the Nilfs version 1: overview

Nilfs team
NTT Cyber Space Laboratories
NTT Corporation
`http://www.osrg.net/nilfs/`
`nilfs@osrg.net`

1 Introduction

To enhance reliability of the Linux file system, we adopted a Log Structured File System (LFS)[1]. In the past, the advantages of LFS were high write performance and faster recovery time. In addition, LFS prevents data write operations from overwriting the disk blocks, thereby minimizing damage to the file data and file system consistency and avoiding a hardware failure. Furthermore, LFS writes data and meta-data in the correct order, which helps ensure file system consistency. The position of the writing blocks are evenly distributed around the disk, thus reducing the possibility of a disk hardware failure. LFS can create a snapshot of the file system immediately and makes the system user friendly.

Presently, we are developing the Nilfs (the New Implementation of a Log-structured File System) as a kernel module in Linux 2.6.

2 the Goal of the Nilfs

The goals of the Nilfs are as follows.

1. Respect Linux semantics
2. Avoiding modifications to the kernel code
   This simplifies Nilfs installation and following up a frequent version rise of the Linux kernel speedy.
3. Getting high reliability, availability and operability
   For example, the file systems are reliable, like DBMS, and immediately takes its snapshot.
4. High performance and scalability
5. Adapting the distributed file system or the cluster system
6. Being user friendly
   It reduces the possibility of human errors.

We also included our future work in this development goal. From these, we set guaranteed high reliability and availability as our main goals.

3 the Log Structured File System

The LFS writes all modifications to the disk sequentially, no blocks are overwritten, and log-like structures are appended to the disk instead. Meta-data, such as file inodes and indirect blocks are written to newly allocated blocks. In LFS, data writing does not break any existing disk blocks on system failure such as power down. A new disk block address is assigned to the data on the memory buffer cache during write operations. The assignments are in the following order; file data, file data indirect blocks, inode blocks, and inode indirect blocks. This order is important for maintaining the consistency of the file system structure. Write operations are also performed in this order, without having to seek the disk head.

The write position is contiguous and stable, and the file system check (fsck) process only inspects a small area of the disk.

Because disk capacity is limited, we need a “garbage collection” service (or cleaner) to collect deleted file blocks and logically overwritten blocks. Garbage collection is a major overhead of LFS. However, the garbage collector can efficiently restore fragmented file blocks. For efficient garbage collection, whole disk is divided into fixed sizes (ex. 4 mega bytes). This management unit is called a full segment. Writing out is done sequentially in full segments.

4 Nilfs Disk Layout

In Nilfs, our design goals are to obtain high reliability and availability of the file system. We have not yet begun performance tuning. However, to be able to use the Nilfs in the future, the file size and inode numbers are stored in 64-bit-wide fields, and file blocks are managed by a B-tree[2][3]. The root of the file block B-tree is placed on the inode structure. The inode is managed by the inode block B-tree, the root of the inode block B-tree is stored in the superblock structure of the file system.

In this section, we will first show the layout of the whole disk, and then explain the management structure of file blocks.

4.1 Disk Layout

The disk layout of Nilfs is shown in Figure 1, divided into several parts.

superblock Superblock has the parameters of the file system, the disk block address of the latest segment

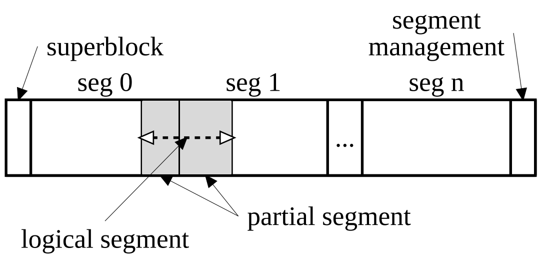

Figure 1: disk layout of the Nilfs

being written, etc. The inode block B-tree root is stored on the latest segment, so it represents a current valid user data tree. Superblock information is important, and it is replicated on another block of the disk.

full segment Each full segment consists of a fixed length of disk blocks. This is a basic management unit of the garbage collector.

partial segment Write units. Dirty buffers are written out as partial segments. The partial segment does not exceed the full segment boundaries, as shown in detail in section 4.3.

logical segment The partial segment sequence includes inseparable directory operations. For example, shadowed part of Figure 1 is a logical segment that consists of two partial segments. In the recovery operations, the two partial segments are treated as one inseparable segment.

