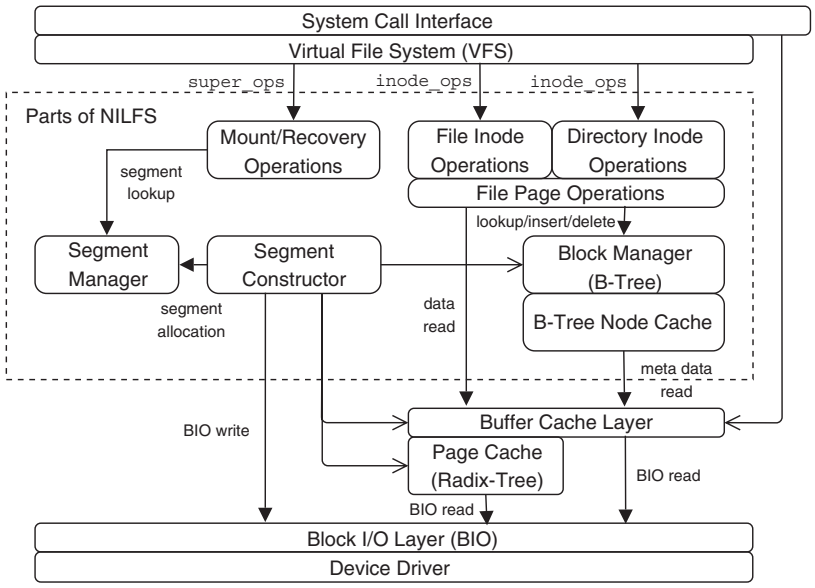

Figure 2: Block diagram of NILFS

blocks, inode blocks, inode B-tree node blocks, and a check point block. The B-tree node blocks are intermediate blocks composing B-tree structures. The check point block points to the root of the inode B-tree and holds inode blocks on the leaves. Each inode block includes multiple inodes, each of which points to the root of a file B-tree. File data and directory data are held by the file B-trees. These payload blocks are a collection of modified or newly created blocks. Since pointers to unchanged blocks are reused, the B-tree node blocks or unchanged inodes may have pointers to blocks in past segments. Thus, the check point block represents the root of the entire file system at the time the logical segment was created. Snapshots are realized as the ability to keep the on-disk blocks trackable from all or selected check points. The segment summary and the check point use cyclic redundancy checksums (CRCs) to assure validity of themselves and the payload blocks.

The full segments are managed using two more special types of blocks, the super block and the segment usage chunk. The super block holds basic information of the file system, disk layout parameters, and a pointer to the latest partial segment having a valid check point. The segment usage chunk includes the information needed to allocate the full segments. It also includes bidirectional links to give a logical sequence among the full segments. These blocks are redundantly stored between full segments.

2.3 Implementation architecture on Linux

Figure 2 shows the block diagram of NILFS. The upper part of NILFS is realized by implementing the object-orient interface of the Linux Virtual File System (VFS). It is composed of mount operations, file inode operations, and directory inode operations. The mount operations read the super block and build on-memory data structures representing a file system instance. During unmounting, it cleans up all resources of the file system instance. Recovery is executed as a part of the mount operations. It is realized by locating the valid checkpoint over partial segments. This search starts from the partial segment pointed to by the super block. The file inode operations read, write, delete, and truncate files through the manipulation of file inodes and file data blocks. The directory inode operations perform lookup, listing, creation, removal, and renaming of nodes on directories through the manipulation of directory inodes and directory data blocks.

The lower part of NILFS is entirely new and is composed of a block manager, a segment constructor, and a segment manager. These parts call generic functions of the Linux file cache layer or directly call the Linux block I/O (BIO) Layer. The Linux file cache layer consists of the buffer cache and the page cache. The buffer cache is a legacy interface, and recent kernels basically utilize page caching. The block manager uses its own cache for B-tree node blocks, called the B-tree node cache, which we describe later. The block manager is realized as the B-tree function itself. It gives the functions required to manage blocks such as lookup, insert and delete for both the inode blocks and the file blocks. The segment constructor collects newly created blocks and modified blocks and builds the partial segments needed to hold them. These on-memory blocks to be written to disk, called dirty blocks, are bulk written to the partial segments via the Linux BIO layer.

The segment manager serves as a full segment allocator and a cleaner. The cleaner is one of the essential LFS functions. It identifies unwanted blocks and makes continuous empty regions. This process is called garbage collection or cleaning. It cleans out segments by copying in-use blocks and their referrers. Without snapshot, it is conceptually simple as just described. However, for LFS with many valid snapshots, it is hard to implement this procedure efficiently because many blocks and links are involved. The cleaner is currently under development.

