Charles University in Prague

Faculty of Mathematics and Physics

DOCTORAL THESIS

Mikuláš Patočka

Design and Implementation of the Spad Filesystem

Department of Software Engineering

Advisor: RNDr. Filip Zavoral, Ph.D.

Abstract

Title:	Design and Implementation of the Spad Filesystem
Author:	Mgr. Mikuláš Patočka email: mikulas@artax.karlin.mff.cuni.cz
Department:	Department of Software Engineering Faculty of Mathematics and Physics Charles University in Prague, Czech Republic
Advisor:	RNDr. Filip Zavoral, Ph.D. email: Filip.Zavoral@mff.cuni.cz
Mailing address (author):	Mikuláš Patočka Voskovcova 37 152 00 Prague, Czech Republic
Mailing address (advisor):	Dept. of Software Engineering Charles University in Prague Malostranské nám. 25 118 00 Prague, Czech Republic
WWW:	`http://artax.karlin.mff.cuni.cz/~mikulas/spadfs/`

Abstract:

This thesis describes design and implementation of the Spad filesystem. I present my novel method for maintaining filesystem consistency — crash counts. I describe architecture of other filesystems and present my own design decisions in directory management, file allocation information, free space management, block allocation strategy and filesystem checking algorithm. I experimentally evaluate performance of the filesystem. I evaluate performance of the same filesystem on two different operating systems, enabling the reader to make a conclusion on how much the performance of various tasks is affected by operating system and how much by physical layout of data on disk.

Keywords: filesystem, operating system, crash counts, extendible hashing, SpadFS

Acknowledgments

I would like to thank my advisor Filip Zavoral for supporting my work and for reading and making comments on this thesis.

I would also like to thank to colleague Leo Galamboš for testing my filesystem on his search engine with 1TB RAID array, which led to fixing some bugs and improving performance.

The work was partly supported by the project 1ET100300419 of the Program Information Society of the Thematic Program II of the National Research Program of the Czech Republic.

Content

1. Introduction

Filesystems are an important component of operating systems. The quality of filesystem implementation designates the overall performance of an operating system and affects the user's experience. Filesystems are usually implemented in the kernel of the operating system [1].

The goal of this thesis was to design a filesystem that has good performance, good reliability and that has small codebase and is relatively simple to implement.

To facilitate these requirements — performance, safety and simplicity — I selected the simplest and fastest algorithms suitable for use in filesystems and I implemented them in my Spad filesystem and experimentally evaluated performance. This thesis describes my design decisions as well as the algorithms used in other deployed filesystems.

The goal of my work is to design filesystem data structures and to develop two implementations — one on Linux 2.6 and the other for experimental kernel Spad. I evaluate the performance of both implementations and other popular Linux filesystems and I show how much the performance is affected by filesystem and how much by the underlying operating system.

1.1 The structure of this thesis

My thesis has the following structure:

In the rest of section one I introduce basic filesystem concepts. I also present other popular filesystems used nowadays. I describe their history and show brief list of features. The features are discussed in depth in sections 3 to 7.

In section two I outline basic design goals of my work.

In section three I describe my novel method for maintaining filesystem consistency across system crashes — crash counts — and discuss methods in other filesystems: journaling, log-structured filesystem, phase tree and soft updates.

In section four I describe the structure of directories in other filesystems and in the Spad filesystem. I show different indexing methods and placement of directory information.

In section five I describe methods for storing file allocation information in other filesystems and in the Spad filesystem. Various methods for mapping file blocks to physical disk blocks are shown.

In section six I describe my method for managing free space in other filesystems and in the Spad filesystem.

In section seven I present block allocation strategy in Ext2, FFS and in the Spad filesystem and discuss each. This strategy decides where on disk files should be allocated in order to minimize seeks.

In section eight I outline goals of filesystem-checking utility and describe the checking algorithm for Ext2/Ext3 and the Spad filesystem.

In section nine I show a summary of features of evaluated filesystems.

In section ten I present the experimental performance evaluation of the Spad filesystem and other filesystems.

1.2 General filesystem architecture

A filesystem is a part of an operating system that stores data on storage devices (usually disks). The storage device consists of sectors of fixed size. The purpose of a filesystem is to create some structure on the device and store and retrieve data.

A filesystem stores information in files. Today, a file is standardized as a variable-length stream of 8-bit bytes. In the past there were some other approaches, such as using machine word size as a unit of the file [2]. Some operating systems (such as OpenVMS) have files consisting of records [68] but this approach hasn't become mainstream — the main problem with processing records at operating-system level is that it is hard to transfer files between different operating systems (various encodings are needed) and almost impossible to read or modify the file content on an operating system other than where it was created.

On Windows NTFS and MacOS(X) HFS the file may consist of multiple byte streams — one default stream and several named streams. Multiple streams are rarely used and their use is discouraged because the streams will be lost if the file is copied to a filesystem not supporting the streams or transferred over a network with a protocol that doesn't support multi-stream files. Transferring such files over a network requires the user to encode and decode them with a special utility, for example BinHex on MacOS.

Files are organized in directories. In early multi-user operating systems, filesystems usually had one directory per user, with files identified by file name with fixed size. Hierarchical directories were invented in Multics operating system [69] — each directory can contain files and other directories, so the directories for a tree. This concept of hierarchical directories has been used since then.

Besides files and directories, some filesystems store other objects:

- **Device nodes** — a device node represents a device in the operating system. When some program opens the device node, the filesystem does a name lookup, finds that the file is a device node and sets up the file descriptor in such a way that further operations on the file descriptor are passed to the appropriate device driver. The purpose of device nodes is to allow userspace programs that were intended to operate on files to work with other system devices too.
- **Hard links** — on some filesystems (mostly in Unix operating systems), there may be more directory entries pointing to one file. They are called hard links. The file may be opened with any of the hard links and the user sees the same content. The file contains a count of hard links pointing to it and it is physically deleted when the count reaches zero (i.e. when the last hard link is deleted).
- **Symlinks** — a symlink is a file that contains a path to another file. When some program attempts to open a symlink, the file it points to is opened instead.

  Hard links and symlinks differ in the following ways:
  - When you create a hardlink, the original file and its hardlink are indistinguishable. When you create a symlink, the symlink differs (it contains 'l' character in the file list).

  - When you delete a hardlinked file, the real data on disk is released when you delete the last link to that file. When you delete a file with symlinks, the file is deleted regardless of the symlinks and the symlinks point to a non-existing file.
  - Symlinks can point to files on a different volume. Hardlinks can be created only to files on the same volume.
  - Symlinks can point to directories. Hardlinks to directories are forbidden.
- **Reparse points** — reparse points exist on Windows 2000, XP and Vista on NTFS filesystem. Reparse points can be viewed as generalized symlinks; the destination path is not stored on the filesystem but provided by another kernel driver. When a reparse point is encountered, the kernel asks loaded filesystem filter drivers if some of them can process the reparse point. If some driver can process the reparse point, the driver returns a new path where directory search should continue. Reparse points only apply to directories; it is not possible to create a reparse point for a file. Reparse points can be used to implement:
  - Unix-like symlinks to directories.
  - Unix-like mount points (if assigning a driver letter to a volume is not desirable).
  - Hierarchical storage management — a method for storing rarely accessed files on slower and bigger storage devices.
- **Pipes** — Processes on Unix operating systems can create two types of pipes: Unnamed pipes are created with `pipe` syscall; these pipes exist only as long as file descriptors are open. Unnamed pipes do not interact with the filesystem at all. An alternative is to use named pipes; named pipes are created with `mknod` syscall. A named pipe exists as an object on the filesystem and any process trying to open the object will open the pipe.

  The filesystem contains only a flag indicating that the object is a pipe — the content of the pipe is not held on the filesystem, it is held in memory. Thus, the content is lost when all processes close the named pipe.
- **Unix domain sockets** — Unix domain sockets enable fast communication between processes on the same computer without the overhead of networking protocols (such as TCP/IP). They are used for local communication, for example between an X Window server and X Window clients or between a database server and database clients.

  Unix domain sockets are referenced by a name on the filesystem. The filesystem contains only a flag that the given object is a Unix domain socket; it doesn't contain any socket state information or data waiting in the socket queue.

  Unlike named pipes, unix domain sockets can't be opened and accessed like normal files. They are accessed with network syscalls, `socket`, `listen`, `accept`, `connect` and others. The file name is embedded in `struct sockaddr_un` passed to some of these syscalls.

In the next subsections I briefly introduce key components of a filesystem.

1.2.1 File allocation subsystem

A file allocation subsystem maps logical blocks in a file to physical blocks on the disk. This subsystem processes requests to find a physical block for a given logical block or to extend the file or to shrink it. Some filesystems support sparse files (files, that don't have all blocks allocated on the disk), for these filesystems the file allocation subsystem also processes requests to add blocks to the middle of the file.

The FAT filesystem uses a simple linear list to describe blocks belonging to a file, whereas most other filesystems use some more advanced method (B-tree or radix tree) with $O(\log n)$ complexity.

Some filesystems store the mapping for every logical-to-physical block pair. Because files are mostly allocated in consecutive blocks, the storage can be compressed by storing triplets (*logical_block*, *physical_block*, *number_of_blocks*) — they are called extents [3].

1.2.2 Directory organization subsystem

The purpose of this subsystem is to store file names in directories. This subsystem processes requests to add, remove or lookup file names, or to list the whole directory. Simple filesystems use linear store with $O(n)$ complexity, advanced filesystems use some better method, usually B-trees, hashing or B-trees combined with hashing.

1.2.3 Free space management

This subsystem stores information about free space. It processes requests to allocate and free blocks (these requests are usually made by a directory organization subsystem and file allocation subsystem). Bitmaps, where each bit represents one block, are most often used to implement free space management. But there are some other approaches possible, such as using extents and placing them in B-tree or other structure.

1.2.4 Block allocation strategy

This subsystem makes decisions where new files and directories should be allocated in order to reduce long-time fragmentation of files and free space.

A block allocation strategy subsystem generates hints where structures should be allocated. Free space management then tries to find some free space near the hint and allocate the structure there.

1.2.5 Crash recovery subsystem

A crash recovery subsystem manages data consistency in the case of a crash. After a system crash, some blocks have been written and some not. Modern disks have writeback caches and can reorder writes in the cache, so when the filesystem submits several sectors for write, they may be written in any order. The purpose of the crash recovery subsystem is to bring the filesystem back to a consistent state. Journal is the most used method for this purpose.

A filesystem without crash recovery must run a filesystem-checking utility after a crash. This utility scans the whole disk and repairs the filesystem. This operation is slow; it can even take several hours on large partitions.

1.2.6 Transparent compression

Some filesystems support transparent compression and decompression of data. The compression or decompression is done by the filesystem driver when the application writes or reads data from the filesystem.

The filesystem must support random access to any file location, so classical stream compression algorithms cannot be used on the whole file. The compression is usually implemented by grouping several blocks and applying the stream compression algorithm on them. When the application wants to read from a given file location, the filesystem locates the compressed group of blocks pertaining to this area, and runs the decompression algorithm from the start of the area.

1.2.7 Security information

Filesystems contain information about the security of individual files, directories or other objects. The operating system checks this security information and verifies if the access by a specific user to a specific object is allowed. The format of the security information is dependent on the operating system. Unix operating systems store the security information as user ID, group ID and bit mask with access rights for read/write/execute for user, group and others.

A newer method (available on Windows NT, 2000 and later and on some newer Unix operating systems) is Access control lists — i.e. lists that allow different access rights to be specified for each individual user.

In most operating system architectures, the security is checked by an operating system and not by the filesystem driver and thus I will not cover security matters in this thesis. The job of a filesystem driver is to get the security information from the operating system kernel, store it, retrieve it when the file is accessed again and hand it over to the kernel. The filesystem driver doesn't do security checks on its own.

The security information stored on the filesystem is enforced only as long as the operating system kernel is running and is not compromised. If the user boots his own kernel (for example a live-CD distribution) he can completely bypass the security stored on filesystems on hard disk and access or modify any files. Similarly, if the user physically removes the disk and connects it to his own computer, he can access all the files without restrictions. Security information is not sufficient in these cases; encryption must be used to protect the data if the attacker can gain physical access to the computer.

1.2.8 Transparent encryption

Some filesystems support encryption of files. There are two general methods for disk encryption:

- Encrypting the whole disk — this can be done without any filesystem interaction at all and it can be done with any filesystem. The encryption layer is inserted between the filesystem and the block device driver. It has the advantage that everything, both metadata and data, is encrypted and an attacker can't find any information about the device. The disadvantage is that there is one master key to decrypt the whole device, and it is impossible to create different keys for different users.
- Encrypting individual files — this is supported by some filesystems. In this method, the filesystem encrypts some files and doesn't encrypt others. The advantage is that different keys for different users may exist. The disadvantage is that metadata are still accessible to an attacker — for example, an attacker can't see the content of an encrypted file, but he can see the size of it (in some implementations he can see also the file name).

1.3 Related work

In this thesis I compare algorithms in the Spad filesystem to other filesystems deployed in the industry. In this section I show an overview of related work. I discuss features of these filesystems more thoroughly in sections 3 – 7.

1.3.1 The FAT

The FAT filesystem (File Allocation Table) was created by Microsoft in 1977 [70] [4]. Originally it supported only floppy disks; its files could have only an 8-character long name and 3-character long extension and it didn't support subdirectories.

In 1983 in MS-DOS version 2, the FAT filesystem was extended for hard disks (FAT16) and support for subdirectories was added.

In 1993 in MS-DOS version 6, the possibility of transparent compression was added to the FAT filesystem.

In 1995 in Windows 95, long file name support was added to the FAT filesystem (VFAT).

In 1996 in Windows 95 OSR2, support for hard disks larger than 2GB was added to the FAT filesystem (FAT32).

The FAT filesystem uses a simple linear list for directory organization. It allocates space in fixed-size clusters and uses a file allocation table for maintaining file allocation information and free space information. For each cluster, the file allocation table contains the number of the next cluster in the file, end-of-file mark or a free cluster mark. The FAT filesystem doesn't have any method for crash recovery; in the case of a crash the whole filesystem is scanned when the operating system boots up.

The FAT filesystem is very simple; it doesn't have good performance and lacks many features, but it is still used today, primarily for portable media. Because of its simplicity, it is supported by most operating systems and it is easy to implement in embedded applications.

1.3.2 The HPFS

HPFS stands for "High Performance File System". It was designed by Microsoft and introduced in OS/2 version 1.2 in 1989 [5]. Previous versions of OS/2 worked only with the FAT filesystem.

The HPFS has some important advantages over the FAT: it supports file names of up to 255 characters, B-trees for directories and for file allocation information, extents for reduced storage space for file allocation information and bitmaps for reduced storage space for free space information [6].

Today, the HPFS is supported by OS/2 and by Linux [71] operating systems. It used to be supported by early versions of Windows NT (3.1, 3.5, unofficially 4.0).

1.3.3 The NTFS

The NTFS filesystem was introduced in Windows NT version 3.1 in 1993.

Similar to the HPFS, the NTFS has B-trees, extents and bitmaps, but their implementation is different. NTFS's advantages over the HPFS are:

- Fast recovery from system crashes using journaling.
- Storing master file table redundantly in two places, making the filesystem resistant to bad sectors on disks[1].
- NTFS supports transparent compression and encryption.
- Since Windows 2000, NTFS supports reparse points.

The NTFS is supported by Microsoft Windows NT operating systems (Windows NT 3.1, NT 3.5, NT 4.0, 2000, XP, 2003, Vista). The layout of the filesystem differs for different versions of Windows, making it unsuitable for portable media.

[1] File data is not stored redundantly, only the directories and information about files. Thus the user can still receive I/O errors when attempting to access files stored in bad sectors, but the directory tree shouldn't be damaged by disk errors.

Because of NTFS's complexity and closed source code, NTFS support in other operating systems is limited. Read-only NTFS support exists in Linux, BSD and OS/2 operating systems, knowledge was gained via reverse engineering [72]. Write support was non-existent or limited to overwriting existing files without changing their size.

Recently, drivers to write to the NTFS filesystem under Linux have emerged:

- Captive NTFS [73] is a wrapper that loads original Windows driver `NTFS.SYS` into Linux userspace process and emulates a subset of Windows kernel interface for it — allowing a user to read and write the NTFS filesystem in Linux.
- NTFS 3G [74] is a userspace implementation of the NTFS filesystem based on thorough reverse engineering. This implementation allows writing to the NTFS filesystem and is considered stable [75].

These implementations have performance limitations because they run in userspace. So far no one has presented an open source kernelspace driver that could write to the NTFS filesystem.

1.3.4 The FFS

The FFS [7] is a filesystem for the BSD family of operating systems. Its structure originates from the traditional Unix filesystem [2].

The FFS filesystem uses linear lists for directories and bitmaps for storing free space information. For storing file allocation information, FFS uses radix trees (with depth up to 3) containing all the file's physical blocks.

Variants of the FFS filesystem for different versions and branches of BSD operating systems are the same in concept but differ in details, making filesystems incompatible. All recent BSD systems support management of consistency with Soft updates [8] [9].

FreeBSD 5 has a new version of the FFS filesystem with more features: snapshots (allowing a system administrator to take a frozen snapshot of a filesystem without interrupting operation) and background fsck (allowing recovery of the filesystem after a crash while other processes write to it) [76] [10].

The FFS filesystem is not supported on many non-BSD operating systems. Linux has read-only FFS support (and experimental read-write support for some brands of FFS).

1.3.5 Ext2

The Ext2 filesystem was developed by Rémy Card for Linux in 1993 [11]. It is based on the traditional Unix filesystem (like FFS), but it was developed independently from the FFS — thus it utilizes many similar concepts (bitmaps, directories, file allocation radix trees), but the data layout is completely incompatible with FFS.

Transparent data compression existed for Ext2, but it never made it into the official Linux release and it is not supported since kernel 2.4.

Despite being very old, the Ext2 filesystem is still widely used today. Its simplicity enables very good performance in general-purpose use, although it lacks speed in specialized scenarios, such as storing many files in one directory or very large files. The Ext2 filesystem is very stable; no data corruption bugs have been found in the last few years.

Thanks to Linux's popularity, Ext2 is supported on many other operating systems, including OS/2 [77], Windows NT [78], BSD branches and Mac OS X [79].

1.3.6 Ext3

The Ext3 filesystem is a journaled variant of Ext2. It has the same data layout as Ext2 (thus the same partition can be mounted as Ext2 or Ext3 whenever a system administrator wants), but adds fast recovery after a crash using a journal. Ext3 also adds optional indexing of directory content using a B+ tree (again, backward compatible with Ext2: when Ext2 writes to a directory indexed by Ext3, it destroys the index but doesn't destroy the directory content) [12] [13].

Ext3 was designed by Stephen C. Tweedie in 1998 [14]. Some experimental versions for Linux kernels 2.2 were released but they were never merged to the main 2.2 kernel. Ext3 was added to kernel 2.4.15 in 2001.

Today, Ext3 is the most commonly used filesystem on Linux. Its performance is somehow lacking compared to Ext2, due to journaling. Ext3 didn't destroy data, but its driver had deadlock bugs [80] for some time. Nowadays it's considered stable.

Ext3 isn't supported on non-Linux operating systems but Ext3 partitions can be mounted as Ext2 by operating systems supporting Ext2 (losing journaling capability).

1.3.7 Ext4

Ext4 was created as a code fork of Ext3. It is currently under development in the Linux kernel. It avoids some limits that exist in Ext2 and Ext3: it removes the restriction of 2^{32} total blocks on the filesystem and the limit of 2^{15} subdirectories in one directory. It has timestamps with nanosecond precision.

To improve performance, Ext4 can optionally store file allocation information in extents. Delayed allocation is under development but it hasn't been committed to the Linux kernel yet. Another advanced feature under development is an online defragmenter, allowing the user to defragment a filesystem while it is being mounted.

Ext3 partition can be mounted with an Ext4 filesystem driver. But when some files with extents are created, the filesystem cannot be further read by the old Ext3 driver.

1.3.8 ReiserFS

The ReiserFS filesystem was developed by Hans Reiser [81]. The first version was released in 1999 for the Linux 2.2 kernel. Versions 2 and 3 were released quickly that same year. Version 3.5.5 released in November 1999 included journaling.

Version 3.6 was included in Linux kernel 2.4.1 in 2001 and this version of filesystem is still used today.

ReiserFS is optimized for small files — the whole filesystem is composed of one B+ tree and it stores all objects in it. It stores small files and tails of large files in that tree too, making sure that it uses every byte of storage capacity. It doesn't pad files to block size like other filesystems do. ReiserFS uses bitmaps for maintaining free space information.

ReiserFS is supported only on Linux. FreeBSD recently added read-only implementation.

1.3.9 Reiser4

Reiser4 is the new filesystem developed by Hans Reiser and his team. Reiser 4 brings many concepts from the previous Reiser 3 filesystem, but it is a complete rewrite.

It is also optimized for small files; it includes atomic updates (it can batch any number of updates into one atomic operation — but currently, there is no API to let userspace applications utilize this functionality), delayed allocation (allocation is done when the cache is flushed, not when `write` syscall is called) and plugins for external modules. Reiser4 has very good performance but it is not considered stable yet and it is not included in the standard Linux kernel.

Reiser4 is supported only on Linux.

1.3.10 The JFS

JFS stands for "Journaled File System". As the name suggests, JFS uses journaling to maintain filesystem consistency across crashes. It was developed by IBM for the AIX operating system. It was introduced in 1990 in AIX version 3.1 (JFS1).

The JFS1 was completely redesigned and released for the OS/2 operating system in 1999 (JFS2). In 1999 the JFS2 source code was released under open source license and work began to port it to Linux. In 2001 the JFS2 was released for AIX 5.1. The JFS2 was added to Linux kernel 2.4.20 in 2002 and it was included in Linux kernel 2.6 released in 2003.