There are two flag bits, Logical_Begin and Logical_End, at the segment summary of the partial segment.

segment management block The block is comprised of 50 chunks for storing segment usage information, which is used by the garbage collector and segment allocator. The position of the block is fixed, so there is one replica per chunk.

There are two important fields per full segment in segment management block, “logically previous full segment” and “logically next full segment.” These fields are filled by newfs tool and garbage collector. A full segment allocation routines simply follow the “next” field. The file system recovery routines (or tools such as fsck) can determine the sequence of full segment correctly.

When the Nilfs file system is mounted, all chunks are read for the block allocator. It takes about one second to read one hundred chunks, distributed over the whole disk (10-msec for one head seek operation).

The superblock and the segment management blocks are overwritten in Nilfs.

4.2 Disk Block Management

In the local file system, there are three kinds of mappings. First, the file block address is mapped to the disk block address. In traditional file systems, it is managed by an index array in the inode structure and by indirect blocks. Second, the inode numbers are mapped to the disk block address containing the inode structures. They placed in the permanently allocated disk blocks in the traditional file systems. Third, the file path name is mapped to the inode number of the file. This is known as the directory.

The Nilfs adopts the B-tree structure for both file block mapping and inode block mapping. The two mappings are implemented in the common B-tree operation routine. It simplifies coding, is efficient for managing big files, and keeps inode management flexible. For example, Nilfs has no inode usage bitmap. When we need an unused inode number, call the internal B-tree routine. The routine looks up unused inode number as candidate of new inode number.

There are some variations of B-tree, such as B$^+$-tree and B*-tree. The B$^+$-tree is designed for the sequential access acceleration, while B*-tree is designed for efficiently using memory space. However, Nilfs adopts a basic B-tree. The reason is as follows.

The B$^+$-Tree is suitable for file block management. File blocks are often accessed sequentially. Each leaf of the B$^+$-Tree has a pointer to another leaf on which the address continues. On LFS, when a modified block is moved to a new address, the pointer indicates the block that must be rewritten. The rewritten block is also moved to a new address, and the block pointing to the moved block must also be rewritten. So, one modified block results in all tree blocks being rewritten. Therefore, this operation must not be done in the LFS. Sequential access acceleration of the file can be achieved using the read ahead technique.

The B-tree intermediate node is used to construct the B-tree. It has 64-bit-wide key and 64-bit-wide pointer pairs. The file block B-tree uses a file block address as its key, whereas the inode block B-tree uses an inode number as its key. The root block number of the file block B-tree is stored to the corresponding inode block. The root block number of the inode block B-tree is stored to the superblock of the file system. So, there is only one inode block B-tree in the file system.

Presently, a file block B-tree is constructed for a small (even 1 block) file. Improving the space efficiency for small file is a future work.

File blocks, B-tree blocks for file block management, inode blocks, and B-tree blocks for inode management are written to the disk as logs.

A newly created file first exists only in the memory page cache. Because the file must be accessible before

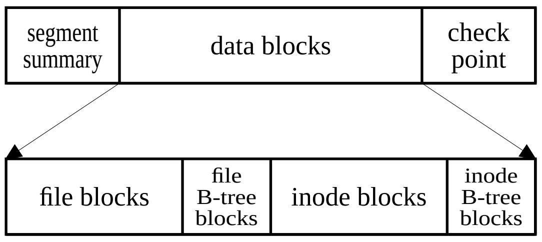

Figure 2: layout of the partial segment

being written to the disk, the B-tree structure exists even in memory. The B-tree intermediate node in memory is on the memory page cache, the data structures are the same as those of the disk blocks. The pointer of the B-tree node stored in memory holds the disk block number or the memory address of the page cache that reads the block (distinguished by MSB of the memory word). When looking up a block in the B-tree, if the pointer of the B-tree node is a disk block number, the disk block is read into a newly allocated page cache before the pointer is rewritten. The original disk block number remains in the buffer-head structure on the page cache.