3. IMPLEMENTING WRITE MECHANISM

This section describes details of how we realized the segment constructor on Linux. Although the Linux kernel is designed to efficiently share common functions for simplifying the implementation of a number of different file systems, they are not strongly supportive of LFS:

1. Static placement of blocks:
   Because LFS changes the on-disk location of both data blocks and meta data blocks when writing them out, it needs to modify their block addresses. For the meta data blocks, a reindexing on the page cache is also required because they are indexed by absolute address over a block device [1]. We be careful of timing concerns and the side-effects of these operations.

   Managing newly allocated blocks is another topic to be considered. The position of these blocks cannot be decided until just before writing them. Consequently, their locations are transiently indeterminable and cannot be managed via the block address.

2. Granularity of writing:
   The typical Linux file system writes data blocks per page, and meta data blocks per block. Even though the page cache and the BIO layer have the ability to write multiple pages at a time, they are written separately with each file and with each block type. Such fine grained control is convenient for the file systems since they use the write order among various blocks, but not for LFS, because the atomicity of LFS writing is per segment not per page.

3. Differences in the treatment of meta data and data:
   Data blocks are mainly treated by shared code, whereas meta data are handled differently by each file system. Way of writing, synchronizing, locking, and error handling, is different between data and meta data. Even though LFS can write both types of blocks in the same way, we had to distinguish them to adapt to the shared kernel code.

To allow meta data blocks to change block address, we have introduced our own meta data cache. The cache is realized by extending the standard one to enable holding blocks with no fixed block address. For managing data blocks, we have adopted the standard page cache because these blocks were indexed with file offset and then it was reusable.

The segment constructor carries out both background writes and foreground writes. The foreground writes are requested by synchronous write operations of the VFS layer. The segment constructor consolidates write timing and provides a unified mechanism to write out the blocks to be written. This aggregation is natural since it supports the segment-based atomicity of writing, exclusion controls, and so on. It also provides straightforward implementation through direct manipulation of the BIO layer.

[1]Strictly speaking, the indexing is realized per page; multiple blocks may be held within a single page.

3.1 Procedure of segment construction

Since the B-tree node block of NILFS points to child blocks with disk block numbers, the pointers must be changed when the location of a child block changes. This change makes the B-tree node block dirty, and makes it the block to be written out. It also changes the pointer in its parent block. Thus, the dirty state propagates through all ancestors up to the checkpoint block.

Because B-tree construction may dramatically change due to insert and delete operations, the current NILFS version delays this propagation and performs it during segment construction. This also prevents the dirty B-tree node blocks from occupying memory for long periods.

To fix the number of dirty blocks to be written in the next partial segment before renumbering the disk block number, NILFS separates dirty state propagation from renumbering; segment construction is realized as follows:

1. Collecting dirty blocks.
2. Propagating dirty state upstream.
3. Getting a complete set of dirty blocks and building a segment.
4. Renumbering the absolute address of each block, replacing the pointers to the block, and moving it in the page cache.
5. Calculating CRC and finalizing both the segment summary and the checkpoint.
6. Submitting write requests using BIO and waiting for completion.

Figure 3 depicts the workflow that we employ to realize these procedures. Instead of making a whole image of the logical segment, we build, one-by-one, the partial segments composing it. For this we employ a buffer (segment buffer) whose capacity is just one full segment. This on-the-fly approach simplifies the handling of the long logical segments that cross over multiple full segments.

First, a partial segment is allocated from a current full segment. If no space is left in the current segment, the segment constructor calls the segment manager to allocate a new full segment. It then carries out procedures 1-3 simultaneously. This is shown as the collection phase in Figure 3. In the collection phase, one or more segment summary blocks and payload blocks, are registered with the segment buffer. If the number of registered blocks reaches capacity of the allocated partial segment, or the collection for one logical segment completes, then procedures 4-6 are applied for blocks of the partial segment using the segment buffer. The collection phase is designed to be suspendable and continuable. It is repeatedly applied until the construction of the logical segment is completed.

The collection phase has five stages which are executed sequentially. The FBLK stage collects dirty file blocks through gang lookup of the page cache. It also tracks back the B-tree file node blocks and propagates the dirty flag upstream.