The JFS uses journaling to maintain filesystem consistency [15]. The JFS2 had some new features that were not present the in original JFS1 [82] — indexed directories, extendible inode space and increased maximum partition size. Some features present in the JFS1 were dropped from JFS2 — dividing blocks to fragments and transparent compression.

The JFS2 is currently supported on Linux, OS/2 and AIX systems.

Throughout this thesis I refer to the JFS2. I won't describe features of the JFS1 because it is closed source and not used much anymore.

1.3.11 The XFS

The XFS filesystem was developed by Silicon Graphics Inc. [83] It was first released in 1994 for IRIX version 5.3. In 2000 it was released as open source and porting to Linux started [16]. The XFS for Linux was released in the 2.6 kernel in 2003 and in the 2.4.25 kernel in 2004.

The XFS has journaling and uses 64-bit values; thus it has very large limits on the number of blocks, number of files and file size. It uses B+ trees for management of directories and for storage of file allocation information and free space information. The XFS filesystem can support a real-time volume, with guaranteed access speed and latency.

Today, the XFS is supported on Irix and Linux operating systems. Porting to some BSD systems is under way.

1.3.12 The ZFS

ZFS [84] is a filesystem developed by Sun Microsystems. It was introduced in 2005. The filesystem is supported on the Solaris operating system. Unofficial ports for MacOS X, Linux and FreeBSD exist. These ports do not have production quality yet.

Unlike traditional filesystems, ZFS combines the filesystem and logical volume manager. ZFS does not operate on a single partition, it operates on the storage pool that may consist of several partitions on different devices. ZFS can use techniques similar to RAID-0, RAID-1, RAID-5 (called RAID-Z) or RAID-6 (called RAID-Z2) to increase performance and protect data from failure of a disk.

ZFS makes checksums on all metadata and data sectors, and if configured with RAID redundancy, it can transparently recover data even in the case when the disk does not report an error but returns incorrect data. [17] [18] This checksumming makes ZFS consume more CPU than other filesystems.

ZFS supports taking static snapshots of parts of the filesystem [19].

To maintain allocation information of files, ZFS uses radix tree of variable size. The tree contains pointers to blocks and their checksums. Directories are indexed using extendible hashing [20].

SpadFS relates to ZFS, as it makes checksums too, but only on metadata. When checksums fail, SpadFS refuses to access the metadata block and waits until the condition is corrected with the filesystem-checking utility.

1.3.13 The GFS

GFS stands for "Global File System". GFS was developed in 1997 at the University of Minnesota [21] [22]. The first version was created for SGI IRIX, and in 1998 it was ported to Linux. It was commercially supported by Sistina Software and it is now supported by Red Hat, Inc.

Unlike traditional filesystems, GFS runs over a shared storage — a disk device is simultaneously accessed over a network by several computers in a cluster. The computers synchronize their access to the storage. The computers can detect failure of one cluster node and replay its on-disk journal to maintain filesystem consistency and crash recovery that is transparent to other cluster software.

A forked version, GFS2 is now being developed [23].

GFS uses radix tree to maintain allocation information of files. GFS uses extendible hashing for directory indexing.

SpadFS relates to GFS and ZFS in directory management, as SpadFS uses extendible hashing too.

2. Design goals of the Spad filesystem

The main purpose of my design is to create a filesystem that has good performance, survives system crashes and has reasonably small code base. Current journaled filesystems are big, their code size approaches 1MB and as a consequence, they are hard to implement. Small filesystems, such as FAT or Ext2, are easy to implement, but they lack many performance features (such as extents or indexed directories) and require scanning the whole filesystem after a system crash.

In this section I describe basic design goals that influenced the filesystem design. I also describe the basic design decisions that I have made to fulfill these goals.

2.1 Extensibility to arbitrarily fast devices

The filesystem should be extensible with device speed. If you take a 10 times faster disk and read or write file content 10 times faster, the filesystem CPU consumption should scale well. A filesystem having this feature will be extensible to storage devices of arbitrary speed and will guarantee that CPU consumption will not become a bottleneck. A key dewing principle to achieve this goal is not to work with a list of blocks anywhere in the code. Anywhere on disk or in memory, a filesystem works with pairs of (*block*, *length*) rather than with individual blocks.

We try to achieve good scaling in the filesystem. There are other parts of the operating system that may not scale well, for example the page cache or the application itself. So the CPU consumption of the whole system is not completely independent of the disk speed, but I have tried to make the CPU consumption inside the filesystem as little as possible.

Older filesystems do not try to follow this principle — for example Ext2/Ext3 and FFS [7] filesystems store list of blocks for each file in inode or indirect blocks.

Newer filesystems such as JFS, XFS or NTFS store file allocation information in extents. An extent is a triplet of (*logical_block*, *physical_block*, *number of blocks*) describing each continuous area allocated to a part of a file. If you increase the transfer speed but do not fragment the file, the filesystem still works with one extent regardless of the speed; so CPU consumption devoted to working with extents will be independent of the transfer speed. However these filesystems still store free space information in bitmaps and this technique exhibits the same problem. This problem can be somewhat mitigated by using larger block sizes, however this solution does not solve it completely — block size can be increased only up to some limit, page size (4096 on most architectures) on Linux or 65536 bytes on FreeBSD. Increasing block size decreases the effectiveness of storing small files.

The Spad filesystem solves this problem by storing free space information in extents too. Free space information is stored in lists of pairs (*sector*, *number of sectors*). When the filesystem needs to allocate and write an arbitrarily large file, its CPU consumption is not dependent on the size of the file — it finds an extent of an appropriate size in the free space list.

Experimental measuring of CPU consumption with increasing device speed can be seen in section 10.2.5.

2.2 Good efficiency for both small and large files

Some workloads (such as compiling large programs) create a lot of small files, while other workloads (such as working with video) create a few large files. A filesystem should have good performance for both types of workloads. Because of the inefficiency when working with many blocks, system administrators usually select a large block size to achieve better performance. However this makes the filesystem use a lot of space when storing many small files, because each file is padded up to a block size and this padding consumes additional space.

The FFS [7] solves this problem by dividing a block into mostly 8 fragments and allocating files smaller than the block size in fragments (a typical configuration preselected on FreeBSD installation is a block size of 16384 and fragment size of 2048). However, bitmaps describing free space have to contain a bit for each fragment, not block, and this slows down file allocation.

ReiserFS solves this problem by storing small files and tails of large files directly in the tree, among inodes and directory entries. This is the most space-efficient strategy of all filesystems at the expense of large code complexity.

Other filesystems (JFS, XFS, Ext2/Ext3) do not try to efficiently store small files and pad each file to the block size.

Because CPU consumption of the Spad filesystem should not be dependent on the block size (as described in the previous chapter), it can work with a block size as small as 512 bytes without performance impact. Thus it pads each file only to a 512-byte sector size and it achieves good space efficiency without the complexity of ReiserFS.

This optimization applies only to SpadFS on the Spad kernel. The Linux kernel has inefficient memory management when small blocks are used, so SpadFS uses page-size blocks by default, like most Linux filesystems.

Efficiency when working with small files can be seen in section 10.2.7 (postmark benchmark), efficiency when working with large files can be seen in the section 10.2.4 (one-file access) or in the section 10.2.8 (FFSB benchmark).

2.3 Reduced code complexity

Filesystems developed in business environments are usually too complex, which leads to various problems with performance, stability and code maintenance. Complex software takes more time to write and test and is prone to bugs. In extreme cases, complexity can reach *critical mass* [85], a condition when one bug fix introduces one or more new bugs. Software in such a condition is unmaintainable and must be rewritten from scratch.

The Spad filesystem is designed for a small complexity of data structures and algorithms. The filesystem doesn't use B-trees [86]. Although B-trees are popular in database design because they can be used to optimize evaluation of a large amount of queries, simpler methods can be found for use in filesystems.

Instead of B-trees, my algorithm is based on extendible hashing [20] for fast lookup of directory entries. It requires even a smaller number of disk accesses than B-trees and is easier to implement.

The size of the code of various filesystems can be seen in section 10.3.

2.4 Fast recovery after crash

It is important that the time needed to recover a filesystem after a crash is not dependent on the filesystem size. In the industry, journaling has been widely used to achieve this goal. However journaling has many issues (the need to properly order requests, the need to account space in a journal) and solutions to these problems increase code complexity. These solutions are also prone to bugs [15] [87] [88].

The Spad filesystem uses a novel method, crash counts, to manage data consistency in the case of a crash [24] [25]. Crash counts reduce code complexity and improve performance.

The crash count method is described in detail in chapter 3.

2.5 Data layout suitable for prefetching

The transfer speed of disks is constantly growing while seek time remains almost the same. Reading one sector typically takes $10\mu s$ while seek takes $10ms$. The cost of reading additional sectors is almost negligible, so it is desired to pre-read more data than requested and use this additional data to fulfill further requests without any disk access. Current filesystems do prefetch within one file; the Spad filesystem allocates space on disk in a way that allows prefetching of metadata. When the user accesses many files (for example with `find` command), the Spad filesystem can read metadata in large requests, prefetching entries for several hundred files. Because of this, file searches or checking the whole filesystem is faster.

Metadata prefetch is possible because the Spad filesystem allocator uses zones. Filesystem has three zones — for metadata, small files and large files. Metadata are allocated in one zone near each other, without interleaving data; thus if filesystem reads large continuous area, there is a high probability that many file structures will be contained in this area and accessing all these files in the future won't require any disk access. On filesystems without zones, metadata prefetch is not efficient because if you read past requested directory or file structure, you will likely read file content or unallocated space — not the other file structures.

The allocator prefers to allocate data in corresponding zone; however it allows allocation in other zones if the requested zone is full. If a user creates too many files so that they fill up a metadata zone, metadata will be allocated anywhere on the filesystem. The only effect will be that the prefetch won't work on these files and access to them will be as slow as in filesystems without zones. On the Spad filesystem, disk space can be used for any purpose; there is no preallocated part for inodes such as in the Unix filesystem.

The effect of prefetching when traversing a directory tree with various commands can be seen in section 10.2.10.

3. Crash count method

In this chapter, I will describe crash counts — a novel method for maintaining filesystem consistency. The method is described in [24] and [25]. So far no other filesystem has used this method. First, I will discuss other methods used to keep a filesystem consistent.

3.1 Related work

3.1.1 Journaling

Several methods have been developed for maintaining consistency. Journaling [14] [26] is the most common one.

One part of a filesystem is reserved as a journal (some filesystems can have a journal on an external device). A transaction is a set of modifications that transfer the filesystem from one consistent state to another. For example, when creating a file, a transaction consists of marking the new inode allocated in the inode bitmap, writing the new inode and updating the directory. The transaction is first written to the journal and no metadata are written to their normal disk positions. When the transaction is completely written in the journal, the disk cache is flushed, a special commit mark is written to the journal, the disk cache is flushed again and the metadata can be written to normal positions. If the system crashes before the commit mark is written, the journal is ignored on the next mount. If the system crashes after the commit mark has been written, operations in the journal are replayed — written to disk positions where they should be. This causes either the whole transaction to be ignored or to be written to the disk.

Most filesystems store only forward changes (the changes that they intend to do) to the journal. The NTFS [7] stores both forward and backward changes (original data) in the journal, so that it can rollback transaction in the case of write I/O error or journal fill-up.

Although journaling is a widely used industry standard, its correct implementation is very complex and it is prone to bugs [15] [24] [24]. There are several problems associated with journaling:

- **Buffer pinning** — The data must be first written to the journal, then the commit mark must be written to the journal, and only after that, can the data be written to their permanent location. Buffers containing the data have to be pinned in memory until the journal is committed. If free memory goes low, the system cannot write dirty buffers and free them, thus potentially deadlocking in a wait for free memory that never comes (this condition is called low-memory deadlock [89]). This problem can be solved by integrating the memory management code with the journaling code to force commit when memory goes low, however it increases code complexity.
- **Free space accounting** — A journal must not fill up; thus before starting the transaction, the system must compute the upper limit of journal space to be consumed by the transaction and make sure that the sum of all block counts of all in-progress transactions does not overflow journal free space. The NTFS solves this problem

differently — it writes both undo and redo records to the journal and undoes the transaction in the case of journal fill-up.

- **Forced write ordering** — Journaling forces data ordering in many situations — for example, a journal commit mark may be written only after journal content has been written. Data can be written only after a journal commit mark has been written. This ordering is implemented by sending a command to flush the hardware disk cache.

3.1.2 Log-structured filesystems

Log-structured filesystems (such as LFS [27] [28] [29] [30], Spiralog [31]) take the following approach to maintain consistency: they can be viewed like journaling filesystems, except that they don't have any data area and the journal spans the whole device. While the filesystem operates, free space on the disk becomes fragmented and so does the journal.

These filesystems have very good write performance — because writes are consecutive without seeking — and low read performance — because they don't group files from the same directory near each other. For example, a classic filesystem will try to store files in one directory to one location and files from some other directory to another location (so that when all files in one directory are read, seek time will be minimized). On the other hand, a log-structured filesystem writes new data to a place where the log head currently is — thus files that are in different directories, but were written at the same time, will be placed near each other and files that are in the same directory and were written at different times will be placed far from each other. Another problem with these filesystems is that they fragment a lot — i.e. writing to the middle of a continuous file will fragment it.

Log-structured filesystem design was based on the assumption that most read requests hit the cache and the workload is dominated writes. However, this assumption was disproved [32].

Log-structured filesystems have one functional advantage over classic filesystems — they allow fast and easy creation of a snapshot. During filesystem operation, a special process called a *cleaner* searches for the areas in a log that have been overwritten by newer records and deallocates them. If you stop the *cleaner* and take a pointer to the current log head, you can view a static layout of the filesystem. While you examine the snapshot, applications can continue writing to the filesystem, and you don't see these new changes on the snapshot. These snapshots are most commonly used for the purpose of backup.

3.1.3 Phase tree

A phase tree [90] is a different way to maintain filesystem consistency. The whole filesystem is viewed as a tree of pointers: a superblock points to an inode directory and free-space bitmap directory; an inode directory points to inodes; a bitmap directory points to bitmaps; inodes point to the actual content of the inodes. New data or metadata are always written only to the unallocated space on the disk — thus they don't disrupt the filesystem layout in the case of a crash. When a new superblock with new pointers is written, the filesystem atomically transfers from one consistent state to an other. Phase tree is used in Linux filesystem Tux2. It is probably slower because it has to write more blocks (the whole path from the leaf of the tree up to the superblock) than normal filesystems when doing metadata updates.

3.1.4 Soft updates

Soft updates [8] [9] [10], unlike journaling and phase tree, do not preserve filesystem consistency. Instead, they ensure that the filesystem can be corrupted only in minor non-severe ways. When the filesystem driver marks dirty buffers, it specifies the order in which the buffers must be written to the disk. For example, when a file is created, the driver specifies that the inode should be written first, then the inode bitmap should be written and finally the directory entry. Kernel bufdaemon thread writes buffers in this order and issues disk cache flush operations correctly. If the system crashes at any time, it may cause lost inodes or blocks but no pointers pointing to unallocated or uninitialized inodes or blocks. Soft updates are implemented in the family of BSD operating systems.

A performance evaluation of soft updates and journaling can be seen in [33] [34].

FreeBSD 5, because of its filesystem snapshot feature, allows checking of the filesystem while it is mounted in read/write mode — a snapshot is taken, that static snapshot is checked, and fsck instructs the kernel to free lost blocks and inodes on the mounted filesystem (instead of writing to the snapshot, which is not allowed). This checking of a live filesystem can only fix errors caused by a crash (lost blocks and inodes); it can't fix general filesystem errors caused by hardware failure or kernel bugs. With this feature, FreeBSD 5 allows an immediate recovery after a crash, except that performance is degraded for the time fsck is running.

3.1.5 Conclusion

Each of these solutions has some problems — complexity in the case of journaling, fragmentation and lack of data locality in log-structured filesystems and phase tree and the need to check the whole filesystem in soft updates.

Thus I developed my own novel method — crash counts.

3.2 Overview of crash counts

The crash counts method uses generation counts on structures to allow atomic commitment of several modifications.

Each data structure that must be kept consistent with other structures (for example, directory blocks, file descriptors and free space maps) contains two values — a crash count and a transaction count. The disk has a preallocated crash count table. The crash count and transaction count in a structure together with a crash count table determine if the structure is valid. Several structures can be atomically made valid with only one write to the crash count table.

The filesystem driver writes new structures with such a crash count and transaction count that all the new structures will be considered invalid in the case of a crash. When the filesystem driver needs to commit changes (when a writeback timer expires or `sync` or `fsync` syscall is issued), it writes one sector to a crash count table and makes all previously written structures valid.

The filesystem on the disk is always consistent; there's no need to do any checking or journal replay after a crash. With write to crash count table, the filesystem makes a transition from one consistent state to another.

The crash count method solves many problems found in journaling:

- It does not impose any ordering on data writes — the filesystem can write many megabytes of data without ever issuing a disk cache flush request. This allows the filesystem to fully utilize the disk cache. The disk cache must only be flushed when updating a crash count table.
- No data are ever pinned in memory. Memory management can flush buffers whenever it needs, so no possibility of low-memory deadlock [27] exists.
- Unlike in journaling, metadata don't have to be written twice. This further reduces code complexity and improves performance.

3.3 Data structures

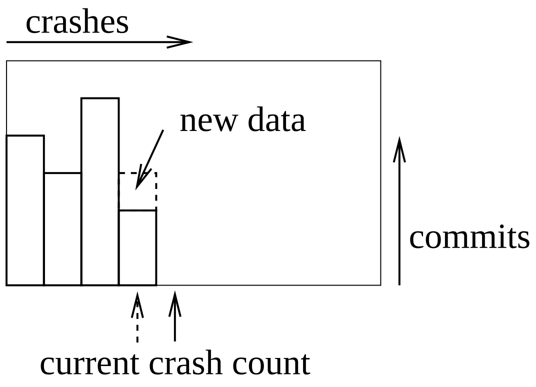

Figure 3.1: Crash count table overview

The disk contains a single value — a crash count (in my implementation a 16-bit integer). When the filesystem is mounted, the crash count is increased and when the filesystem is unmounted, the crash count is decreased. Thus, the crash count value corresponds to the number of system crashes since the filesystem was created.

The disk also contains a crash count table — an array that contains a transaction count for each crash count. The transaction count is 32-bit wide in my implementation. Thus the size of the crash count table is $4 * 2^{16} = 256\text{kB}$.

During operation, the crash count and the crash count table is loaded in the memory in the structure describing the mounted filesystem (here, I call this structure `fs` and I will refer to the crash count loaded in memory as `fs->cc` and the transaction count loaded in the memory as `fs->cct[]`). The crash count and the crash count table in the memory slightly differ from their content on disk — an exact description will be given in the next section.

Each structure on disk that needs to be kept consistent with respect to other structures has two additional values — a crash count and a transaction count. I will refer to them as `structure->cc` and `structure->txc`. `structure->cc` is an index into the crash count table and `structure->txc` together with the value in the crash count table determines structure validity.

A crash count table is shown in figure 3.1. Solid lines represent the state on the disk. Dotted lines represent the state in the memory. New data are written with crash count and transaction pointing to the dotted square. When the data are *synced*, a crash count table is written to the disk, making the structures atomically valid.

3.4 Mounting, unmounting, syncing

On *mount*, the filesystem loads a crash count and crash count table from the disk to the memory. It increases `fs->cct[fs->cc]` in the memory, but leaves the old value on the disk. It increases `cc` on the disk, but leaves the old value in the memory.

On *sync*, the filesystem

- 1. writes all dirty buffers and pages to the disk
- 2. flushes the hardware disk cache
- 3. writes one sector of the crash count table from the memory to the disk
- 4. flushes the hardware disk cache
- 5. increases its in-memory value `fs->cct[fs->cc]`

If a crash happens before the write in step 3 completes, the old version will be permanently valid; if a crash happens after the write in step 3 completes, the new version will be permanently valid. Because some disks have a hardware write cache, it is unclear if the write request initiated in step 3 will be finished in step 3 or 4. However, when the sequence passes step 4, the new state is permanently committed for sure.
Sync is issued either upon user request (the `sync` command) or by the filesystem itself in certain situations (unmounting, running out of space ... see below). *Sync* is also issued periodically with a configurable period to make sure that modified cache blocks are not held indefinitely.

On *unmount*, the filesystem is *synced*,the crash count on the disk is decreased and the hardware disk cache is flushed.

When increasing values, we must care about integer value overflows. If the value in a crash count table would overflow over the 31st bit, the value is not modified and `fs->cc` is increased instead. When `fs->cc` overflows the 16th bit, a complicated action must be taken — the whole filesystem must be scanned and the crash counts and transaction counts on all structures must be reset. Then `fs->cct` must be filled with zeros and `fs->cc` reset to zero. This is very slow; however, it doesn't happen at all in practice. It would happen either after 2^{16} crashes or after 2^{47} *sync* operations (given the fact that current disks can do at most about 100 *syncs* per second, the count would overflow after 44597 years of continuous operation consisting of only syncing).

Note that the transaction count does not necessarily increase with every transaction, more transactions can be batched into one transaction count increase. The transaction count can be viewed more like a checkpoint count known from journal filesystems.

3.5 Managing directory entries using crash counts

Data structures that form lists (directory entries in my implementation, however the concept is general) contain (`cc`, `txc`) pair. Here I will refer to a directory entry in a list as `dir_entry`. The structure `dir_entry` is considered *valid* if the following condition is true: `cct[dir_entry->cc] - dir_entry->txc >= 0`. If the structure is not *valid* it is skipped when searching the list during `open` or `readdir` syscall.