We plan to adopt the B-tree structure to construct a directory that will keep a mapping from the path name to the inode number. However, B-trees that use variable length names as the key are much complicated and have not yet been implemented. The Nilfs directory is ported from the traditional ext2 file system. So, large directory operation performance is not good.

4.3 Layout of the Partial Segment

In this section, we describe the details of the partial segment. Figure 2 shows the layout of a Nilfs partial segment. The partial segment consists of three parts.

segment summary The segment summary keeps the block usage information of the partial segment. The main contents are checksums of the data area, the segment summary, the length of the partial segment, and partial segment creation time. On the usual data access, the segment summary is not referred. It is needed by the garbage collector and file system recovery process.

data area There are file data blocks, file data B-tree node blocks, inode blocks, and inode block B-tree node blocks in order.

check point A checkpoint is placed on the last tail of the partial segment. The checkpoint includes a checksum of the checkpoint itself. The checkpoint accuracy means successfully writing the partial segment to the disk. The most important information in the checkpoint is the root block number of the inode block B-tree. The block number is written out last, and the whole file system state is updated.

4.4 Guaranteeing Reliability

To guarantee file system reliability, which is the goal of Nilfs, we actualized the following functions.

checksum The Nilfs stores a checksum as follows. The segment summary maintains the checksum of the data blocks and the segment summary. The superblock and checkpoint keep their own checksums. The segment management block keeps its own checksum. And finally, all checksums are calculated using the CRC32 method. There are no checksums for individual data block.

The Nilfs newfs tool generates one 32-bit random number and stores it in the superblock. Whenever calculating checksums, the stored number is used as the CRC32 initial value. If there is an old data on previously newfsed Nilfs file system, then its checksum will not match because it is calculated it from another initial value. The Nilfs recovery process can find current version of checksummed data using the random initial value.

retention of write order The Nilfs keeps writing in the correct order. First, file data blocks are written to the disk; next, the file data B-tree node blocks; and then the inode blocks; and finally, the inode block B-tree node blocks last. These block disk addresses are assigned in order, so, the correct write order and high write performance are both archived.

minimize overwriting The superblock, its replica, segment usage chunks, and their replicas are overwritten to update information. No other blocks are overwritten using data modification.

These three functions guarantee the reliability of the Nilfs file system.

5 Nilfs Implementation

5.1 Nilfs Architecture

Figure 3 shows the architecture of Nilfs. Rounded box parts are implemented as Nilfs.

Mount/recovery operations call a buffer management module (line (1) in Figure 3) of Linux Kernel 2.6 to read the superblock and segment summary that wrote last mounted time. File page operations use the Nilfs's block management module (2) to lookup/insert/delete appropriate disk blocks via the Nilfs B-tree operations. Normal file read operations execute by the file page operations module using buffer management module directly (3). When amount of dirty pages are exceeded an internal limits, a segment construction module is triggered (4) to start a segment construction. The segment construction module calls the buffer management module

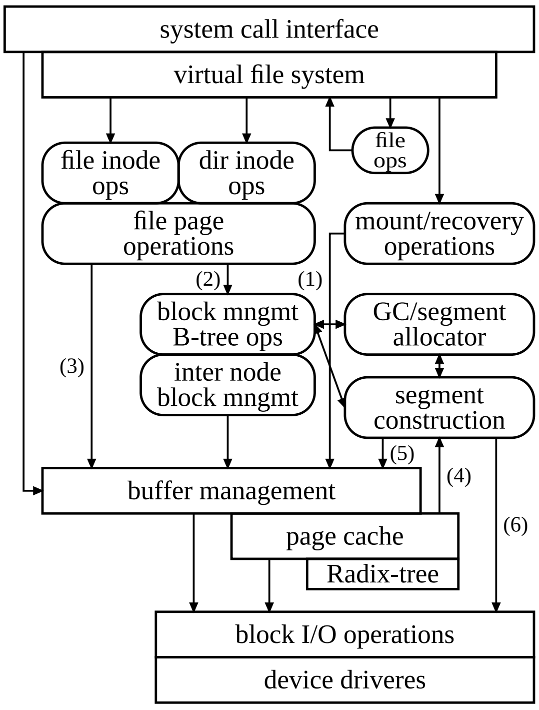

Figure 3: architecture of the Nilfs

(5) to arrange the dirty pages, and call block I/O operations (6) for writing out the constructed segments.