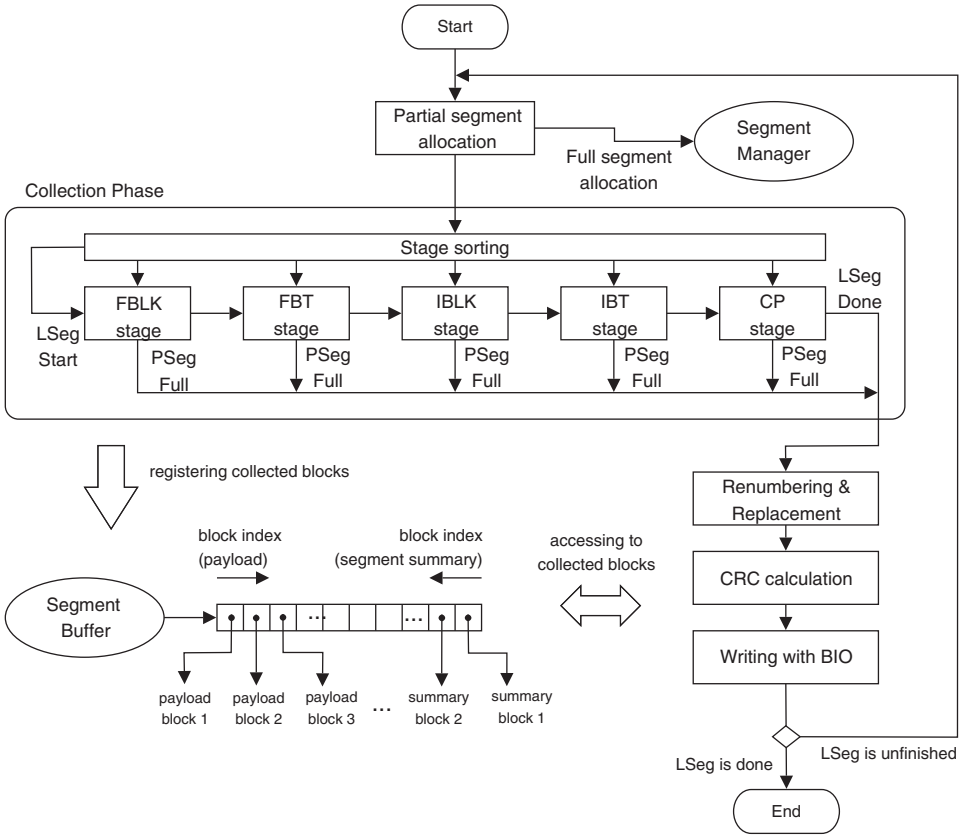

Figure 3: Main workflow of the segment construction

The FBT stage propagates the dirty flag from the B-tree file node blocks and collects all dirty ones. The IBLK stage and the IBT stage do the same thing to the inode blocks and the inode B-tree node blocks, respectively. The CP stage adds a checkpoint block to the segment buffer. The summary block is extended and appended along with those stages. Because the number of summary blocks depends on the configuration of the segment, the segment buffer fills them backward from the tail end.

3.2 Techniques to mitigate construction overhead

Some techniques can be applied to reduce the load created by segment construction. First, we prepare a list chaining the inodes whose files are regarded as dirty. The list, called the dirty file list, allows the FBLK stage and the FBT stage to avoid exhaustive lookup over the page cache. The inode of dirty file is inserted in the list when pages of the file are committed to write. The inode is removed every time the logical segment is completed.

Avoiding unnecessary construction is also effective because it cuts down on the number of high cost disk writes. Early cancellation, in addition, mitigates overhead due to the lookup operations over the page cache and B-trees. The segment constructor of NILFS reconfirms the need for construction on a few occasions, by using the dirty file list and the flag that indicates whether the B-tree is modified or not.

With regard to block reindexing for the meta data cache, architecture dependent optimization was used. Since each page may have multiple blocks, the B-tree node blocks have to be separated and to be copied from the original page when their disk block number changes. This copying can be replaced by moving if the block size equals the page size. Although this brings in some asymmetry, it works with most common 32-bit PCs.

4. EVALUATION

We compared NILFS to ext3 using the "iozone" benchmark program. The measurement computer had a Pentium 4 3.0GHz CPU, 1-Gbyte of memory, and 7,200rpm IDE hard drives with 8-Mbyte disk cache and Ultra ATA connection. The Linux kernel version was 2.6.13. Both the disk block and page have sizes of 4 Kbytes.