Entries are created with `dir_entry->cc = cc` and `dir_entry->txc = txc[cc]`. After the creation the above condition holds true — so the entry is considered *valid*. If the system crashes before *sync*, the value `txc[cc]` on the disk will be one less than the value in the structure and it won't ever be changed (`cc` will be increased) — thus the structure will be considered *invalid* forever. If the system crashes after *sync*, the structure will always be considered *valid* because `txc[cc]` on the disk will be increased and it's never decreased.

If the directory entry needs to be deleted, the filesystem sets `dir_entry->cc = cc` and `dir_entry->txc = txc[cc] ^ 0x80000000`. This is a trick with bit operations just to save one flag in the structure and one conditional expression in the code. The above condition defining *validity* will work just the same way — the structure will be considered *invalid* from now. If the system crashes before *sync* it will be *valid* back again. If the system crashes after *sync* it will be always *invalid*.

For proper handling of directory entries deletion, the test if the entry is *valid* must be `cct[dir_entry->cc] - dir_entry->txc >= 0`. Simply comparing two transaction counts (`cct[dir_entry->cc] >= dir_entry->txc`) is not correct. Note that the subtraction and comparing against zero will properly handle both inserted and deleted structure (provided that the machine has 2-complement arithmetics), while comparing two values will handle only insertion, and another test for deleted structures would be needed.

Directory entries have different sizes and empty entries are reused. Care must be taken to not reuse an entry that would become *valid* in the case of a crash. Generally, an entry may be *reused* if its not *valid* and if it hasn't been deleted since the last *sync*. If the entry was deleted since the last *sync*, it can become valid after a crash — so it can't be *reused* in this case. The exact test if the entry has been deleted since the last sync is `dir_entry->cc != cc || dir_entry->txc != txc[cc] ^ 0x80000000` (see above: when the entry is deleted, the values are set as `dir_entry->cc = cc; dir_entry->txc = txc[cc] ^ 0x80000000` — so when the values differ, a *sync* operation was already performed). Two smaller empty directory entries can be joined into one larger if both can be *reused*. Directory entries must not cross the disk sector boundary (512 bytes) because then they would not be written atomically.

3.6 Managing other disk structures

Data structures that are not lists (free space maps and hash tables for large directories in my implementation) can be managed with crash counts, too. Each structure has two versions on the disk and one (`cc`, `txc`) pair. In this chapter, I refer to the structure as a *bitmap*, although the concept is general and can be used for any structures.

If `cct[bitmap->cc] - bitmap->txc >= 0` the first version is considered *valid*, otherwise the second version is.

When writing to the bitmap, a check is made if the bitmap has been modified since the last *sync* operation (The exact test is this: `bitmap->cc == cc && (bitmap->txc & 0x7fffffff) == cct[cc]`). If true, the filesystem can write to the currently *valid* version of the bitmap directly. In the case of a crash, the second version will be considered *valid*. If false, the filesystem must not modify the bitmap, because modifications to the *valid* version would be visible after the crash. It copies the *valid* version to the *invalid* version and sets its (`cc`, `txc`) pair so that the new version is now considered *valid* and in the case of a crash, the old version would be considered *valid*. The exact operation of setting `bitmap->cc` and `bitmap->txc` is

```
if (cct[bitmap->cc] - bitmap->txc >= 0) {
   bitmap->txc = txc[cc] | 0x80000000;
} else {
   bitmap->txc = txc[cc];
}
bitmap->cc = cc;
```

When allocating blocks, the filesystem must not allocate blocks that are free in the current bitmap but are marked allocated in the bitmap that would be valid in the case of a crash. If the bitmap has been modified since the last *sync* (i.e. `bitmap->cc == cc && (bitmap->txc & 0x7fffffff) == cct[cc]`) both bitmap versions must be checked. If no free block exists, *sync* is performed. If no free block exists even after *sync*, the request is aborted and the "disk full" error is returned.

3.7 Managing file allocation info

File allocation descriptions could be managed with lists of runs like directory entries or doubling pointers like bitmaps and directory hash pages. I decided to use a different method that makes file descriptors smaller[1].

Each file has a list of runs — i.e. pairs of (disk offset, number of sectors). These runs are organized in unix-like direct/indirect block structure. The first two runs are stored directly in the file descriptor for fast access to files with only two fragments. This layout has the advantage that runs can be added without affecting already existing ones. An on-disk file descriptor contains two 64-bit size values (`size0`, `size1`) and a (`cc`, `txc`) pair which determines which of the size values is *valid*. If `cct[file_descriptor->cc] - file_descriptor->txc >= 0`, `size0` is considered *valid*, otherwise `size1` is. The (`cc`, `txc`) pair of file descriptors is managed the same way as with other structures — i.e. when extending or truncating a file it is set so that in the case of a crash the old size version will be considered *valid*. Space described by run descriptors is considered allocated only if a run descriptor offset falls within the *valid* `size` in the file descriptor. So the run descriptors don't have to be protected with a (`cc`, `txc`) pair; all that has to be protected is the file size. This saves space in the run descriptors and simplifies their handling.

Special care must be taken when the file is truncated and extended with no *sync* between these operations. If I allocated new blocks for extending the file, the allocation information for the old runs would be overwritten and this would cause inconsistency in the case of a crash. There are two methods to deal with this. The simpler (and slower) method is to invoke *sync* in this case.

sync is slow and blocks the writing process, so a better method can be used to deal with this situation. Note that blocks freed in a previous truncate operation are still marked allocated in *invalid* versions of bitmaps (these versions would become *valid* in the case of a crash) — so the allocator hasn't reused the blocks yet. If it is needed to extend the file, these blocks are marked as allocated in the currently *valid* bitmaps and the file is extended at the place where it was before the truncate operation. This operation is called ”resurrecting” blocks. Thus the allocator has three functions: to allocate blocks, free blocks, and resurrect blocks.

Crash counts are used only to manage allocation information, not the data itself. Data are written asynchronously in any order (it is even possible that reordering happens inside the hardware disk writeback cache). Data is flushed only when the application calls function `fsync` or `fdatasync` or when the system requests *sync*.

3.8 Requirement of atomic disk sector write

The crash count method, as described above, requires that the disk is capable to write one sector atomically. i.e. that in the case when power is interrupted, the sector contains

[1] The size of structures is important in filesystem design — if you make on-disk file descriptors larger, scanning the directory tree will take proportionally longer.

either old or new data. Most disks have this feature. If the disk has, for example, 40MB/s write speed, it takes 13us to write one sector. The disk can be operational for this time on power from its capacitors when the power is interrupted.

Now, let me write some consideration for RAID [35]:

In the case of RAID-0 array[2], the behavior during power outage is the same as in the case of a single disk, except that sectors may be written out of order (the sector that is submitted later is finished before the sector that is submitted earlier). This out of order writing can also happen with a single disk, if write cache is enabled. The filesystem can handle the out of order completion of writes; it issues flush operation between writes that must not be reordered. So it can safely work with write cache enabled or with RAID-0 array.

In the case of RAID-1 array, the situation is the same as in the case of a single disk. After a crash, the RAID-1 driver resynchronizes parts of the device that were being modified prior to the crash.

The most complicated scenario comes with RAID-4 and RAID-5. If the RAID is not degraded (all the disks are operational), the situation is analogous to RAID-0 array. In the case of a crash, the RAID driver resynchronizes parity blocks in regions that were being modified while the crash occurred. If RAID-4 or RAID-5 is degraded (one disk is missing), data consistency in the case of a crash is not guaranteed; but it is not guaranteed on any journaled or non-journaled filesystem. In the case of desynchronized RAID array, a write to one sector updates the sector and the parity block. If one of these updates finished and the other not, and the system crashes, the content of the sector on the missing disk that was protected by the parity chunk is modified, although the sector was not written by the filesystem. No filesystem can deal with this situation. The user should not run RAID in a degraded mode for long periods of time, or he should use hardware RAID controller backed by battery that is resistant to this scenario.

Correctness of the crash count method was tested with FSSTRESS and LVM2 snapshots as described in section 10.3.

3.9 Avoiding blocking of filesystem activity during sync

While the filesystem is performing *sync*, the method (as described in section 3.4) would block concurrent filesystem activity during syncing. This would have a very undesirable effect on performance in scenarios like when one process is performing frequent `fsync`s and other processes are accessing the filesystem. In this chapter I describe three possible solutions to this problem, as I implemented them. The possibility to implement these solutions depends on the capabilities of the operating system.

[2] The analysis of RAID-0 reliability can be found in [36].

3.9.1 Writing buffers twice

This method was first implemented on the Linux driver. *sync* is done in the following steps:

- 1. write all dirty buffers and wait for completion
- 2. acquire rw-lock for writing, preventing metadata updates
- 3. write all dirty buffers again and wait for completion
- 4. write a sector of the crash count table and wait for completion
- 5. increases in-memory value `fs->cct[fs->cc]`
- 6. release rw-lock

Note that the rw-lock is acquired *for read* on any writes to the filesystem — thus the writes can happen concurrently with respect to each other and they are serialized only on *sync*. The lock is optimized in such a way that when more CPUs take it for reading concurrently, it doesn't cause the cache line ping-pong effect [91].

This method blocks filesystem activity during *sync*, but double writing of dirty buffers minimizes the time of blocking. Most data are written during the first buffer write, but the lock is not held, allowing other concurrent writes to create more dirty buffers. Then the lock is taken and all dirty buffers are written again — those are only buffers that were made dirty during the first write. Finally the filesystem driver flushes the crash count table, waits for completion and releases the lock. Concurrent activity is blocked only during the second write of buffers and the update of the crash count table.

3.9.2 Write barriers

Write barriers were removed from the Linux kernel in version 2.6.37, so I removed barrier support from Spadfs driver as well. This section describes how barriers were used in previous versions of the Spadfs driver.

Linux offered write barriers [92] to maintain ordering of disk I/Os.

Any submitted write request could have a `WRITE_BARRIER` flag. Linux block device I/O layer made sure that:

- All requests that were submitted prior to the request with a `WRITE_BARRIER` flag complete before the request with `WRITE_BARRIER` flag completes. The requests that were submitted prior to the `WRITE_BARRIER` request complete in unspecified order with respect to each other.
- All requests that were submitted after the request with a `WRITE_BARRIER` flag complete after the request with `WRITE_BARRIER` flag completes. The requests that were submitted after the `WRITE_BARRIER` request complete in unspecified order with respect to each other.
- If the disk has a hardware cache, it is flushed just before the `WRITE_BARRIER` request and immediately after it completes — to make sure that in the case of power failure, ordering is not violated.
- All requests (requests before, after and the `WRITE_BARRIER` request itself) complete asynchronously. The process may wait for any of these requests but it doesn't have to.

I used this asynchronous capability of barrier requests to further reduce blocking of filesystem activity during *sync*. The *sync* operation is implemented in this way:

- 1. acquire rw-lock for writing, preventing metadata updates
- 2. write all metadata buffers, do not wait for completion
- 3. write a sector of the crash count table with `WRITE_BARRIER` flag, do not wait for completion
- 4. increase in-memory value `fs->cct[fs->cc]`
- 5. release rw-lock

Linux kernel completes writing metadata buffers and updating the crash count table asynchronously with respect to this code sequence. The write barrier makes sure that the updates will be flushed to the disk in a correct order.

As we see, there is no waiting while the lock is held. Concurrent requests that go into the cache are not blocked at all and concurrent requests that go to the disk are blocked, only by the disk I/O, not by the lock.

When the *sync* operation is invoked on behalf of the `sync` or `fsync` syscall, the process must wait for the write to complete anyway. The important fact is that this waiting is done outside the locked code, not blocking concurrent activity.

Linux didn't support write barriers for all types of block devices (it supports them for the most common devices — IDE and SCSI disks). For unsupported devices, the previous method is used.

3.9.3 Cache in VFS

In the Spad operating system, the cache is in the VFS layer, above the filesystem driver. The user can create, write and delete files or directories without any interaction with the filesystem driver itself. The VFS layer submits more operations to the filesystem driver in batch. (More notes on this method are in section 9)

This model completely bypasses the problem of concurrent operations during *sync*. When the *sync* operation is in progress and another process wants to do modifications to the filesystem, the modifications are stored in the cache above the filesystem driver, and the control is returned immediately to the process.

3.10 Summary

In table 3.1 you can see a summary of the methods for maintaining consistency in filesystems.

Some filesystems allow atomic updates of data (i.e. a write to the middle of a file either finishes or doesn't modify anything and writes are not reordered). Ext3 and Ext4 have optional journaled data but their support for atomic updates is very limited (it guarantees that writes won't be reordered but it can't guarantee the atomicity of a request larger than the block size). Windows Vista, Reiser4 and ZFS have full atomicity support. In Reiser4 and ZFS this functionality is not exported to the application interface because there is no application interface for Unix-like systems allowing applications to do atomic filesystem modifications. In Reiser4 atomic updates are exported to in-kernel Reiser4 plugin modules. It should be noted that atomic updates are not required for running databases because databases have their own journal within a file and they can handle reordering of write requests without any data damage.

	Method for maintaining consistency	Atomic data updates
FAT	none	n/a
HPFS	none	n/a
NTFS	Journaling	Vista only
FFS	Soft updates	No
Ext2	none	n/a
Ext3	Journaling	Optional
Ext4	Journaling	Optional
ReiserFS	Journaling	No
Reiser4	Log-structured	Yes
JFS	Journaling	No
XFS	Journaling	No
ZFS	Log-structured	Yes
GFS	Journaling	No
SpadFS	Crash counts	No

Table 3.1: Methods for maintaining consistency across crashes

4. Directories

4.1 Methods in existing filesystems

Modern filesystems (HPFS, NTFS, JFS, XFS, ReiserFS, ZFS, GFS) use some advanced data structures that allow fast lookup of filenames in directories without the need to scan the whole directory. Older filesystems (FAT, Ext2, ISO9660) use a linear scan of a directory when looking up a directory entry. Ext3 has recently added the capability for fast directory lookup in such a way that it is backward compatible with Ext2 and old Ext3 implementations.

Most of these implementations use B-trees or B+ trees for directories. Some are indexed by a full filename (HPFS); some are indexed by a part of a filename (JFS — this has the advantage that the tree-management code is simpler because of fixed-size nodes; however if there are too many directory entries that have long filenames and differ only in the last few characters, lookup generally degrades to a linear scan). ReiserFS, Ext3 and XFS hash the filename to a 32-bit value and uses this hash value instead of the filename for lookup in the tree.

4.1.1 The HPFS

The HPFS [5] [6] is probably the oldest filesystem that introduced the concept of trees in directory management. It uses B-trees alphabetically indexed by file names. Unlike the formal B-tree specification, nodes have different lengths depending on the length of their file name. Directory entries contain information sufficient to satisfy a directory listing with `DIR` command (i.e. file size, access/modification/create times, DOS file attribute flags, size of extended attributes) — so that there is no need to seek to other places of the disk when listing a directory. Directory entry also contains a pointer to an on-disk structure representing a file (fnode). Fnode contains information that is needed when the file is open (i.e. allocation information, content of extended attributes and access control list). An example of a HPFS directory is shown in figure 4.1.

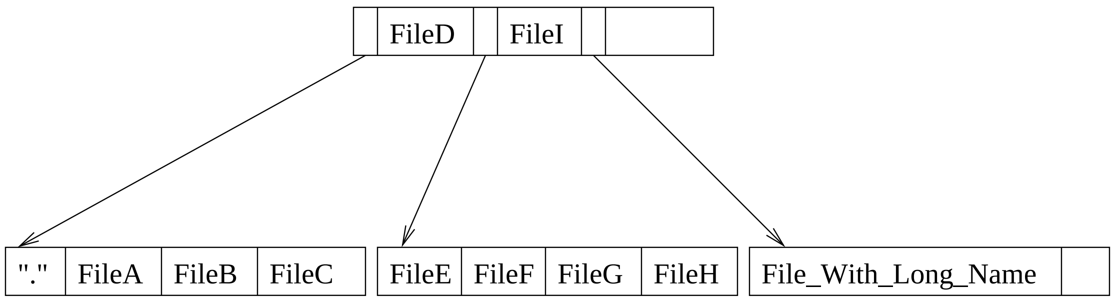

Figure 4.1: B-tree-based directories in the HPFS

Although B-trees look nice in theory, their implementation is rather hard. I've written code to manage B-trees in implementation of the HPFS filesystem [7] and it showed several problems. First — if disk space runs out in the middle of B-tree splitting, serious problems arise. The operation can't progress forward and splitting can't be finished and if it returns an error, the directory will be in an inconsistent state and some entries will be missing. Second — because inner nodes of B-trees contain full filenames, removal of a file entry can actually replace the node with a longer filename, split the tree and allocate more space. Allocation of space on file deletion is a weird feature. OS/2 solves these problems by maintaining 20 "emergency" directory pages and it allocates them only when running out of space in the middle of B-tree operation. When it runs out of these pages, the whole system crashes. This is not good, because it allows the user to stop the system with a sequence of legal filesystem operations.

I have solved these problems by testing if there's enough space before the start of the splitting, but it is rather slow. The test for available space can still fail when deleting files, leaving the user in an absurd situation when he has a full disk and can't delete a file because that would consume more space (this happens only when the user tries to delete a file that is an internal tree node, so the vast majority of files can still be deleted on a full disk).

4.1.2 The JFS

IBM's next filesystem, the JFS [93], uses trees too. Instead of B-trees, it uses B+ trees — i.e. it stores directory entries only in leaf pages. Internal tree nodes contain the first 30 bytes of a filename. Another advantage over the HPFS is that B+ tree pages contain nodes in a random order and contain a sorted array with pointers to actual nodes. Thus, when adding or removing a node, only the array has to be moved, not the entries themselves.

4.1.3 The XFS

The XFS [37] filesystem hashes the filename to a 32-bit integer and uses the integer for lookup in a B+ tree. Because the B+ tree contains integer keys instead of filename keys, more entries can be placed into one B+ tree page. Directories in the XFS have the format of files — i.e. there is one file allocation description mapping logical blocks to physical blocks and the B+ tree is contained within these logical blocks. This unfortunately slows down directory lookup because two lookup operations have to be performed for each step in B+ tree walking: finding the logical block of the next node and finding the physical disk block corresponding to this logical block.

The XFS has two versions of directories, legacy `DIR` and new `DIR2` layout. `DIR` is a plain B+ tree; `DIR2` added the ability for its own management of free space within the logical file.

4.1.4 ReiserFS

ReiserFS [94] uses a B+ tree as well, with one noticeable difference from the previous filesystems: there are no separate trees for directories; the whole filesystem is composed of one tree. Keys in this tree have 128 bits: a 32-bit directory ID, 32-bit object ID, 60-bit offset and 4-bit type. Precedence of parts of a key is this: 1st directory ID, 2nd object ID, 3rd offset, 4th type.

There are four types of objects in the tree:

- stat items — they are like inodes in the traditional Unix filesystem.
- direct items — they contain content of small files or tails of larger files. ReiserFS can store file content directly in the tree — that makes it the most space-efficient filesystem for small files compared to all other existing filesystems.
- indirect items — they contain pointers to content of larger files, stored out of the tree.
- directory — contains a directory entry: a name and a pointer to the corresponding stat item.

The 32-bit directory ID is the most significant part of the key when comparing keys. Its purpose is to group items related to one directory to the same part of a tree to increase performance when accessing files in one directory.

Directories are organized in the following way: the file name is hashed with one of three user-selectable algorithms, 25 bits of this hash form the upper part of the offset value and a 7-bit generation number forms the lower part of the offset (for the purpose of directory lookup, only a 32-bit offset value is used). The generation number is normally zero; entries with increased generation number are created only when hash collision occurs.

Reiser4 uses the same method; it just increased the number of bits in a key.

4.1.5 Ext3

Ext3 filesystem has the same layout as Ext2, except that it uses a journal. The same partition can be mounted either as Ext2 or as Ext3. When the partition is mounted as Ext2 and the system crashes, `fsck` program must be run to check the partition. The directory in Ext2 has the same format as in the traditional Unix filesystem — i.e. it is just a linear list of records containing pairs (file name, inode number). The record size is variable, depending on the length of the file name. Mapping of logical directory blocks to physical disk blocks is the same as for regular files — i.e. it uses direct/indirect blocks.

A long time ago (in Linux kernel 2.2) Ext2 added the possibility to store the type of an inode into a directory entry. This can significantly improve the speed of certain operations, such as searching for a file with a given name in the directory tree. Before that, `find` utility had to read the inode of every directory entry to determine if the entry referenced a subdirectory and if it was needed to descend into that subdirectory. With the file type in directory entries it doesn't have to read files' inodes, it only needs directory inodes.

Directory indexing was added to kernel 2.6. File names are hashed (the user can select one of three algorithms) and this hash is used to look up in a B+ tree. Leaf pages of the B+ tree contain variable-sized directory entries; internal nodes contain only hash values and pointers to lower levels.

The index is created in such a way that the data format is fully backward and forward compatible between new and old implementations. When an old implementation not supporting the index writes to the directory it destroys the index but doesn't destroy the directory content. The first block of a directory is the root node of the B+ tree. Legacy Ext2/Ext3 implementations view this block as empty and so they create new directory entries there — this destroys the index and informs new index-aware implementations that the directory is no longer indexed. Blocks containing internal tree nodes are also viewed as empty to the old implementations, so that they skip them when scanning the directory or overwrite them when creating new entries. The index destroyed by an old implementation can be rebuilt only by `e2fsck` program — until it is run, the new implementation uses a linear search on the directory.

The index contains logical directory blocks, not physical blocks. This somewhat slows down the lookup — for each accessed B+ tree node, several reads have to be performed — zero, one or two accesses to indirect blocks for mapping logical block number to physical block number and one read of the block itself. On the other hand, using logical blocks inside a B+ tree improves compatibility — when an old filesystem defragmentation tool not supporting the index is run on a filesystem with indexed directories, it doesn't destroy anything. If this tool was run on a directory index with physical block pointers, the results would be disastrous (pointers in the tree pointing to unallocated parts of the disk or to completely unrelated files).