Linux Kernel parts (square box) are not modified to implement the Nilfs.

5.2 File Buffer Management in Kernel 2.6

Reading from or writing to the usual file is done using page cache. When reading a block from a file, it is necessary to decide whether the block has already been read to the page cache. When modifying a file, the file block is first read to page cache. To write data to the new portion of a file, a page cache is prepared first, then the data is written to the page cache, and the page cache is written to the disk later. The indirect blocks of the file also use page cache, because they need to determine whether the block exists on page cache or not, for all file read and write operations. This must be done efficiently to maintain read/write performance. Therefore, Linux 2.6 adopts the radix-tree structure to keep the page cache that read the file block. The root of the radix-tree is stored in the corresponding memory inode structure, the tree uses the file logical block number as its index key. On the other hand, the indirect file blocks have no logical block numbers - only the physical disk block number. So, Linux 2.6 keeps the page caches of the indirect blocks in the radix tree, whose root is in an inode of the block device (bd_inode). All page caches that keep the indirect block data of all files in the file system are registered to the radix-tree.

5.3 Nilfs Data Structure in Memory

We decided not to modify the Linux kernel, block device input/output, page cache management, or system call execution.

With Nilfs, the newly created B-tree intermediate node block has no disk block number, so bd_inode's radix tree does not retain the corresponding page cache. Therefore, we used the buffer-head memory address of the page cache as the radix-tree's key. The radix-tree for the B-tree's intermediate node is stored to the memory of the corresponding inode structure. Consequently, the Nilfs radix tree is smaller than the Linux radix tree.

5.4 Segment Construction

The data write process started by the sync system call and Nilfs kernel thread, advances in the following order.

1. Lock the directory operations
2. The dirty pages of the file data are gathered from its radix-tree.
3. The dirty B-tree intermediate node pages of both file block management and inode management are gathered.
4. The dirty inode block pages are gathered.
5. The B-tree intermediate node pages which will be dirty for registered block address being renew are gathered. See Section **??**
6. New disk block addresses are assigned to those blocks in order of file data blocks, B-tree node blocks for file data, inode blocks, B-tree node blocks for inodes.
7. Rewrite the disk block addresses to new ones in the radix-tree and B-tree nodes.
8. Call block device input/output routine to writing out the blocks
9. Unlock the directory operations

In order to comply with POSIX semantics, directory operations must be exclusively executed. The segment construction operation relates to the whole file system, the directory operations are locked during the construction operation.

5.5 Snapshot

The Nilfs snapshot is a whole consistent file system at some time instant. The snapshot is very useful for making a backup of the file system, and being rolled back to previous state after modifying many files in the file system. In LFS, all blocks remain as is (until they are collected by garbage collection), therefore, no new information is needed to make a snapshot. In Nilfs, the B-tree structure manages the file and inode blocks, and B-tree nodes are written out as a log too. So, the root block number of the inode management B-tree is the snapshot of the Nilfs file system. The root block number is stored in the checkpoint position of a partial segment. The Nilfs checkpoint is the snapshot of the file system itself. Actually, user can specify the disk block address of the Nilfs checkpoint to Linux using the “mount” command, and the captured file system is mounted as a read-only file system.

However, when the user use all checkpoints as the snapshot, there is no disk space for garbage collection. The user can select any checkpoint as a snapshot, and the garbage collector collects other checkpoint blocks. The user does not need any commands “before” taking a snapshot.

6 Conclusion

We described overview of the design and implementation of “Nilfs” and implemented LFS using modern technology to get a high performance file system.

References

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

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

[3] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–138, 1979.