Figure 4 shows the measurement results for writing. The journaling mode of ext3 is the default ordered mode. Because current NILFS has no cleaner and is not tuned at all, these are just preliminary values. To compare actual disk write performance, write synchronization was forced in two ways. The left graph shows write throughput synchronized by the fsync() system call, whereas the right graph shows that synchronized with the O_SYNC open system call option. The horizontal axis gives the write buffer size. In the random write measurements, an existing file is rewritten. In the sequential write measurements, new files are created. NILFS shows higher write throughput in both random write

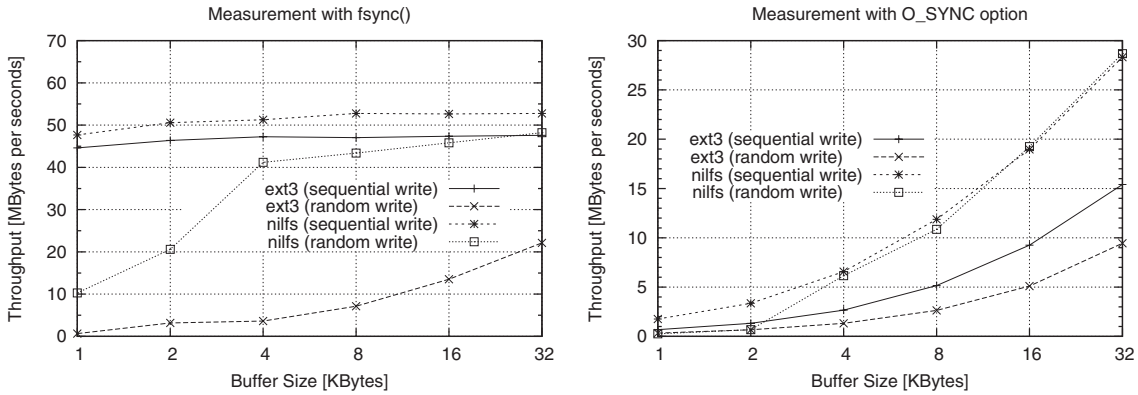

Figure 4: Comparison of write performance

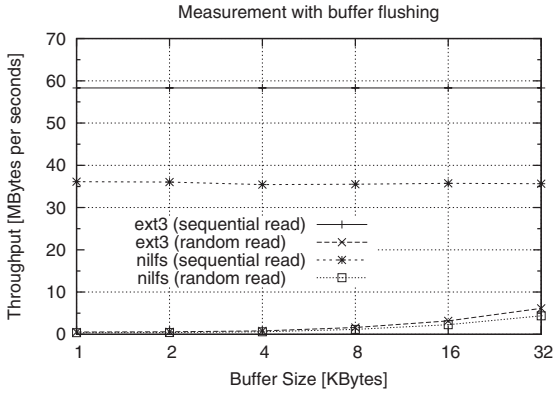

Figure 5: Comparison of read performance

and sequential write. The superiority of NILFS is particularly noticeable in the random write. These results are due to the smaller number of disk seeks.

On the other hand, the read performance of the current NILFS version is unsatisfactory as shown in Figure 5. To get real read performance, caches are flushed through unmount and mount operations. These operations can flush both the on-memory cache and the disk cache. Under the condition where the caches are flushed in advance, it achieves only 61% or so of throughput of ext3 for sequential read, and 72% for random read. In this regard, however, LFS assumes the existence of the disk cache. If the disk cache is enabled, both file systems show approximately the same throughput because many blocks are read from memory. Anyway, we recognize that read performance tuning is a priority target.

5. CONCLUDING REMARKS

We described the implementation of NILFS focussing in the LFS write mechanism. NILFS is distributed under GPL through the NILFS homepage (http://www.nilfs.org/). NILFS is a reimplementation of LFS and adopts B-tree based block management and supports snapshots. It offers an LFS alternative to the Linux operating system. Although it does not have an essential portion, the so-called cleaner, we are expecting that it will satisfy the very high level of reliability demanded for modern open-source operating systems.

6. REFERENCES

[1] JFS for Linux. http://jfs.sourceforge.net/.

[2] Project XFS Linux. http://oss.sgi.com/projects/xfs/.

[3] ReiserFS. http://www.namesys.com/.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[5] C. Czezatke and M. A. Ertl. LinLogFS — a log-structured filesystem for Linux. *Freenix Track of Usenix Annual Technical Conference*, pages 77–88, 2000.

[6] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *ACM SIGOPS Operating Systems Review*, 23(1):11–28, 1989.

[7] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[8] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. *USENIX Winter*, pages 307–326, 1993.

[9] S. C. Tweedie. Journaling the Linux ext2fs filesystem. *LinuxExpo '98*, 1998.