4.1.6 Extendible hashing: ZFS and GFS

ZFS and GFS use Fagin's extendible hashing for management of directories.

The original extendible hashing, as designed by Fagin [20], uses fixed-size data pages and a variable-size linear hash table. The hash table size is a power of two. The hash table contains pointers to data pages. More pointers in the hash table can point to the same data page. There can be no `NULL` pointer in the hash table — i.e. each hash slot points to one data page.

Records are looked up in the following way: the record key is hashed. If the size of the hash table is 2^k, k lowest bits of the hash value are used to locate an entry in the hash table. This entry points to that data page. The record is found by sequentially scanning this page.

Initially, a file is created with one data page and hash page with 1 entry, pointing to a data page. When records are added to the file, they are added to the appropriate data page corresponding to their hash value. When the data page is full, it is split into two pages and pointers in the hash table pointing to the previous table are changed to point to the new pages. When there is only one pointer to the page and the page is split, the hash table size is doubled and a new hash table is created.

In a filesystem, it is not possible to allocate a continuous hash table of arbitrary size. Thus, GFS and ZFS directories are encapsulated in files and use file allocation information to map logical blocks in the file to physical blocks on disks. The layout of file allocation information will be described in section 5. The access to a directory entry thus doesn't require just two disk accesses (as described by Fagin); additional accesses to file allocation structures to map logical blocks to physical disk blocks are needed.

ZFS stores the whole directory in a file, whereas GFS stores only the hash table in a file and the data pages are stored in standalone blocks.

4.2 Directories in the Spad filesystem

In the Spad filesystem, I did not use B-trees or B+ trees. I decided to use an algorithm based on extendible hashing. The main reason is the simplicity of the implementation.

Moreover, extendible hashing requires fewer disk accesses than B-trees (see section 10.2.6).

If files do not have hard links, the file structure (called Fnode) is stored directly in the directory to save one disk seek when accessing the file. This approach has been presented and evaluated in [38].

4.2.1 Modification of the algorithm for the Spad filesystem

I solved the problem with a long linear table differently from GFS and ZFS. I decided to not encapsulate directories in files; I use a radix tree instead of a linear table. The size of a node can be set when the filesystem is created; the default is 2048 entries. Each entry has 6 bytes.

A radix tree points to data pages with variable-size records. Each record specifies one file and its size is dependent on the file name length and amount of extended attributes. When the data page fills-up and more records need to be added, the page is split and pointers from the tree are changed. When there was only one pointer to the data page, extendible hashing would double the hash size — however this is hard to do in a filesystem — so my implementation instead allocates the next radix tree node at a lower level with all pointers pointing to the data page to be split and then splits the page.

Extendible hashing has another problem: When collisions in a hash function happen so that too many records' keys hash to the same value, extendible hashing can't add the records into the database — it will try to extend the hash table again and again, running out of disk space. I solved this problem by adding emergency chain pointers to the data pages — when a data page can't be split, a new data page is allocated and a pointer to the new page is added to the previous page. When such excessive hash collisions happen, the filesystem will allocate three radix tree nodes (assuming a default node size of 2048 entries and hash size of 32 bits), then it finds out that it can't split the data page anymore, so it allocates a new data page and links it to the previous one. If more collisions occur, data pages can form a chain of arbitrary length — this degrades performance to a linear search, but at least keeps the filesystem operating under any circumstances (note that most other filesystems will also degrade to a linear search under extreme collisions; ReiserFS will refuse to create more files in this case. The only filesystem that has guaranteed $O(\log n)$ directory lookup complexity is the HPFS).

An example of a directory on the Spad filesystem is displayed in figure 4.2.

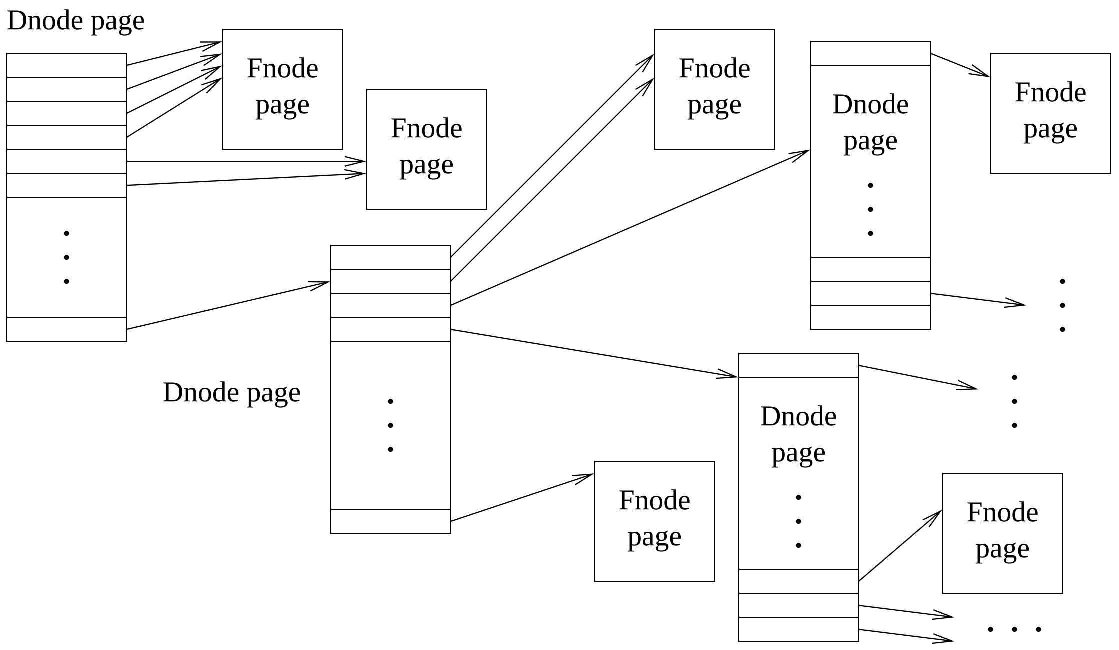

Figure 4.2: Directories in the Spad filesystem

4.2.2 Description of the algorithm for managing directories in the Spad filesystem

This is the algorithm for directory entry lookup. Inner tree nodes are called *dnode pages*. Each dnode page contains an array of pointers to other blocks on disks. Leaf pages containing directory entries are called *fnode pages*. An fnode page contains a variable number of directory entries and optionally an additional pointer to the next and previous chained fnode page (if the chains are used, due to excessive hash collisions).

- 1. Hash the filename
- 2. Read the root directory page
- 3. If it is a dnode page, take the least significant bits of the hash and take the pointer at this index in the dnode page. Shift the hash right by this number of bits. Read the page and go to step 3.
- 4. If it is not an fnode page, report corrupted directory.
- 5. Scan entries in the fnode page and look for the one with the given filename. If found, exit and report success.
- 6. If the fnode page contains a next chain pointer, follow the pointer and go to step 4.
- 7. Exit and report directory entry not found.

Dnode pages have a structure similar to a trie. Step 3. represents walking the trie. The steps 4.–7. represent walking the linear list that may be formed at the tail pointers.

This is the algorithm for adding a directory entry:

- 1. Hash the filename
- 2. Read the root directory page
- 3. If it is a dnode page, take the least significant bits of the hash and take the pointer at this index in the dnode page. Shift the hash right by this number of bits. Read the page. Go to step 3.
- 4. If it is not a fnode page, report corrupted directory.
- 5. Scan the fnode page for free space, if a continuous block of the appropriate size is found, insert the new entry there and exit.
- 6. If the fnode page contains a next chain pointer, follow the pointer and go to step 4.
- 7. If this is the root fnode page (i.e. no dnode pages were read in step 3) and the fnode page is smaller than the maximum fnode page size, allocate a new larger fnode page, move existing entries there, insert the new entry into a free space and exit.
- 8. If some chain pointers were walked in step 6, allocate a new fnode page and link it to the previous fnode page's chain pointer. Insert the entry into the new fnode page and exit.
- 9. If all hash bits were exhausted in step 3, go to step 8 (i.e. allocate a new fnode page and add it to the chain).
- 10. If this is the root fnode page or if the parent dnode contains only one pointer to this fnode page, allocate a new dnode with all pointers pointing to the fnode page we are processing and change the pointer from the upper level (either a parent dnode page or the directory's root pointer) to point to the new dnode. If the allocation failed, allocate a new fnode page (as in step 8.) and link it to the previous fnode page's chain pointer, insert the entry into the new fnode page and exit.
- 11. Allocate two new fnode pages and split the fnode contents as described by Fagin [20]. Change pointers in the parent dnode page.
- 12. Go to step 5. If not enough space is found, repeat the steps after step 5 — this will eventually do another splitting.

Steps 1.–6. are the same as in the find algorithm. They find the right page where to place the new entry. If there is some space in the page, the new entry is added there without modifications to the rest of the structure.

In step 7, we check if we have only the root fnode page and if we can extend it. For small directories (up to a configurable limit, maximum fnode page size), it is more efficient to have the directory in s linear page and scan it linearly.

In step 8, we check if chaining is already used. If it is, we must create the next chained fnode and place the new entry in it.

In step 9, we check if we already used all hash bits. In this case we cannot split further so use chaining.

In step 10, we check if there is nothing to split (there are either no dnode pages or the last bit of the dnode page is already used). If this is the case, we create a new dnode page with all the pointers to the same fnode. The dnode page will be split in the next step. If the dnode page already has some common pointers to split, this step is skipped. In step 11, we split the fnode page and update the pointers in the dnode. Then we continue the process and attempt to add the new entry into one of the new fnodes. If the fnode is still full, this splitting is repeated.

This is the algorithm for deleting a directory entry:

- 1. Find the entry as described in the "lookup" algorithm.
- 2. Delete the entry.

These two steps would be sufficient to perform deleting; however, the directory can be shrunk so that it doesn't occupy too much space. The steps for shrinking a directory follow:

- 3. If the fnode page is not empty or it contains a chain pointer to a next fnode page, exit, shrinking is not possible.
- 4. If the fnode page contains a chain pointer to a previous fnode page, delete the fnode page, delete the next chain pointer of the previous fnode page, read the previous fnode page and go to step 3.
- 5. If this is the root fnode page (i.e. there are no dnodes in the directory), exit.
- 6. Delete this fnode page, set the pointer in the dnode pointing to it to zero.
- 7. If the dnode contains only zero pointers and it is not a root dnode, delete the dnode, set pointer in the parent dnode to zero and go to step 7.

In steps 3 and 4 we delete empty pages from the end of the chain. We maintain the chain in such a state that the last page of the chain has at least allocated entry. If the last page becomes empty, we delete the last page and repeat the check for the previous page. If we have only one fnode page (checked in step 5) we exit; the directory must contain at least one page, so we tolerate an empty page in this case.
In step 6 we delete an empty fnode page and clear the pointer from the dnode. If the dnode contains all zero pointers, the dnode is deleted as well; we check the upper dnode and delete it again if it becomes empty.

4.3 Summary

In table 4.1 you can see a summary of directory organization methods in filesystems. Filesystems used in the Unix world have case-sensitive file names; filesystems used on some other operating systems (DOS, Windows, OS/2, VMS) have case-insensitive file names. Filesystems used in both Unix and non-Unix environments have optional case-sensitivity.

The indexing method is most often a B-tree or a B+ tree — except for the GFS, ZFS and the Spad filesystem that use modified extendible hashing [20]. Some filesystems use the filename as an indexing key; some hash the filename first and use the hashed value as a key.

Directory structures may be either encapsulated in files or placed as stand-alone structures on the filesystems. Placing directories in files reduces the performance somewhat — each time when following one B+ tree pointer, a few additional accesses are required to translate the logical file pointer to the physical sector number (the translation is done as described in the following section — File allocation information — and it may involve another B+ tree walk). Stand-alone directories contain physical sector numbers directly in them and thus are faster — one access to a directory structure equals one access to a cache or physical disk. GFS has its directories partially in file — the hash table is managed as a file and data blocks are standalone.

The "complexity" field shows time complexity of directory operations. Note that some filesystems have directories placed inside files, so their complexity is combined file and directory access complexity. If files are accessed with $O(\log n)$ and directories are organized as trees placed inside files, then directory access complexity is $O(\log \log n)$.

	Case sensitive	Indexing method	Indexing key	Placement of directory content	Complexity
FAT	No	none	n/a	In file	$O(n)$
HPFS	No	B-tree	File name	Stand-alone	$O(\log n)$
NTFS	Optional	B-tree	File name	In MFT records	$O(\log n)$
FFS	Yes	none	n/a	In file	$O(n)$
Ext2	Yes	none	n/a	In file	$O(n)$
Ext3	Yes	B+ tree	Hash of name	In file	$O(\log \log n)$
Ext4	Yes	B+ tree	Hash of name	In file	$O(\log \log n)$
ReiserFS	Yes	B+ tree	Hash of name	In global tree	$O(\log n)$
Reiser4	Yes	B+ tree	Hash of name	In global tree	$O(\log n)$
JFS	Optional	B+ tree	First 30 characters	In file	$O(\log \log n)$
XFS	Yes	B+ tree	Hash of name	In file	$O(\log \log n)$
ZFS	Yes	Ext. hashing	Hash of name	In file	$O(\log \log n)$
GFS	Yes	Ext. hashing	Hash of name	Partially in file	$O(\log \log n)$
SpadFS	Optional	Ext. hashing	Hash of name	Stand-alone	$O(\log n)$

Table 4.1: Directory organization in filesystems

Benchmarks of directory efficiency in filesystems are shown in section 10.2.6.

5. File allocation information

The purpose of the file allocation information subsystem is to keep track of blocks allocated to a specific file. The subsystem has to translate a logical block number (the block number within a file, starting with 0 up to $(file\_size + block\_size - 1)/block\_size)$) to a physical block number (the block number on a partition). This subsystem must also handle adding blocks (extending file) and removing blocks from the tail (truncating file). These operations should be fast, because they significantly affect the speed and CPU consumption of reading and writing large files: translating logical to physical block number affects CPU consumption when reading files and when writing inside existing files. Extending affects CPU consumption when writing at the end of the file.

Some filesystems support sparse files — a sparse file is a file that is missing allocation information for some logical blocks. The operating system kernel fills these blocks with zero when userspace program reads them. When some program writes to these blocks, the blocks are allocated on the disk. On Unix, you can create sparse files when you move a file position with `lseek` syscall past the end of the file and write to the file. The blocks from the end of the file to the position where you write will be sparse. Another way to create a sparse file is to use `truncate` or `ftruncate` syscall to extend the file size. There is a serious inconvenience in the Unix API — it can't create holes in existing files. It can only create holes at the file end with the two methods described above. Because of this, sparse files are rarely used by applications. For example, when a database application deletes a page in the middle of a file, it would be natural to delete the data from disk and make the file sparse; however, there is no syscall that do it.

5.1 Methods in existing filesystems

5.1.1 The FAT filesystem

The FAT filesystem (and its variants VFAT and FAT32) uses the most inefficient method for maintaining allocation information. The disk has a FAT (file allocation table) that contains a value for each cluster[1] on the filesystem. The value can be a pointer to the next cluster in a file, or three special values: End-of-file, Free-space, Bad-cluster.

File allocation information is stored in the following way: directory entry of a file contains a pointer to the first cluster. The second cluster number is stored in the FAT at the position of the first cluster. The third cluster number is stored in the FAT at the position of the second cluster. And so on up to the end of a file. The last cluster contains an End-of-file mark in the FAT. See figure 5.1. The length of the cluster chain in the FAT must be consistent with the size of the file in its directory entry.

This method is very inefficient because when logical cluster n needs to be accessed, the filesystem driver has to perform n lookups in the FAT. At best, the clusters in a

[1] A cluster is a group of sectors of fixed size used for allocation purposes. In Unix terminology, the term *block* is used instead.

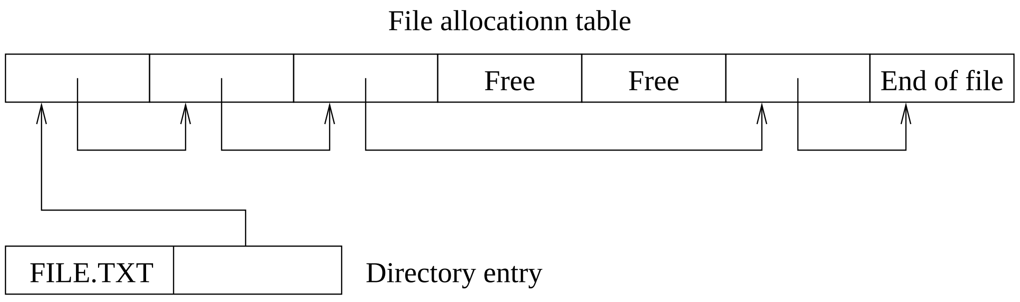

Figure 5.1: Chain of pointers in file allocation table

chain will be near each other, so that the search will utilize a buffer cache and will only consume CPU. At worst those n lookups will perform disk requests, slowing the whole operation seriously.

5.1.2 Traditional Unix filesystem

This method was developed in the traditional Unix filesystem and is still in use today in filesystems such as Ext2 [11] and Ext3 on Linux or FFS [7] on BSD.

Each file is represented by an inode structure. An inode contains file information and physical block numbers of the first 12 blocks. It also contains three pointers to an indirect blocks of depth 1, 2 and 3. The first pointer points to indirect block of depth 1, and this indirect block contains pointers to blocks starting with logical number 12. The number of pointers in the indirect block depends on the block size — usually one pointer takes 4 bytes. If all pointers in this block are exhausted, an indirect block of depth 2 is allocated — it contains pointers to other indirect blocks of depth 1 and they contain pointers to data blocks. Finally, for large files, an indirect block of depth 3 can be allocated. The structure of direct and indirect blocks is shown in figure 5.2.

Thus, direct blocks can map 12 blocks; the 1st indirect block can map $blocksize/4$ blocks; the 2nd indirect block can map $(blocksize/4)^2$ blocks and the 3rd indirect block can map $(blocksize/4)^3$ blocks. For blocksize of 4096 bytes, this can support a maximum file size of 4TB.

The traditional Unix filesystem supports sparse files — pointers to direct or indirect blocks can contain a zero value and the filesystem driver interprets it as information that the corresponding location has no disk block assigned. When a userspace program is attempting to read these parts of the file, zeros are returned.

The traditional Unix filesystem has much more efficient file allocation information than the FAT filesystem. To access any part of a file, four disk accesses at most are required: reading at most three indirect blocks and the data block itself. If you access blocks with similar logical block numbers (for example during a sequential read), the accesses to indirect blocks will be cached, and only one disk access is required.

However, this method can still be optimized: files are usually allocated sequentially: if logical block a is allocated at physical block x, then logical block $a+1$ is usually allocated at physical block $x+1$, logical block $a+n$ is usually allocated at physical block $x+n$, etc. To store allocation information for n consecutive blocks, the traditional Unix filesystem

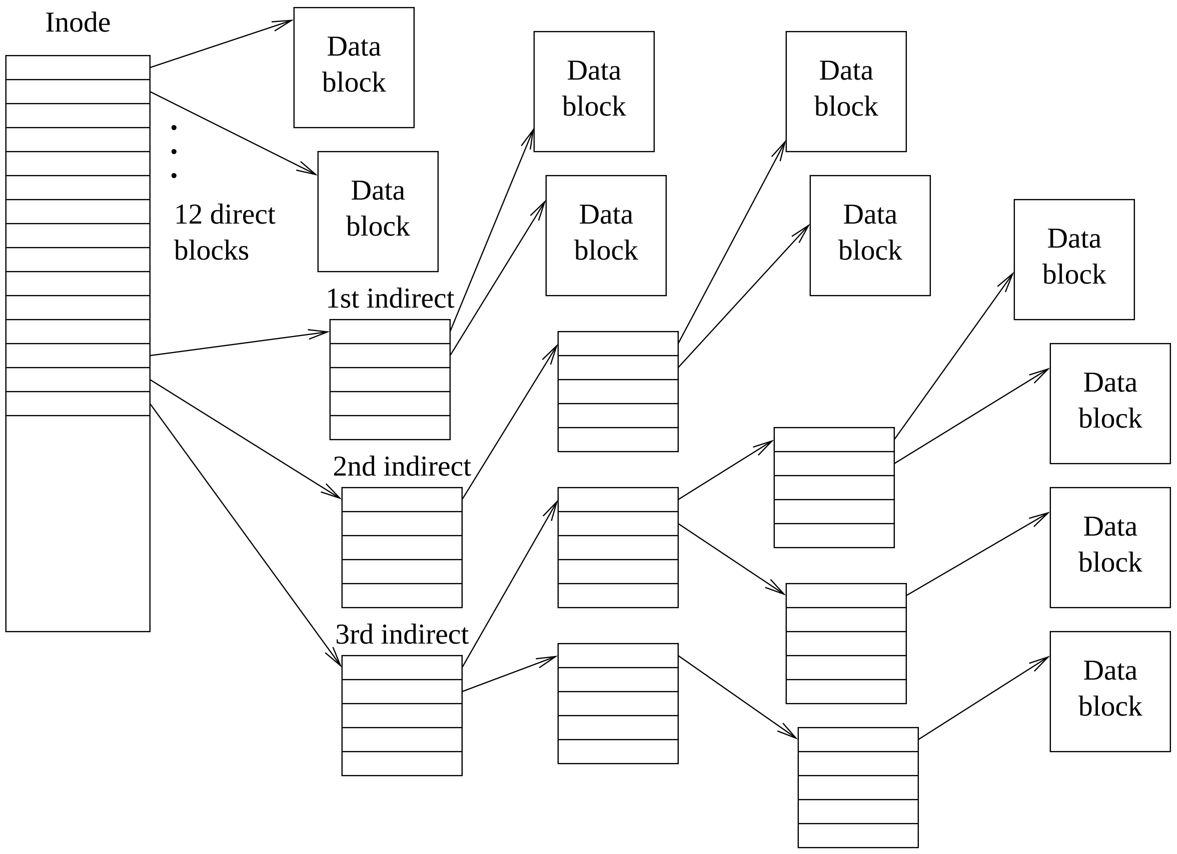

Figure 5.2: Direct and indirect blocks of 3 levels on traditional Unix filesystem

requires n pointers, while in fact we could store only three pointers: logical block number (a), physical block number (x) and number of sequential block (n). One such consecutive sequence of blocks is called a *run* or *extent*. Advanced filesystems use extent-based file allocation information — thus allocation information takes less disk space and lookups of blocks are faster.

5.1.3 Radix tree of fixed size depth: ZFS, GFS

ZFS and GFS filesystems use direct and indirect blocks containing a list of blocks too. The difference from the traditional Unix filesystem is that in ZFS and GFS all indirect blocks have the same depth.

Initially, the inode contains a list of data blocks. When there are too many data blocks allocated so that they don't fit into the inode, the pointers are moved to the 1st level indirect block and a pointer to this indirect block is written to the inode. When more data are allocated, more 1st level indirect blocks are created in the inode. When 1st level indirect blocks fill the inode, the pointers are moved to the 2nd level indirect block and the inode contains one pointer to this block, and so on. The indirection level can increase as the file grows. This data structure is called a *radix tree*.

The inode contains the number describing, the indirection level of its blocks.

ZFS combines a filesystem and volume manager into one package — so its block pointers are not simple pointers; each block pointer is a 128-byte structure containing at most 3 pointers to data replicated on 3 distinct disks and a 32-byte checksum [17]. If the disk returns invalid data without notifying the operating system about an error, ZFS finds that the data is invalid and reads it from a different disk.

5.1.4 Extent-based allocation information: HPFS, JFS, XFS

An extent [3] is a triplet of these values: logical block number, physical block number, run length. Some way to organize extents in a file must be defined. The most common way to organize extents is to use a B-tree or a B+ tree.

The HPFS stores files in a B+ tree. A leaf node has 12 bytes (triplet: logical sector, physical sector, length). An inner node has 8 bytes (pair: logical sector, pointer to lower tree level). One tree page has 512 bytes and can contain 40 leaf nodes or 60 inner nodes. There can be 8 leaf nodes or 12 inner nodes directory in a fnode[2]. This structure could allow sparse files; however they are not created by the OS/2 operating system. The structure is always extended or truncated on the end — it simplifies its manipulation. An example of a B+ tree on the HPFS filesystem is displayed in figure 5.3.

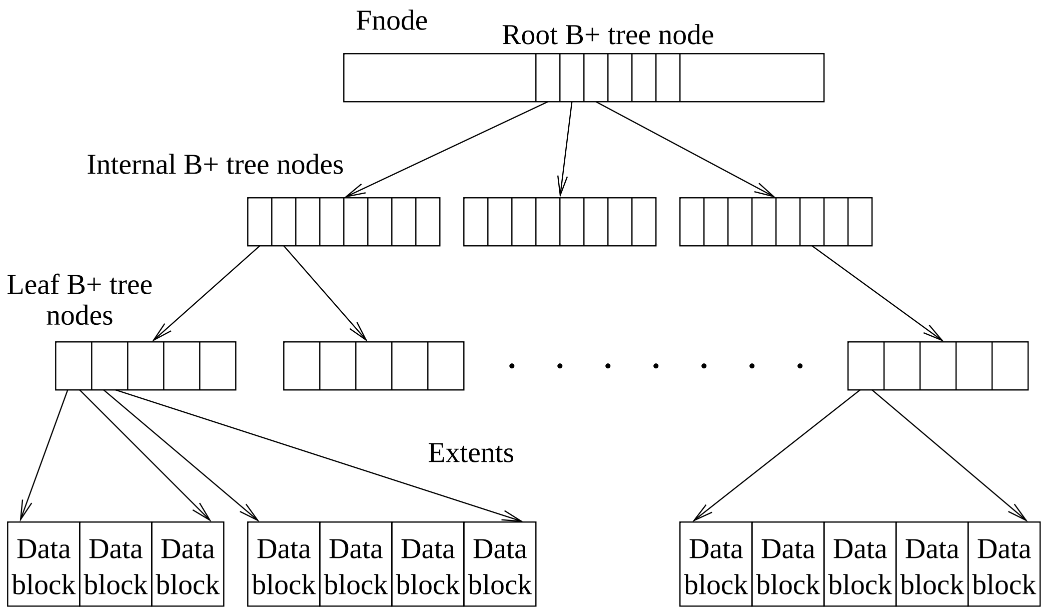

Figure 5.3: B+ tree containing extents on the HPFS filesystem.

The JFS uses B+ trees, too. It encapsulates all on-disk structures (not only files, like the HPFS) into the same structure in the form of a B+ tree. It is questionable, if encapsulating non-file structures in B+ trees is good or not — it simplifies some operations and complicates others. A tree node has 16 bytes: 5 bytes for logical and physical block number, 3 bytes for length, and the rest for flags. One tree node has 4kB.

[2] Equivalent of inode on Unix.

The XFS uses a B+ tree too, in a similar way to the JFS (also with a 16 byte node). The XFS B+ tree code is general enough that it is also used for space allocation trees (to be described in section 6.2.3), only with a different node length.

5.1.5 ReiserFS

ReiserFS uses a different approach. As I described in section 4.1.4, the whole ReiserFS filesystem is composed of one B+ tree. There are no separate trees for each file and directory, like in most other filesystems. Indirect tree items are used to store file allocation information. The item is indexed by the file's directory ID and object ID and the offset in the file. Because directory ID has precedence over file ID and file ID has precedence over the offset, the filesystem tends to place indirect items for the same file near each other in the tree and indirect items for files in the same directory near each other. The structure of the filesystem can be seen in figure 5.4.

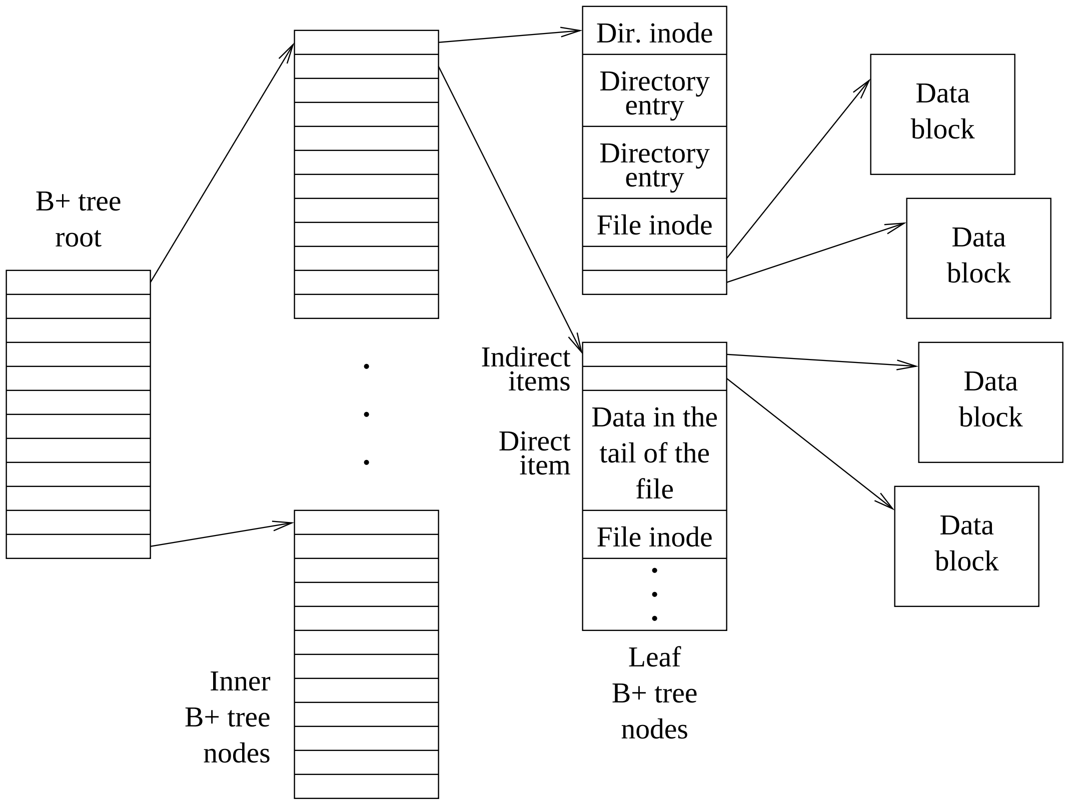

Figure 5.4: The structure of ReiserFS. The whole filesystem is just one B+ tree.

There is a bad design deficiency in ReiserFS — indirect tree items are actually lists of blocks, not lists of extents. ReiserFS doesn't use extents at all and stores information for each file block-by-block. For a large file, a list of all the file's blocks consumes significant space in the tree while it could be simply compressed into extent triplets. There is another performance problem: the indirect tree entry is not cached anywhere in memory; thus when accessing each file block, the whole filesystem tree is walked. The walk is usually cached; thus no disk accesses are done, but walking the tree in memory consumes CPU time.

This deficiency was fixed in the Reiser4 filesystem — it has extents just like most other filesystems.

5.2 File allocation information in the Spad filesystem

Storage of file allocation information has to be simple and fast. I decided not to use B-trees or B+ trees, because their implementation is complex. I use extents because they are the most efficient of all known methods for storing file allocation information. I do not store extents in a B-tree, but in a structure similar to direct/indirect blocks on the traditional Unix filesystem. I decided not to support sparse files, because they are rarely used and are not required for the function of any high-performance application.

The first two extents are stored directly in a *fnode*[3] located in the directory entry. All numbers in the Spad filesystem are in sectors, not blocks (thus, if blocksize is larger than 512, the numbers need to be divisible by (blocksize / 512)). Because of the limited size of a *fnode*, these extents have a 48-bit physical sector number and 16-bit run length. There is no need to store a logical sector number, because for the first extent the logical sector number is always 0 and for the second extent, the logical sector number is the run length of the first extent.

When I decided how much allocation information to store directly in the *fnode*, I had to make a choice between storing too much information (fast access to files, but big fnode size implying slow scan of directories) and too little information (slow access to files, smaller fnode size and faster scan of directories). I chose to store two extents, each having maximum of $2^{16} - 1$ sectors as a balance between these two objectives.

If more than two extents are needed, a structure *anode* is allocated. Its size is 512 bytes and it contains 30 extents — 20 direct extents and 10 indirect pointers of depth advancing from 1 to 10 — similar to the traditional Unix filesystem (see figure 5.5). Each extent has 16 bytes and it is a pair of two 64-bits values (`blk`, `end_off`)[4]. `blk` is the physical sector number and `end_off` is the logical sector number + run length. Note that the logical sector number for a given extent is the `end_off` value of a previous extent and run length of a given extent can be calculated by the difference of `end_off` values of this and the previous extent.

The anode has 512 bytes, because 512 is the lowest sector size used in common disks and it is the lowest sector size supported by SpadFS. It contains 10 indirect pointers to support the worst-case fragmentation on the largest possible device. The largest supported device size is 2^{48} sectors; thus a file can have at most 2^{48} fragments. Each indirect block has 30 extents. An indirect tree of level 9 can map at most 30^9 extents; an indirect tree of level 10 can map at most 30^{10} extents. $30^9 < 2^{48} < 30^{10}$ — from this inequation we see that 10 is the lowest depth that can support the worst case fragmentation scenario.

[3] *Fnode* is the structure representing file or directory on the Spad filesystem. It is equivalent to Unix inode.

[4] See `struct extent` in file `STRUCT.H` in the sources.

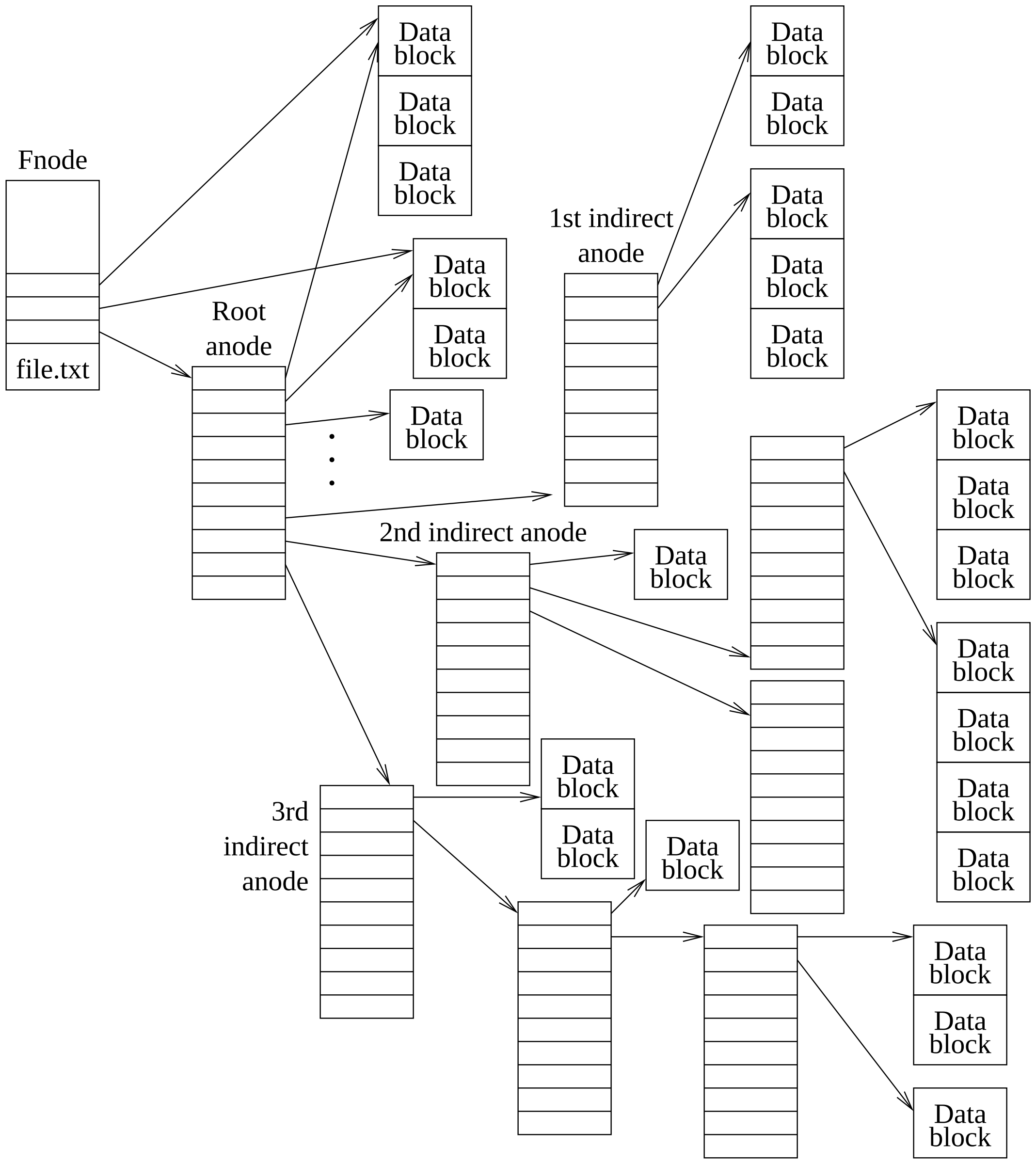

Figure 5.5: File allocation information in the Spad filesystem

For indirect pointers, `blk` is a pointer to the descendant anode and `end_off` is the last logical sector stored in this anode plus 1. The definition of `end_off` for direct extents might seem counterintuitive — but it makes the code slightly simpler when defined this way.

There is one difference from the traditional Unix filesystem in indirect block format — indirect blocks of a depth greater than 1 on the Spad filesystem always contain one direct pointer and 30 indirect pointers (while on the traditional Unix filesystem indirect blocks of a depth greater than 1 contain all indirect pointers). The reason for this oddness is this — if a delayed allocation is implemented, the filesystem needs to do reservation for allocated space and make sure that it doesn't return success to a `write` syscall if it could later happen that there is no space for the data. This is accomplished by maintaining an upper boundary for the number of blocks that may be consumed by delayed allocations and making sure that this upper boundary is never lower than the free space on the filesystem. If the indirect blocks contain at least one direct pointer, the upper boundary can be defined: one logical block allocates at most two disk blocks. If indirect blocks contained only indirect pointers, such a statement could not be made.

An overall example of the file allocation information structures can be seen in figure 5.5. Note that the first two extents are pointed to by both *fnode* and *anode* — the reason is an improved reliability. If the fnode is destroyed, full file content can still be recovered using only the tree of anodes.

5.2.1 Algorithms for managing file allocation information

This is the algorithm for lookup of logical-to-physical sector mapping.

- 1. If the logical sector is within the first two extents embedded in a fnode, return the mapping directly without any disk access.
- 2. Set local variables `depth_now = 0` and `depth_total = 0`. Set local variable `ano` to the root anode.
- 3. Find the number of direct entries, with the function `find_direct(depth_now, depth_total)`.
- 4. Read the anode pointed to by `ano`. Use binary search to find an anode entry corresponding to the desired logical block (function `find_in_anode`).
- 5. If the found block is direct (its index is less than the number of direct blocks), return the appropriate extent directly.
- 6. Set variable `ano` to point down the tree to the anode pointed to by this anode entry.
- 7. Call function `update_depth(&depth_now, &depth_total, anode_entry_index)` to update depth counts. This function will increase `depth_now` to denote that we are going to descend one level in a tree. If this was the root anode, the function will also store the total depth of the indirect block in `depth_total`.
- 8. Go to step 3.

In step 1 we resolve the fast case when no additional accesses are needed.
In step 2 we are initializing the tree walk. `ano` points to the current anode. Each position in the tree can be determined by two variables, the current depth from the root (`depth_now`) and total depth of the branch we are processing (`depth_total`). The number of direct entries in the anode (found in step 3) is determined by these two variables. The root anode (`depth_now == 0`) has 20 direct entries; the leaf anodes (`depth_now == depth_total`) have all 31 entries direct. The inner anodes have 1 direct entry and 30 indirect entries.
In step 4 we read the anode and find the appropriate entry with binary search.
In step 5 we test if the found block is in the direct entry; if it is, we return the extent directly.
In steps 6 and 7 we walk the tree down to the lower node and update `depth_now` and `depth_total`. The rule is: we always increase `depth_now` and if we are in the root entry, we set `depth_total` to the order of the indirect entry we are following (so that the first indirect entry will set `depth_total` to 1, the second indirect entry will set it to 2 and so

on). It should be noted that once we are in the leaf anode, all the entries are direct, so this branch is never executed and the algorithm is finite.

This is the algorithm for adding a new extent to a file.

- 1. If the fnode doesn't have the first extent, add the new extent directly to it and exit.
- 2. If the fnode doesn't have the second extent, add the new extent directly to it and exit.
- 3. If the fnode doesn't have a root anode, allocate a root anode and store the first two extents and the new extent in this new anode. Set fnode's anode pointer to it. Exit.
- 4. Set local variables `depth_now = 0` and `depth_total = 0`. Set local variable `ano` to root anode. Set local variable `ano_l` to −1 — this variable will denote the last anode, where we can add some extents.
- 5. Find the number of direct entries at this tree level with the function `find_direct(depth_now, depth_total)`.
- 6. Read the anode pointed to by `ano`.
- 7. If the number of entries in the anode is less than the number of direct entries at this level, add the new extent directly to the anode and exit.
- 8. If the number of entries in the anode is less than the maximum number of entries in the anode (31), set `ano_l = ano`. The purpose of this assignment is this: if we later find that the tree under this anode is full, we will return to `ano_l` and add new pointer.
- 9. If the number of entries in this anode is greater than the number of direct entries at this level (i.e. there is some indirect pointer), take the last indirect pointer, set `ano` to the sector it points to, update `depth_now` and `depth_total` and go to step 5.
- 10. Allocate a new anode and add a pointer to it to the anode pointed to by `ano_l`.
- 11. Add the new extent to the new anode and exit.

In steps 1–3 we resolve the simple case where there is no anode.
Next we will walk the tree. In step 4 we initialize the tree walk just like in step 2 of the lookup algorithm. We remember the additional variable, `ano_l`, the pointer to the last anode where a new indirect block can be added.
In step 5 we find the number of direct entries in the current node, the same way as in step 3 in the lookup algorithm. In step 6 we read the anode.
If there is a possibility to add a direct entry (step 7) to the anode, we add the new entry here and exit.
If there is a possibility to add an indirect entry (step 8), we record the anode to `ano_l` for later use.
In step 9 we walk down the tree. We want to find the last entry so we take the last indirect pointer, follow it and repeat the search. When we reach the leaf anode (we know this from `depth_now` and `depth_total`), there are no indirect entries, so this loop is finite.
If there is no indirect pointer we follow with steps 10 and 11. If we get here we are in an anode with all direct entries used. We can't add another direct entry, so we must add an indirect entry. If this anode has some available indirect entries, it was set as `ano_l` in step 8 — so we add the new entry to this anode. If this anode has no available indirect

entries, `ano_l` points to some other anode with free indirect entries, so we add our new entry there. If `ano_l` was never set (it contains -1) it means that the anode tree is full — this shouldn't happen because the tree is large enough to cover the worst possible fragmentation resulting in 2^{48} extents.
We create a new anode and link it to the appropriate indirect entry, note that each anode has at least one direct entry, so we can add the new extent directly to the anode.

Truncating a file: Because it is needed to keep file allocation information consistent with free space information, we cannot just truncate an anode tree — in the case of a crash before commit, truncated data could be needed. A truncate is performed in the following way: the anode tree is not changed in any way; anodes mapping only the truncated area are freed and file size is lowered — file size is protected with (crash count, transaction count) pair; thus in the case of a crash before commit, the old size will be used.

After a truncate, the anode tree maps a larger part than the file size, but it doesn't matter — only a part of the anode tree that maps valid data within the file size will be used. The rest of the anode tree mapping space past the file size is marked as free in file allocation structures, and these sectors can be reused for other purposes after the next *sync*.

5.3 Summary

Table 5.1 shows a summary of features of various filesystems. Extents and methods of their organization have been discussed in previous chapters.

Another important feature is how the filesystem handles small files. On non-optimized filesystems a small file consumes one allocation block that usually has a 4kB size. Some filesystems (HPFS, SpadFS) can use the smallest possible sector size — 512 bytes; thus small files consume only this space (on HPFS it actually consumes 1024 bytes — one sector for data and one for the FNODE structure describing the file). The FFS stores small files in fragments (fragment size can be selected when creating the filesystem); the minimum fragment size is 512 bytes. The NTFS actually attempts to do small file optimization by storing the data of small files directly in MFT record (equivalent to Unix inode); however, because the MFT record is quite large (usually 4kB), this optimization isn't very space-efficient. ReiserFS has the best optimization for small files — it stores them directly in the B+ tree with other metadata; thus small files consume only the exact space required.

Some filesystems support sparse files — files that have some areas not allocated on disk. This feature is rarely used.

The NTFS supports compressed files. There are some unofficial patches for compressed files on Ext2 on old Linux kernels. Reiser4 also supports compressed files, but this feature is not yet considered ready for production use.

Benchmarks of filesystems with respect to file allocation information are shown in sections 10.2.4, 10.2.5 and 10.2.8.

	Has extents	Organization method	Complexity	Optimized for small files	Sparse files	Supports compression
FAT	No	Chain of pointers	$O(n)$	No	No	No
HPFS	Yes	Per-file B+ tree	$O(\log n)$	512-blocksize	No	No
NTFS	Yes	Linear list of extents	$O(n)$	Files in MFT	Yes	Yes
FFS	No	Direct/indirect blocks	$O(\log n)$	Fragments	Yes	No
Ext2	No	Direct/indirect blocks	$O(\log n)$	No	Yes	Unofficial
Ext3	No	Direct/indirect blocks	$O(\log n)$	No	Yes	No
Ext4	Optional	Direct/indirect blocks	$O(\log n)$	No	Yes	No
ReiserFS	No	Global B+ tree	$O(\log n)$	Yes	Yes	No
Reiser4	Yes	Global B+ tree	$O(\log n)$	Yes	Yes	Unofficial
JFS	Yes	Per-file B+ tree	$O(\log n)$	No	Yes	No
XFS	Yes	Per-file B+ tree	$O(\log n)$	No	Yes	No
ZFS	No	Radix tree	$O(\log n)$	No	Yes	Yes
GFS	No	Radix tree	$O(\log n)$	No	Yes	No
SpadFS	Yes	Direct/indirect blocks	$O(\log n)$	512-blocksize	No	No

Table 5.1: File allocation information in filesystems

6. Block and inode allocation subsystem

In this chapter I will describe the subsystem of the filesystem that keeps information about which blocks are allocated and which are free. The purpose of the allocation subsystem is to process requests to allocate and free blocks on a partition. To minimize file fragmentation the allocator should be able to allocate large continuous sector runs, and to minimize seek time when fragmentation happens the allocator should be able to allocate blocks near a given block.

The block allocation subsystem should process the following requests:

- Allocate n blocks near a specific block (the exact allocation strategy — selection of this "specific block" will be discussed in the next chapter).
- Allocate up to n blocks at a specific location.
- Free a given number of blocks.

The first request type is used when creating a new file or when creating a new extent in a file. The second request type is used when extending an existing file. The third request type is used when deleting or truncating a file.

6.1 The relationship between data block allocator and inode allocator

6.1.1 Separate space for data and inode blocks

Some filesystems have separated space for data blocks and inodes and so they have two allocators — one for data and one for inodes. This has the disadvantage that the system administrator has to know an estimated number of files in advance (when installing the operating system and creating the filesystem). If he overestimates the number of files, too much space will be occupied by unused inodes and this space can not be used for regular data. If he underestimates the number of files, the filesystem will reject the creation of new files even if there is plenty of free space — because it has no free inodes. The traditional Unix filesystem (and its implementations — ExtFS, Ext2, Ext3, FFS) fall into this category.

6.1.2 Extendible inode space

Newer filesystems solved the problem of running out of inode space by maintaining space reserved for inodes and allowing it to grow. When the filesystem's inode allocator has no free space and the block allocator has some free space and when a request to allocate a new inode is made, the inode allocator allocates some space from the block allocator and extends its inode space. This new space will be permanently marked as "allocated" in the block allocator and the inode allocator manages its map of free/allocated inodes within it. This has the advantage that the filesystem never refuses to create a file if it has some free space. The disadvantage is a bit problematic freeing of this new inode space back into the block allocator and an increased code complexity because it still has two allocators. This method is implemented in the JFS, XFS and NTFS[1].

6.1.3 One allocation space used for both data and inodes

There is a third possibility: use just one allocation space with one allocator and allocate both data blocks and inodes from it. This method is used by HPFS, ReiserFS and the Spad filesystem. This method simplifies the design but has a disadvantage too: one inode consumes the whole block. The previous two schemes allowed a block to be used for several inodes.

In the HPFS this disadvantage is mitigated by using the smallest block size possible — 512 bytes. This solution is not possible on Linux or Unix, because these systems' caches handle small blocks very inefficiently and consume too much CPU when operating with this low block size. On the OS/2 each filesystem driver allocates and manages its own cache and so the HPFS cache can handle 512 byte blocks fine.

In ReiserFS this problem doesn't really exist because inodes are part of the tree and one block can be shared by many objects of any type.

In SpadFS, files are embedded directly in the directories; thus they don't need any "inode" — the inode is just a part of the directory. Ganger and Kaashoek have shown [39] that such design improves performance for small files. A block with an inode[2] is allocated only when a hardlink is created — but because hardlinks are created rarely, this doesn't really impact space consumption. A block with a list or tree of extents (anode) is allocated when the file has more than two extents — however, such files are likely large anyway; thus consumption of a single block for a list of extents won't matter much. The important fact is that small files consume no other space than their data blocks.

6.2 Allocation methods in existing filesystems

6.2.1 Lists of free blocks in ExtFS

ExtFS (an ancestor of Ext2 on Linux) implemented free space management with lists of free blocks. Lists have the advantage that they require $O(1)$ time to allocate and free a block.

Lists are implemented in the following way: There are two types of free blocks — free blocks *with the list* and free blocks *without the list*. The content of the free blocks *without the list* is undefined. A superblock has a pointer to one free block *with the list*. This block has 254 optional pointers to other free blocks *without the list* and one optional pointer to another free block *with the list*. Thus, free blocks *with the list* form a single

[1] In the NTFS terminology, the inode space is called the Master File Table.

[2] Called "fixed fnode block" in SpadFS terminology.

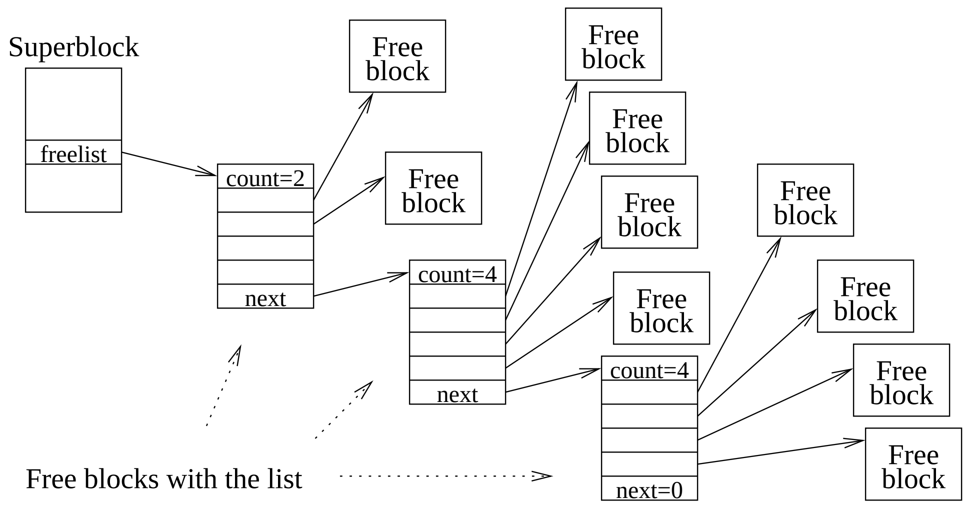

Figure 6.1: Free space management on Ext filesystem

linked list and each of them has at most 254 pointers to other free blocks *without the list*. See figure 6.1.

A block is freed in the following way: if the head of the list exists and some of its 254 pointers are free, the block number is stored in it. Otherwise the freed block is used as a new head of the list; all its 254 pointers are erased and it is linked to the previous head of the list.

A block is allocated in the following way: if the head of a list has some of its 254 pointers used, one pointer is taken and returned as the newly allocated block. If all of its pointers are unused, the head of the list is returned as the newly allocated block and its link pointer is used as a new head of the list.

The list of free inodes is managed in the same way, except that an inode contains only 14 pointers to other free inodes and one pointer to the next inode in the list.

Despite having $O(1)$ complexity, this implementation has very bad performance. Initially, after filesystem creation, the lists are sorted; thus the allocator returns continuous runs and file fragmentation is low. However, after some time of usage, blocks on the list are no longer sorted and the allocation algorithm returns non-continuous blocks scattered randomly over the whole partition. Even if the partition has plenty of continuous free space, the allocator is unable to return continuous blocks if the list is not sorted.

These problems led to the abandonment of free block lists altogether in all newer filesystems. Free block lists were replaced with bitmaps — they have lower theoretical complexity but they can be used to search for continuous free areas on the disk.

6.2.2 Bitmaps

Bitmaps are implemented in most current filesystems. A bitmap is an area of a filesystem that has one bit for each block of the filesystem. If the bit is zero, the block is free; if the bit is one the block is allocated (or vice versa).

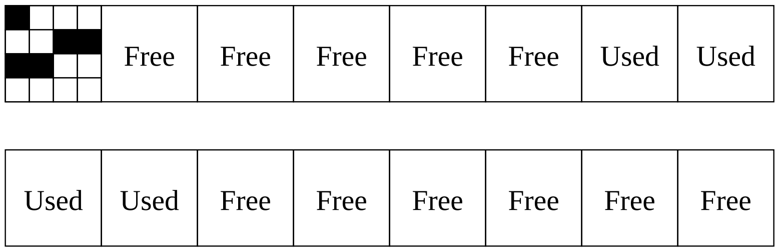

Figure 6.2: An example of bitmap for 16 blocks.
Note that the bitmap block itself is marked as allocated in the bitmap.

When the allocator needs to allocate a continuous area of a specified size near a specified block, it starts scanning the bitmap from a given block, searching for a run of bits.

We can see that, unlike lists, a bitmap allocator can prevent file fragmentation by returning a continuous space of a given size and it can minimize seek time because it can allocate space near a given block. The exact algorithms (which block to select to allocate near and what size to) will be discussed in the next chapter.

There is one downside of the bitmap allocator — it has bad time complexity; at worst, its complexity is $O(n)$ where n is the number of blocks on a partition. To mitigate this problem, filesystems usually subdivide bitmaps into groups and maintain a count of free blocks for each group separately. If some group's count is zero, the allocator does not need to scan bitmaps pertaining to this group. This can reduce the complexity of allocating a single block to $O(\sqrt{n})$ — the allocator scans at most $\sqrt{n}$ counters of free blocks in groups and when it finds a non-zero counter it scans at most $\sqrt{n}$ bits in a group's bitmap. This complexity seems to be acceptable in practice. Note that in most common cases, the starting block is selected with such heuristics that it is free or there are some free blocks near it, thus reducing the complexity to a constant.

In some implementations, the allocator also maintains an upper boundary on the length of the longest continuous block run in each group, allowing it to quickly skip groups that have enough free space but contain only short continuous runs. Maintaining the exact length of the longest continuous run would be very time consuming — it would require a scan the whole group after each change and it would likely consume more CPU time than it would save. However, maintaining the upper bound is cheap — when the group is scanned for a continuous run of x blocks the allocator can meanwhile compute the length of the maximal free block run — y. If the run of x free blocks is not found, y is stored in the group's in-memory descriptor and the whole group is skipped in further allocations requesting more continuous blocks than y. Obviously, the upper bound must be reset or increased when freeing blocks in a group.

6.2.2.1. Binary buddy bitmaps in the JFS

To avoid a linear bitmap scan, the JFS uses a binary buddy system on the top of its bitmaps [37]. This system tracks only runs that have a length of a power of two and are aligned on their length. For each word (32 bits) in the bitmap, the filesystem maintains a value representing $\log_2$ of the length of the longest aligned run within this word. For two adjacent words, the value of the longest run within them may be computed from the values of each word: if both words have value 5 (i.e. all 32 bits free), the doubleword has value 6 (i.e. it has a free run of a length of 64 blocks); otherwise, the value for a doubleword is the maximum of the values of both the words. Doublewords are grouped into quadwords with the same algorithm, quadwords into octawords and so on until we get the length of the longest aligned run for the whole bitmap. An example can be seen in figure 6.3.

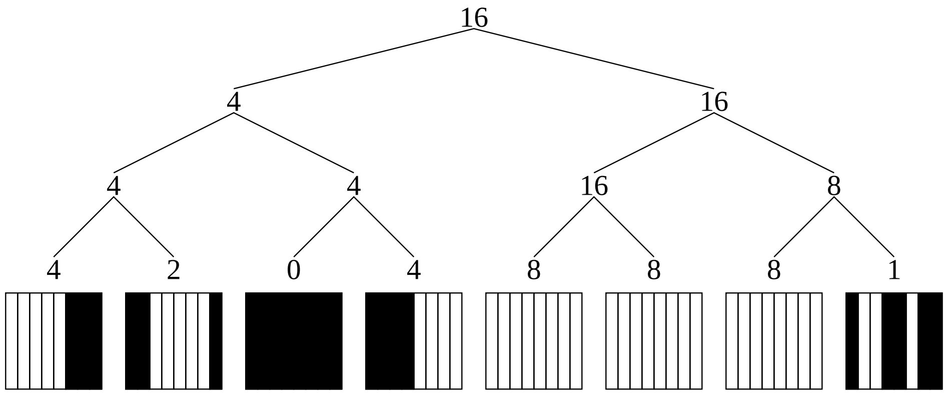

Figure 6.3: An example of binary buddy tree for eight 8-bit words.
Each number represents the largest aligned free block. In real implementation, $\log_2$ of the number is stored.

This algorithm has several advantages:

- Only by looking at the root node of the binary buddy tree can we tell the length of the maximal free run (having the above size and alignment criteria) in it — thus we can decide if we want to search or skip the bitmap.
- The algorithm for finding a free run of a specified size near a specified block can be easily implemented with the complexity of $O(\log n)$ (where n is the size of the bitmap): We start at the top of the tree and go down the tree by nodes that have a run of appropriate size. If both sons have a run of the appropriate size, we go to the son that is closer to the specified block.
- Each node's value depends only on values of its sons — thus the tree can be updated with $O(\log n)$ complexity. When some bits in the bitmap are changed, we simply walk up to the root of the tree and fix values on the nodes.

The main disadvantage of this algorithm is its space consumption. It consumes twice as much space on the disk as a simple bitmap (and much more than a B+ tree or list of extents). The increased space consumption increases the number of accesses to the disk and decreases performance.

6.2.3 B+ trees in the XFS

To achieve the same goal (avoiding a linear scan of bitmaps), the XFS uses B+ trees containing pairs (starting block, number of free blocks). Each allocation group has two

B+ trees of all free runs. One B+ tree is indexed by the starting block of a run and the other by run lengths [37].

B+ trees have the advantage that they consume less space than binary buddy bitmaps and can allocate runs of any size and alignment (not just the power of 2). They have the downside that they can allocate either by starting block or by run length, but not by both, as binary buddy bitmaps can.

6.2.4 List of operations in ZFS

ZFS uses a completely different method to manage file allocation information [95].

ZFS divides space on the disks into regions called metaslabs. Each metaslab has one space map. The space map has different representations on the disk and in the memory. On the disk, the space map is stored as the list of allocation and free requests. When the filesystem processes allocation and free requests, it appends them to the end of the map — with a bit that represents whether the run is being allocated or freed.

In memory, the space map is stored in an AVL tree indexed by the block number.

The allocation request is processed in the following way: find a block run of suitable size near the requested block number in the AVL tree, remove it from the AVL tree and write the offset and length to the end of the on-disk space map.

The request to free blocks is processed similarly; the run of blocks is added to the in-memory AVL tree and appended to the end of the on-disk space map.

The previously described operations make the on-disk space map grow. To shrink it, the following mechanism is used. When it is few times larger than the number of runs in the in-memory AVL tree, the block runs in the AVL tree are written to the disk, making a new condensed space map. The old space map is then dropped.

6.3 Block allocation in the Spad filesystem

As said in section 6.1.3, the Spad filesystem has only one block allocator; there is no separate allocator for inodes. Inodes[3] are embedded either directly in directories or allocated using the block allocator.

I wanted to save disk space occupied by allocator structures, so I decided not to use bitmaps. In a typical usage there are long runs of allocated and free blocks in bitmaps — so the information about free space can be stored in a more efficient way. The Spad filesystem use pairs of (starting sector, number of free sectors) just like the XFS, but it does not organize them in B+ trees — it uses sorted double linked lists.

Free space in the Spad filesystem is described in one or more allocation pages called *apages*. The number of apages can grow during filesystem operation. Each apage contains a fixed number of apage entries — the number depends on the page size selected when creating the filesystem.

[3] Called fnodes in SpadFS terminology.

An apage entry contains the starting sector number (64-bit), the number of sectors (32-bit) and two list link pointers (each 16-bit).

Some of the apage entries contain information about free space runs and form a double linked list. The list is sorted by ascending sector number. The entries are sorted by their list pointers (i.e. if you walk entries by `next` list pointer, you get ascending sector numbers). The entries are not sorted by their position in an apage — the position of entries can be absolutely random.

Remaining entries (those that are not part of the double linked list) are spare entries with the sector number and number of sectors set to zero. They form a single linked list. When the allocator needs a new apage entry, it takes it from this list. When it wants to remove some apage entry from the double linked list, it adds it to this list.

The structure of the apage is shown in figure 6.4.

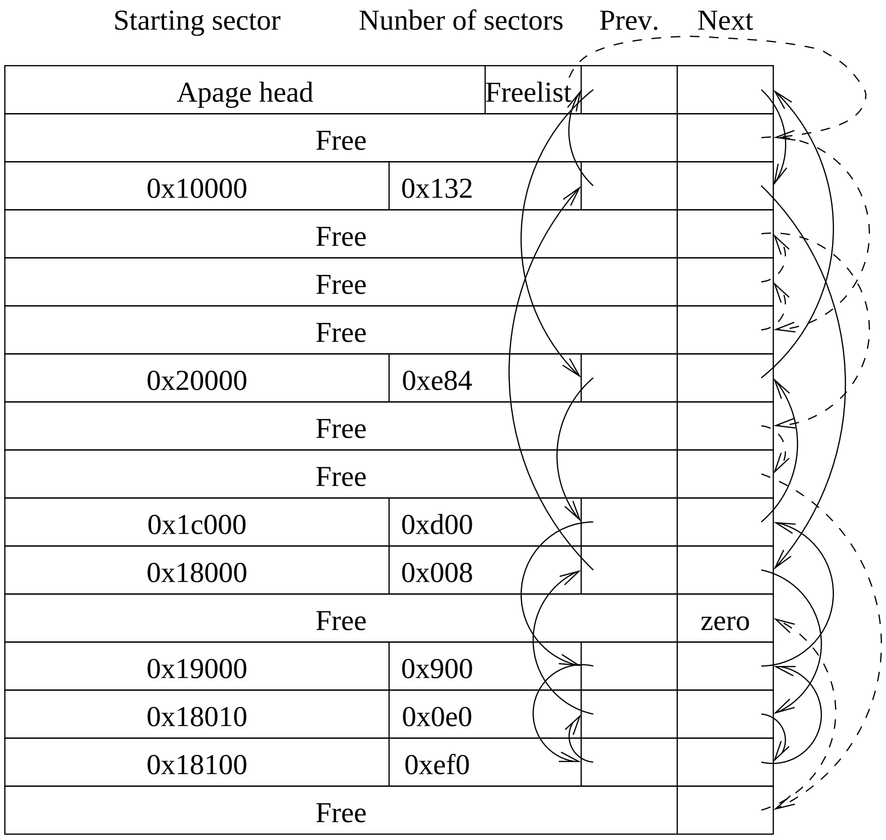

Figure 6.4: Apage with double-linked list

All operations on these structures can be done with average time complexity $O(\sqrt{n})$ (where n is the number of entries in an apage). When the allocator needs to find an entry nearest to a specified sector number, it doesn't need to walk the whole list. It selects any $\sqrt{n}$ entries in the apage and takes the one that has the greatest sector number lower than the desired sector number. Then it walks the list by its next pointer until it finds an entry with the sector number greater than the desired sector — on average, it walks only $\sqrt{n}$ entries in this step.

Apages are pointed to by an apage index (see figure 6.5) — it contains apage index entries — pairs of (sector number, pointer to apage). It denotes that the apage pointed to by an apage index entry manages free space information for blocks less than the block number in the apage index entry and greater or equal than the block number in the previous apage index entry. A binary search in the apage index is used to find the apage that manages free space information for a particular block.

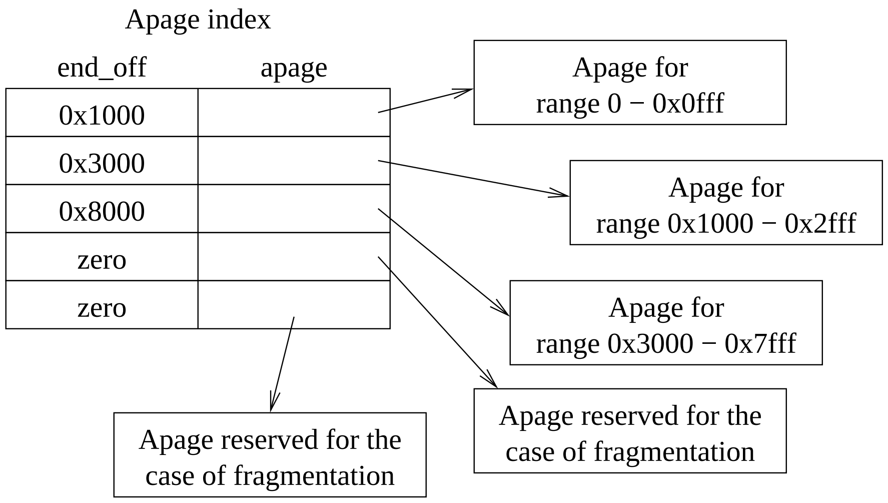
Apage index
end_off
apage
0x1000
0x3000
0x8000
zero
zero
Apage for range 0 − 0x0fff
Apage for range 0x1000 − 0x2fff
Apage for range 0x3000 − 0x7fff
Apage reserved for the case of fragmentation
Apage reserved for the case of fragmentation

Figure 6.5: Apage index

When the apage fills up (i.e. its single linked list of spare entries becomes empty) and new entries are needed, the apage is split into two apages and the apage index is updated. In the case of extreme free space fragmentation, a bitmap would be more space-efficient than the list of free block runs — if this is the case, an apage is not split, but it's converted to a bitmap and standard bitmap operations are used on it.

The filesystem has enough preallocated apages for the worst case fragmentation scenario — i.e. for the case all apages are converted to bitmaps. This worst-case number of apages can be easily estimated based on filesystem size and parameters used during the creation of the filesystem.

6.3.1 Algorithm for block allocation

This is the algorithm for allocation of a certain number of sectors near a specified sector:

- 1. Find the appropriate apage in the apage index, using binary search.
- 2. If the apage is a bitmap, linearly scan the bitmap for a run of desired length. Report success if found or failure if not found, and exit.
- 3. Find an apage entry nearest to the desired sector, as described in the previous section.
- 4. If the desired number of sectors is greater than the number of sectors in this entry, take the entry pointed to by its next pointer and repeat step 4. If we are already at the end of the list, return failure — the upper layer allocator will retry the allocation in a different apage or with a smaller number of requested sectors.
- 5. If all the space represented by this apage entry is allocated, remove the apage entry from the double linked list and insert it into the single linked list of spare entries.

- 6. If only a part from the beginning or the end of the apage entry is allocated, update its start and length entries accordingly.
- 7. If blocks in the middle of the apage entry are allocated, split the apage entry into two.

In step 1, we find the appropriate apage. We use binary search in the apage index array.
In step 2, we test if the apage is a bitmap. If it is, a linear bit scan is used.
In step 3, we find an entry in the double-linked list where we start searching. This is the lowest entry that has its sector number greater than or equal to the requested sector number.
In step 4, we search the double-linked sorted list using a linear scan for a block of appropriate size.
In step 5, we use the found apage entry. The newly allocated range is deleted from the range in the apage entry (because these sectors will be no longer free). The apage entry might be split into two as a result of this deleting. To reduce code size, the split is actually done by running the following algorithm for freeing blocks — it can result in a split of apages or conversion of the apage to a bitmap.

This is the algorithm for freeing a certain number of sectors at a certain position:

- 1. Find the appropriate apage in the apage index, using binary search.
- 2. If the apage is a bitmap (it has a flag set in its header), erase bits pertaining to the specified sectors and exit.
- 3. Find an apage entry nearest to the desired sector, as described in the previous section.
- 4. If the sector run to free overlaps with this apage entry, report filesystem error (freeing of sectors that are already free) and exit.
- 5. If the freed sector run adjoins this apage entry, simply adjust its starting sector and/or number of sectors. If the freed sector run adjoins two apage entries on both sides, delete one of them and add it to the single linked free list, and adjust the other to span the whole area of two apage entries and the freed sector run. Exit.
- 6. If the free list of apage entries is nonempty, take one apage entry from it and set its start and length to the sector run being freed and add it to the double linked list at a correct position (so that the list is maintained sorted). Exit.
- 7. If the total amount of blocks managed by this apage is greater than the number of bits in this apage, split the apage into two, update the apage index and repeat the whole free operation — go to step 1.
- 8. Convert the apage to a bitmap and go to step 2.

Steps 1–3 are similar to the allocation algorithm — finding the appropriate apage entry.
In step 4 we verify that we are not freeing a space that is already free. This can happen only as a result of metadata corruption.
If we can extend an existing apage entry to cover new freed space, we do it in step 5.
If there is no apage entry that adjoins the freed space, we allocate a new apage entry in step 6 and add it to the double-linked sorted list. If there is some free apage entry, we use it, fill it with the range to be freed.
If there is no apage entry to use, we must split the current apage into two apages. It is done in step 7. If converting the apage to a bitmap is possible (i.e. the apage maps a

space that is small enough), the apage is not split and is converted to a bitmap in step 8. Note that there are enough preallocated apages (made at filesystem creation) so splitting will always be able to find a new apage.
Let's say that the apage contains b bits usable for the bitmap (b is the total number of bits in the apage except the header). The apage is converted to a bitmap if it maps less than or equal to b sectors. Otherwise, the apage is split into two apages each mapping half of the sectors. Thus, each apage (bitmap or non-bitmap) maps at least $b/2$ sectors. If an apage mapping less than $b/2$ sectors existed, it could have been created only by splitting an apage that maps less than b sectors — which is a contradiction to the splitting condition (apage mapping less or equal to b sectors is converted to a bitmap and never split). From this condition we can calculate the number of apages that need to be preallocated: if the disk partition has p sectors and each apage maps at least $b/2$ sectors, the total number of apages needed to preallocate for the worst case is

$$\left\lceil \frac{p}{b/2} \right\rceil$$

6.4 Summary

Table 6.1 shows space allocation features for various filesystems.

Most filesystems have variable block size. In Linux implementations, blocksize cannot be larger than processor page size (usually 4kB), because the Linux kernel can't handle larger buffers[4]. Ext2 used to have a block size limit of 8kB and Ext3 used to have a block size limit of 4kB, but these limitations were dropped recently (but the page size limit still applies — thus larger block size is available only on certain non-x86 architectures). Note that the Linux kernel page cache is severely inefficient with smaller block sizes — the largest possible block size should be used.

ReiserFS and Reiser4 can store tails of files in the tree, thus large block size doesn't waste disk space. However, the page size limit applies: in ReiserFS, block size cannot be larger than page size; in Reiser4, block size must be exactly the page size, making the filesystem non-portable between architectures with different page sizes.

The FFS usually uses larger block size and it can divide a block to 1, 2, 4 or 8 fragments (bitmap bits are per fragment, not per block).

The JFS on Linux has a fixed block size of 4kB. The JFS on AIX and OS/2 has a variable block size — 512, 1024, 2048 or 4096 bytes.

The ZFS can use blocks of variable size on the same filesystem and it selects block size automatically [96].

Almost all filesystems use bitmaps for management of free space, only the FAT uses a file allocation table (it uses it for both file allocation information and free space allocation information); the XFS uses a B+ tree; the ZFS uses a list of allocate/free operations on disk and AVL tree in the memory and SpadFS uses a list of runs. SpadFS uses a bitmap if free space is extremely fragmented.

[4] A few filesystem drivers for Linux (FAT, NTFS, FFS) can handle larger block size by pretending smaller block size to the kernel; most others don't have this capability.

The "search complexity" field shows the complexity to find free space in an allocation page. Filesystems typically maintain some statistics about number of free blocks in individual pages so that they can skip full pages and they don't have to search all bitmaps if they are nearly full.

Some filesystems have separate space for inodes, some don't. ReiserFS, NTFS and SpadFS have the capability to reserve certain disk zone for metadata — metadata will be allocated preferably in this zone while data will be allocated preferably outside of this zone. However this doesn't a impose limit on the amount of data or metadata on the filesystem — if the corresponding zone fills up, data and metadata can be allocated anywhere. ReiserFS has this feature off by default; NTFS and SpadFS have it on by default.

	Block size	Space allocation structure	Search complexity	Separate space for inodes
FAT	512-32k	FAT	$O(n)$	No
HPFS	512	Bitmap	$O(n)$	No
NTFS	512-64k	Bitmap	$O(n)$	Yes (extendible)
FFS	4k-64k	Bitmap	$O(n)$	Yes (fixed)
Ext2	1k-64k	Bitmap	$O(n)$	Yes (fixed)
Ext3	1k-64k	Bitmap	$O(n)$	Yes (fixed)
Ext4	1k-64k	Bitmap	$O(n)$	Yes (fixed)
ReiserFS	512-8k	Bitmap	$O(n)$	No
Reiser4	CPU page size	Bitmap	$O(n)$	No
JFS	512-4k	Bitmap with binary buddy	$O(\log n)$	Yes (extendible)
XFS	512-64k	B+ tree	$O(\log n)$	Yes (extendible)
ZFS	512-1M	List of actions / AVL tree	$O(\log n)$	No
GFS	512-64k	Bitmap	$O(n)$	No
SpadFS	512-64k	List of runs or bitmap	$O(\sqrt{n})$, $O(n)$	No

Table 6.1: File allocation information in filesystems

7. Block allocation strategy

In the previous chapter I described a block allocator that can allocate blocks near a specific goal. Now let's discuss the way to find the "goal" block number. This is the task of the block allocation strategy subsystem.

The purpose of the block allocation strategy subsystem is to prevent file fragmentation in the long term. It must prevent both fragmentation of files and fragmentation of free space. Let's show two naive extreme examples:

- The allocator finds the largest free block run on the partition and starts allocating files from it. This approach minimizes the probability of file fragmentation but causes free space fragmentation — after a long time the allocator exhausts all large block runs and the partition will have only many small runs — that will cause inevitable file fragmentation.
- Another extreme is an allocator that finds the first free block and allocates a file in it (no matter how many free blocks follow it). It will perfectly defragment free space, filling small gaps, but it will do it at the cost of fragmenting all files. It is not a good solution either.

Real block allocators use an approach somewhere between these two extremes. They attempt to find an area with more free blocks where to start allocating, but they do not take this into extreme — they try to save large unallocated areas for future use and not exhaust them immediately.

Another important goal of the block allocation strategy is to minimize seek times by laying related data together. For example, the user often does tasks such as searching the directory tree for a specific file, searching files in a directory for specific text, copying of the directory tree, etc. The allocator thus should attempt to lay out files in the same directory near each other and lay out directories with the same parent directory near each other.

RAID arrays behave similarly to single disks with respect to seek time. They map sectors in such a way that sectors placed near each other on disk have a small difference in their logical sector number in the RAID array. So filesystems usually don't have special modes of allocation for RAID arrays; the algorithms developed for single disk work reasonably well on RAID.

It must be noted that currently there is no benchmark that can measure the quality of a block allocation strategy subsystem. Benchmarks do either repetitive or random accesses — in reality the access to files is neither absolutely random nor absolutely repetitive. For example when the user opens a file in the directory, there is a certain probability that he will soon open another file in the same directory. However no one knows these probabilities and no one has yet constructed a benchmark program that would act according to these probabilities.

I will give an example to show the inaccuracy of benchmarks in this area: log-structure filesystems [27] have traditionally very good benchmark results. These filesystems work by writing all data sequentially — thus, they have much better write throughput than traditional filesystems. But let's see this practical example: a user has a project file with his source code; it has many files; some were written a day ago, some a week ago, some a month ago, etc. Traditional filesystems will attempt to place these files near each other because they are in the same directory — thus searching, copying or compiling the whole project will be fast because of a reduced seek time. On the other hand, a log-structured filesystem will scatter the files over the whole partition — it groups data according to the time they were written, not the directory they belong to. So despite superior benchmark results, a log-structured filesystem has worse performance on tasks such as "search all files in directory for a given text". This is probably the reason why log-structured filesystems are not used in most systems, although the concept is very old. Unfortunately there isn't a benchmark that can show this deficiency.

Because of these problems — deficiencies show only after several months or years of real-world usage and the inability to benchmark algorithms — a block allocation strategy algorithm is the most subtle part of a filesystem. Algorithms are usually developed intuitively; they are based on solutions used in the long term on other filesystems and inventing something completely new is discouraged — the new algorithm may perform well for a few weeks but you can hardly tell what it will do over several years of usage.

There are some efforts to perform measurements of existing filesystems over a long time [40] [41] [42] [43] [44], but this research hasn't yet culminated in any usable algorithm for allocation strategy. I haven't performed tracing of filesystem activity and simulating fragmentation because this task is time-consuming — some of the research even took 5-years [45]; the other took one year to collect daily filesystem snapshots on a production server [40]. I maintain the point that at this stage of development, time spent for this is grossly disproportionate to the utility that this could bring.

I will show an example of a real-world deficiency in block allocation strategy: The NTFS [7] has a master file table that is an equivalent of an inode table on Unix filesystems. Unlike the traditional Unix filesystem, the NTFS can extend the space allocated by the master file table. The master file table is treated just like any other file and its location is described by lists of extents. Windows NT 4.0 created the filesystem with a small master file table and extended it according to its needs. However, because it allocated the space near the master file table for regular files, extending the master file table caused its fragmentation. The access time for a master file table is crucial for the performance of the whole filesystem; thus the filesystem got slower and slower over time as the master file table fragmented more and more. In Windows 2000 the problem was fixed by reserving an area of partition for the growth of the master file table and trying to not allocate regular files in it.

7.1 Flash memory considerations

Flash memories [46] (also called solid state disks) don't have seek latency [47], thus it doesn't matter where the filesystem reads the data from. Flash memories have limited number of rewrites [48]. But they have very inefficient random writes [49] — in some implementations writes to random locations cause massive performance degradation (i.e. Transcend TS64GSSD25-M can perform only 10 random writes per second — it is likely caused by flashing a new erase block with every write). Intel flash memories can perform many small writes per second — they pack these writes into a few erase blocks and update a remapping table — but this results in internal storage fragmentation and overall performance degradation over some time of usage. If the sectors remapped via small writes are read sequentially, it is found that they are not sequential in the storage and their retrieval causes accesses to many flash blocks. The current Intel implementation cannot automatically defragment the storage and recover from this performance degradation; the user is required to perform manual recovery, consisting of backing up the data, rewriting the whole storage linearly and restoring the data. The performance degradation and suggested recovery methods are described in [97].

Therefore, it seems reasonable that optimizing for a flash memory should not be focused on reads, but on writes[1]. The filesystem should attempt to avoid small write requests and submit as big write requests as possible. In the Transcend case this strategy would improve the performance of writes; in the Intel case it would mitigate the long-term storage fragmentation. The efficient algorithm would likely allocate blocks sequentially, not considering any proximity to other files or directories.

7.2 Ext2, Ext3

The allocation system in Ext2[2] is very old and simple; however, it is still reported to perform well. Linux filesystems prior to Ext2 used either lists of blocks (on ExtFS; a very bad solution — it generally returned a random free block on a partition) or a linear scan for the first free block following the file end (on XiaFS; it is better — it attempts to lay out file's blocks near each other, but it is reported to cause big fragmentation, too).

7.2.1 Allocation groups

Ext2 partition is divided into groups. Each group has a bitmap for its data blocks, a bitmap for its inodes, an inode area and a data block area. Most of the space in a group belongs to the data block area. Some groups also have a copy of the superblock — that is not normally used, but in case the main superblock is destroyed the filesystem can be recovered from one of these backup superblocks. Group size is $block\_size^2 * 8$ (block size can be set to 1024, 2048, 4096, 8192, 16384, 32768, 65536; block size must be less than or equal to page size) — groups are so large that the bitmap for blocks in a group consumes exactly one block. The number of inodes in a group can be set when creating the filesystem. The purpose of groups is to minimize seek times — the disk head moves only a little distance from the inodes and bitmaps to data blocks.

A typical layout of an Ext2 group is shown in figure 7.1.

A special area on the disk is used for group descriptors — it contains numbers of free sectors and inodes in each group so that the allocator can skip full groups and make a strategic decision about a group where to allocate.

[1] In contrast, optimizing for a rotating disk is focused on reads because read request prevail in common usage patterns.

[2] Ext3 has the same allocation strategy as Ext2.

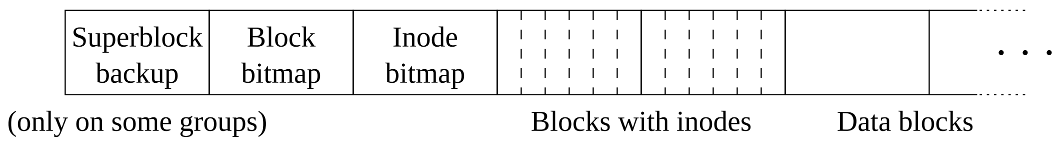

Figure 7.1: Blocks at the beginning of Ext2 group

7.2.2 Block allocation

To allocate a block for a file or directory, Ext2 uses a simple algorithm:

- 1. Select a goal. If this is the first block in a file or directory, the goal is the first block in a group in which the file's or directory's inode is. Otherwise, the goal is the block following the last block in the inode or directory.
- 2. Find a group to allocate in — if the goal's group has some free blocks, use it. Otherwise linearly search for a group with some free blocks. If using other than the goal's original group, set the goal to the start of the group.
- 3. If the goal can be allocated, allocate it and return.
- 4. If there is a free block a few blocks after the goal, allocate it and return.
- 5. Search the bitmap for a free byte (8 blocks) from the goal to the end. If found, try to walk at most 7 bits backwards to make sure that the returned block is placed exactly after another allocated block. Return it.
- 6. Search the bitmap for a free bit following the goal. If found, return it.
- 7. If no bit is found and the goal is not at the start of the group, take the next group and go to step 2.
- 8. If the goal was at the start of the group and no free bits were found in the group, it means filesystem corruption. At step 2 we verified that the group has some free blocks and if there are none, it means that the free block counter is skewed. A filesystem error is reported in this case.

This algorithm scans at most two bitmaps and at most all group descriptors for their free block counters. In the most common situation the goal (a block following the last block in a file) is free, so the algorithm returns immediately.

Note the search for a continuous run of 8 free blocks (one free byte). It is a balance between attempting to search for the largest free run and using the first free block available. As I described earlier, neither of these two extremes is good. Searching for 8 blocks was selected intuitively and it seems to perform well because it hasn't been changed in 12 years.

When the algorithm finds a free block, it does an optional prealloc — allocates up to 8 more blocks. The prealloc is truncated when the file is closed. The purpose of the prealloc is to reduce CPU consumption by calling the allocator 8 times less and to reduce the file fragmentation when two files near each other are written simultaneously — the fragmentation still happens, but the files will be interleaved in 8-block continuous runs.

7.2.3 Inode allocation

When allocating a non-directory inode, Ext2 first tries its directory group and if it's full (it has no free inodes or no free blocks), it picks a starting group based on the directory's inode number[3] and does a quadratic hash search from the directory group. The quadratic hash works in the following way: g is picked as the start of the quadratic hash and n is the number of groups. Groups $(g + 2) \bmod n$, $(g + 4) \bmod n$, $(g + 8) \bmod n$, $(g+16) \bmod n$, $(g+32) \bmod n$ ... etc. are searched until the search wraps around the original group g. The search is really not quadratic but exponential, but the name quadratic hash stuck for some unknown reason. When a group with some free blocks and some free inodes is found, it is used. If the quadratic hash doesn't find anything, all groups are searched linearly and the first group with some free inodes is used.

A directory inode is found in a different way: all group descriptors are scanned, the groups with a below-average free inode count are rejected[4] and the group with the most free blocks is selected.

This algorithm has not changed since the release of Linux 1.0 in 1994. The only little but important change is that when allocating a non-directory inode, groups with free inodes but no free blocks are skipped too (except the last linear search that accepts even groups with no free blocks). The original algorithm searched only for group with free inodes and didn't look at the free block counter.

The algorithm works well in real world scenarios, so there was no need to redesign it. It keeps files' blocks together and it keeps files in the same directory near each other. But it has one flaw — it does not keep adjacent directories together. A new directory's inode is selected regardless of the location of the parent directory. If the user is reading a directory tree with many small directories (a typical example would be a source code of some program), each directory is in a completely different group. The directory inode allocator tries to keep the free blocks count in all groups equal — thus each directory goes to a group that had the least allocated blocks when it was created. When the user runs `find` or `grep -r` on such a directory tree, the disk head seeks randomly around the whole disk.

This problem was fixed in Linux 2.6 by adding a new Orlov allocator[5] [98]. This allocator changes only the strategy for allocating directory inodes. It linearly scans group descriptors starting from the parent directory's inode and tries to find a group that satisfies these criteria:

- The number of free inodes in a group is greater than or equal to

$$total\_free\_inodes/number\_of\_groups - inodes\_per\_group/4$$

- The number of free blocks in a group is greater than or equal to

$$total\_free\_blocks/number\_of\_groups - blocks\_per\_group/4$$

[3] So that files in the same directory will likely be in the same group.

[4] There must be at least one group with equal or above-average free inode count.

[5] Orlov is the name of the person who invented it. The allocator was first implemented in FreeBSD.

- The number of directory inodes in a group is less than

$$total\_directories/number\_of\_groups + inodes\_per\_group/16$$

- Debt is less than

$$blocks\_per\_group/(total\_used\_blocks/total\_directories)$$

  Debt is a value kept in the memory for each group. It is increased each time a directory inode is allocated in the group and decreased each time a file inode is allocated in the group. Debt is kept in interval 0 – 255; if it falls outside this interval, it is not increased or decreased.
- Debt is less than

$$blocks\_per\_group/256$$

- Debt is less than

$$inodes\_per\_group/64$$

When all these conditions are met, the directory inode is allocated in the group. When these conditions are not met for any group, the search is repeated and the first group with some free inodes is used.

The Orlov allocator performs well when creating a directory tree with related data and keeping these data together is desired; however, it fails badly on another occasion — creating users' home directories. The traditional allocator would place each directory in a different group, but the Orlov allocator keeps them together in one group. The debt counter tries to mitigate this problem, but it is not perfect — it will prevent directory aggregation when many directories are created at once, but it will fail if the directories are created in greater intervals. Due to this, Ext2 added a user-settable flag that says that the directory's subdirectories should be distributed across the whole partition. This flag should be set on `/home`, `/usr`, `/usr/share`, `/usr/local`, `/usr/local/share`, `/opt` and other directories that are expected to contain very large non-related subdirectories.

The performance of Orlov an allocator on FreeBSD can be seen in [50].

7.3 FFS

The FFS [7] is the native filesystem of FreeBSD. It was derived from the traditional Unix filesystem and has several enhancements over it. The FFS uses allocation groups like Ext2. The FFS uses block fragments[6] to be both space-efficient for small files and have good performance for big files.

7.3.1 Block fragments

Each block on FFS is split into 1, 2, 4 or 8 fragments. The size of a block and the size of a fragment can be selected when the filesystem is created. A file has several full blocks and a fragment at its end.

[6] They have nothing to do with file fragmentation.

Free space bitmaps contain one bit for each fragment. Direct and indirect blocks in the inode contain pointers to whole blocks. If the inode has only direct pointers, the last direct pointer is the pointer to a fragment; the size of this fragment is determined by the inode size modulo the block size. Indirect blocks contain pointers only to full blocks, not fragments — it is not worth trying to save space on large files while increasing complexity of a code for manipulation fragments.

When the last fragment in the inode needs to be extended, it can be done either by allocating space following the fragment in a bitmap (if it's free) or by moving the whole fragment elsewhere.

When the filesystem allocates a new fragment, it selects one of these two strategies:

- Allocate an empty block and place the fragment at its beginning. The advantage is that extending the fragment will be fast (no data movement needed). The disadvantage is that when many small files are created, the free space will fragment badly, leaving many small unallocated fragments.
- Allocate a fragment having exactly the desired size. The disadvantage is that extending the fragment will move all data elsewhere. The advantage is that this will nicely defragment free space — files will be allocated in areas having exactly the files' size, thus fragmenting neither files nor free space.

FreeBSD switches between these two algorithms based on the amount of the total free space wasted in fragments. Note that this free space is unusable for storing large files; thus the filesystem must make sure that it never grows too much.

7.3.2 Block allocation

The FFS searches only for one free block (unlike Ext2 which searches for 8 free blocks). However, blocks on the FFS are larger than blocks on Ext2. The upper layer allocator gives a goal to the block allocator. The allocation algorithm is this:

- 1. Test if the goal is free. If it is, allocate it and exit.
- 2. Scan the bitmap from the goal to the end of the group. If a free block is found, allocate it and exit.
- 3. Scan from the beginning of the group to the goal. If a free block is found, allocate it and exit.
- 4. Quadratically hash into other groups starting from the original goal's group (see the description of quadratic hash above in Ext2 section). If a group with at least one free block is found, allocate it and exit.
- 5. Linearly scan all the groups for one free block. If found, allocate it and exit.
- 6. Otherwise, there are no free blocks on the partition. Report failure.

This allocator is used for allocating fragments and inodes as well. The function performing these tasks is generic — it takes another function performing search of a group and allocation as a pointer. Thus it can be used for allocation of all these objects.

FreeBSD doesn't have a prealloc but it has write clustering — it can move already allocated blocks around before they are written and reallocate them to a larger continuous area. The clustering happens before the data is written to the disk but after the blocks have been written with a `write` syscall. Thus the clustering code usually knows the whole size of the file and can allocate the exact number of desired blocks. The performance penalty of clustering is small (clearing old bits in bitmaps and allocating new ones); no real data are moved on disk during clustering. Performance evaluation of write clustering can be found in [40] and [51].

7.3.3 Forcing file fragmentation

The FFS block allocator distributes files much more evenly across groups than the Ext2 allocator. Before allocating the first indirect block and after a configurable amount of blocks (defaults to the number of blocks mapped by one indirect block) the allocator skips to the next block group with an above-average number of free blocks. The next block group is found with a simple linear search, skipping block groups with below-average free space. The purpose of this action is to not fill up any block group, thus preventing allocating inodes' data too far from inodes themselves. It has a downside, too — especially with small block sizes, a large file will be scattered around the partition in small fragments, thus slowing down seeks within the file. With a 4096 byte block size, the default file fragment size will be 2MiB ($4096/8 * 4096$)[7]; with a 8192 byte block size, the fragment will be 8MiB; with a 16384 byte block size, the fragment will be 32MiB.

7.3.4 Inode allocation

Inodes are allocated first by finding a goal, trying inodes in the same group as the goal, and then using the standard allocator with its quadratic hash to find a group with at least one free inode. The routine is exactly the same as in the section "Block allocation" above.

The question is how to find the goal: for non-directory inodes, the goal is the parent directory inode; for directory inodes, the Orlov allocator (exactly the same as in Ext2, including its "magic" constants) is used. Note that the Orlov allocator was first developed on the FFS and later adopted on Ext2.

The Orlov allocator first appeared in FreeBSD 4. Before it, the group with an above-average free inode count and the least allocated directories was used for new directory allocations.

7.4 ReiserFS

ReiserFS has a capability to use zones.

A ReiserFS partition may be divided into two zones: a formatted block zone (used to store tree elements) and an unformatted block zone (used to store file content). This division is caused by mount option `concentrating_formatted_nodes`. An optional ar-

[7] Block number on FFS2 is 64-bit; thus one indirect block contains $block\_size/8$ pointers.

gument to this option determines the percentage of formatted block zone; the default is 10%[8]. Tree elements and data are allocated first in their preferred zone and if it's full, then in the other zone. Using this option will increase performance when scanning a large directory tree (without reading files). These zones are not enforced; when one of the zones is full, the blocks are allocated from the other zone.

7.4.1 Block allocation

The block allocator is given two arguments: a hint[9] and a border. A border is the start of an unformatted block area. Getting the hint will be described in the following section. When allocating unformatted blocks, the allocator searches the partition first from hint to partition end, them from border to hint and finally from beginning to hint (in the last case, the unformatted block zone is full; thus we are allocating unformatted blocks from the formatted zone). When allocating formatted blocks, the allocator searches first from hint to border, then from beginning to hint and finally from border to end (in this case the formatted block zone is full and formatted blocks are allocated from the unformatted block zone).

When allocating unformatted blocks, the block allocator can preallocate a specific number of blocks; the default is 16. Unlike on Ext2, the number of preallocated blocks is determined before the allocation is attempted and the allocator searches for a continuous run of an appropriate number of blocks. The filesystem has two mount options to control preallocation:

- `preallocsize` determines the number of preallocated blocks plus one; the default is 17 (it means to allocate 1 regular block and 16 preallocated blocks).
- `preallocmin` determines the number of blocks that must already exist in an inode before the preallocation is attempted. If the inode has fewer blocks than `preallocmin`, no preallocation is done. The purpose of this option is that small files can be efficiently used to fill small free space runs on the filesystem; thus preallocation may be countereffective for them. The default is 0, which means to always preallocate.

When allocating unformatted blocks, the allocator skips bitmaps that have less than 10% free space unless the inode already has some blocks in that bitmap. When the whole partition has less than 20% free space, this restriction is removed. This restriction has a similar effect as various "below average" or "least allocated blocks" strategies in Ext2 and FFS — the purpose is to not place files in nearly full bitmaps.

[8] The argument may be changed even when files already exist on the filesystem, but it is not recommended — it will likely distort allocation strategy and cause fragmentation. The filesystem should be mounted with the same value for its whole lifetime.

[9] In Ext2/3, FFS and SpadFS terminology, it is called a *goal*. In ReiserFS terminology it is called a *hint*. They both mean the same — the block from where the block allocator should start searching for a free space.

7.4.2 Selecting the hint

Now the task is to select a hint that will be passed to the block allocator. ReiserFS has several algorithms for that. The general idea is: use some existing blocks in the file near the block-to-be-allocated as a hint. When no such blocks are found — either because the file is empty or because the file is sparse and all existing blocks are too far away in the tree — the allocator takes hashes of some of the file keys and uses them as a hint.

There are several options that select allocator behavior:

- `displacing_large_files` — an optional argument is the number of blocks (default 16). Files that have more than this number of blocks are placed in a different location, based on a hash of object ID or directory ID (if option `displace_based_on_dirid` is used). This is an equivalent of forced fragmentation on the FFS.
- `hashed_formatted_nodes` — when set, the location of formatted blocks is hashed also according to the object ID or directory ID (if `displace_based_on_dirid` is set) of the object where the B+ tree split happens.
- `new_hashed_relocation` — does a similar thing as the previous option, with a different hash — directory ID is used unconditionally. When neither one of these two is selected, the hint for the new tree node is set near the block where the split happened.
- `dirid_groups` and `oid_groups` — for unformatted blocks, select the location according to a hash of directory ID or object ID. Directory ID is preferred (and it is the default) — it places files in one directory near each other. Hashing according to an object ID is useful only in special circumstances, for example when having a low number of directories with very large files.
- `hundredth_slices` is another way of hashing unformatted nodes. This way, the disk is divided into 100 groups and the hashed value determines the group where the allocation should start. It can prevent massive fragmentation of free space due to the fact that the hash distributes files to random places.

All these strategies for selection of the hint have one serious shortcoming (compared to Ext2 and FFS) — they do not look at the actual layout of allocated data on the disk. They place data at random places according to the hash and with the hope that the hash will prevent fragmentation. It can result in a disaster when the hash places a new allocation in an almost full area — the file will seriously fragment. The restriction not to allocate in bitmaps with less than 10% free space mitigates the worst effects of this scenario.

There is another allocation strategy that can prevent fragmentation coming from too many hash collisions — selecting directory IDs of files. Each inode has a directory ID — the function of the filesystem doesn't depend on this value. It used to be the object ID of the directory and its purpose is to group objects in one directory together in the tree. Note that directory ID is the most significant value when comparing tree keys. When an inode is moved or hardlinked to a different directory, its directory ID stays the same — the ID of the old directory — but it doesn't matter.

Recently ReiserFS changed to a better allocation of directory IDs — a new inode (applies to both directory and non-directory inodes) will have the directory ID of its parent directory assigned as its directory ID if the bitmap assigned (through the hash) to this ID has more than 60% free space. Otherwise its directory ID will be the object ID of its parent directory. The result of this strategy is this: when creating a deep tree of files and directories, all inodes in this tree will have the same directory ID and all files will be placed near each other in one place. When the bitmap fills up to 40% used space, the object ID of a directory will be assigned as a directory ID of inodes in it, thus moving the directory (and its whole subtree) to a different bitmap. This strategy is very good because it preserves locality even across an arbitrarily deep directory tree and has an equivalent effect as the Orlov allocator for FFS and Ext2.

7.5 The Spad Filesystem

Now I will describe the block allocator algorithm that I developed for the Spad filesystem. The filesystem is divided into three zones: metadata, small files and large files. This is similar to the two zones in ReiserFS with `concentrating_formatted_nodes`, except that I use a separate zone for small and large files. Each zone is divided into groups (see figure 7.2). The number of groups and the size of zones can be adjusted when creating the filesystem. The maximum number of groups is 2^{16}. Files larger than a *cluster threshold* are placed in a large file zone. Files in a large file zone are padded up to a *cluster size* — the purpose of this padding is to prevent fragmentation of free space in the zone. When the appropriate zone fills up, the allocator allocates data in other zones.

The rationale beyond this zones schema is similar as in ReiserFS — to get very fast walking of a directory tree (when the user doesn't read any files, just scans for a filename) and very fast time to check the whole filesystem. The downside is that directory-to-file content seek time is higher than on Ext2 or FFS that attempt to lay inodes near their data. Additionally, zones together with linear mapping through the MD or DM layer can be used to put metadata and data to different disks (such as in [52] to avoid disk head seek from metadata to data).

Note that on other filesystems (Ext2, FFS, ReiserFS), the groups are somehow correlated with the layout of bitmaps on the disk — there is usually one bitmap block per group and it is placed near the group. On SpadFS, the block allocation information is not stored in bitmaps, but rather in sorted linked lists (as described in section 6) — thus the groups and zones are purely virtual; statistics about them (free block counts) are kept in the memory during filesystem operation but they are never written to the disk. They do not have any representation on the disk. The first prototype implementation of SpadFS didn't use groups; it used random allocation goals similar to ReiserFS and this strategy turned out to be quite bad — then I adopted and implemented the group approach[10] from Ext2 and FFS because something better hasn't been invented yet.

All the values can be adjusted when creating the filesystem. The defaults are:

- Number of groups — 512. Because the number of sectors in a group must be a power of 2, the exact value of 512 groups cannot be achieved. The filesystem creating

[10] Because the groups are kept only in the memory, this switch didn't even require a change in the on-disk format.

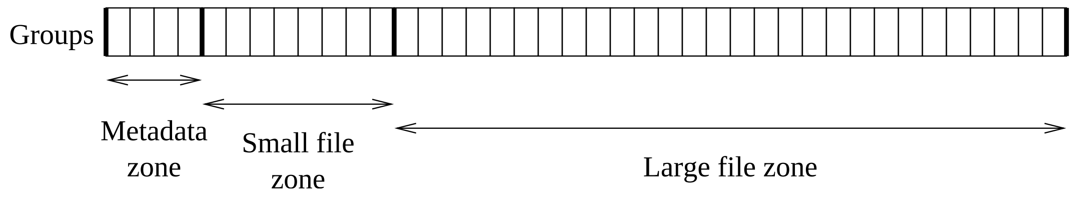

Figure 7.2: Groups and zones on the Spad filesystem

program attempts to select a group size such that the number of groups is nearest to 512.

- Size of metadata zone — 1/64 of the filesystem.
- Size of small file zone — 1/8 of the filesystem.
- Cluster threshold — 2^{17} bytes.
- Cluster size — 2^{15} bytes.

Each directory contains two values — the number of the preferred small file group and the large file group, where its files will be allocated. The rationale beyond this decision is this: I want to reduce seek time between files in the same directory. The seek time from the directory itself to a file content will be high but it doesn't matter much, because file descriptors are directly in the directory. When the user wants to read all files in a directory, the filesystem will read one directory block describing many files and then it will seek from file to file. I want to minimize the file-to-file seek rather than the directory-to-file seek.

7.5.1 Block allocation

When allocating blocks, the allocator is given a goal (selecting the goal will be discussed in the next chapter), a number of blocks to allocate and flags specifying the preferred zone (metadata, small files, large files). It first attempts to allocate from the goal to the end of the group (even if the group does not belong to a preferred zone — if the goal is in a non-preferred zone, the allocation in the preferred zone must have failed some times ago). This allocation from the same zone is not performed if allocating a new directory, when the zone has below-average free space and when the free space percentage in a zone is less than 25% — the purpose of this decision is to spread new directories more evenly across the disk, similar to the effect of the Orlov allocator.

If this allocation fails, it attempts to search in a preferred zone. If the goal lays in the preferred zone, it starts from the group following the goal; if it doesn't, it starts from the beginning of a preferred zone. The allocator uses these four strategies until one of them finds a continuous run of the requested number of blocks:

- 1. Quadratic hash, skipping the groups with below-average number of free blocks.
- 2. Linear search, skipping the groups with below-average number of free blocks.
- 3. Quadratic hash, searching all groups.
- 4. Linear search, searching all groups.

Note that the search usually finds a block in steps one or two. There must be at least one group with above-average free space. It is theoretically possible that all groups with above-average free space are very fragmented and the requested number of blocks can only be allocated in groups with below-average free space, but it is not very probable.

If the above steps didn't allocate the space, it means that the preferred zone is full (or so fragmented that a continuous run of requested size can't be found). So the allocator tries other zones in this order:

- If the preferred zone was a large file, the zones for small files and metadata are tried.
- If the preferred zone was a small file, the zones for large files and metadata are tried.
- If the preferred zone was metadata, the zones for small files and large files are tried.

If allocating from the big file zone, the four-step quadratic and linear search as described in the previous paragraph is used. Otherwise (from the small file zone and metadata zone) the allocator tries these two steps:

- 1. Scan the groups in the zone backwards, trying to allocate in groups with above-average number of free blocks.
- 2. Scan the groups in the zone backwards, trying to allocate from all of them.

The purpose of this modification is that when placing blocks to a group where they don't belong, they will less likely collide with regular data when allocating them from the end of the zone. Note that the space in the groups themselves is searched forward for free space — to maintain a good data layout for the disk prefetch cache. Only the groups are selected in reverse order.

When the block allocator can not allocate the requested number of blocks in any of the above steps, there is no continuous free space of the requested size in either group[11].

To prevent the block allocator from consuming CPU while looping over the whole partition when it is too fragmented, the maximum allocatable continuous run is maintained. When the allocation fails, the maximum is set to the failing value and all allocations larger than or equal to it are refused. When freeing blocks, the maximum is increased, because freeing a block could create a new long run. The failure of the allocation does not necessarily mean the failure of a user's request — the request may be retried with a smaller prealloc (if allocating file data) or when allocating directory hash page allocation fails, the directory will form a chain instead of a hash tree.

The filesystem is designed in such a way that allocation failures of more than one block do not have an effect on the user's programs — the action can always be retried with fewer blocks. If the allocation of one block fails, it means that there is no free space on the partition and an error is returned. This design prevents the situation that the filesystem reports some free space with `df` command but it returns errors on some operations because the free space is too fragmented[12] — on SpadFS allocations are refused deterministically regardless of free space fragmentation.

[11] Note that there may be free space of a desired size starting in one group and ending in another. Such space is not searched.

[12] This scenario can happen on the FFS when most of the space is consumed by incomplete fragments. It can happen for example when you fill-up the filesystem with files having just one block and truncate all of them to the size of one fragment.

7.5.2 Selecting the goal

When extending a file, a goal is the last block allocated for the file. There is one exception to it — when the file grows above the threshold so that it will belong to the large file zone, the goal in the small file zone is not used. When the last block cannot be used as the goal (either because the file doesn't have any blocks yet or because it has all blocks in the small file zone and the next block should be allocated in the large file zone), the goal is selected from the file's directory. Each directory has two 16-bit values denoting the preferred group for small and large files. The goal is set to the beginning of one of these groups (depending whether we want to allocate in a small or large file zone).

The most complex task is how to get these preferred-group values for each directory.

A new directory has its two preferred group values inherited from the parent directory. When the request cannot be satisfied from a directory's preferred group, the filesystem searches the other groups as described above. If the goal was taken from the directory's group (and not from the last block of a file), the directory group is updated to the group of the resulting block. Furthermore, the filesystem walks directories up to the root and updates their preferred group. It stops the walking when it hits the root directory or when it finds a directory that was created more than one hour before.

The rationale for this one hour time limit is this: When extracting archive with deep directory structures, I want the extracted archive to be compact and reside in as few groups as possible — thus when I move to the new group, I update the preferred groups of all parent directories to make sure that new allocations will go to the new group. However, when no directories are being created and the user simply creates many files in existing directories, I want the filesystem to spread data of different directories across different groups — to make sure that files in the same directory are most likely to share the same group and to minimize seek time between them. The one hour time limit on propagation of group change is designed to distinguish between these two cases.

7.6 Summary

The block allocation strategy subsystem is the least researched part of a filesystem design. The algorithms are based on heuristic approaches and they are not published.

In this chapter I described three approaches to solve this problem used in existing filesystems — Ext2/3, FFS and ReiserFS. I obtained the algorithms from the source codes. I based my design on ideas from these existing algorithms.

I made the decision to place metadata sectors in one zone — this significantly increases the speed of operations that traverse the directory tree and slightly decreases the speed of opening files (as will be seen in the section with benchmarks).

8. Checking the filesystem

In this chapter I will discuss the design of the filesystem checking tool[1] [53]. Even if the filesystem has a method that keeps it consistent across crashes, such as journaling, crash counts or phase tree, the filesystem may get corrupted for other reasons. The most common errors that can corrupt the filesystem are:

- Bad blocks on disk
- Bad chipset that causes silent data corruption
- Unreliable cabling, resulting in some blocks being written and some not
- Part of a partition is overwritten because of an administrator's mistake

The filesystem driver in kernel can not recover from these corruptions and an external filesystem-checking utility is needed. The utility scans the whole partition, attempts to recover as much data as possible and brings the filesystem to a consistent state.

The goal of the tool is to guess what the corrupted filesystem structures meant before the corruption occurred and attempt to minimize damage to the user's files. The function of the tool cannot be formally defined — if we wanted to prove that some of the tasks were done correctly, we would need to understand meaning of the data on the filesystem, which would require artificial intelligence (it is called AI-complete problem [99]).

8.1 Requirements for the filesystem checking tool

To give some insight into filesystem-checking related problems, I will tell a story that happened to me many years ago, in the time of MS-DOS.

I ran a MS-DOS filesystem checking program `scandisk`. It found no errors on the disk except a few lost clusters[2]. `scandisk` attempted to follow the goal to recover as much data as possible, and thus decided not to abandon lost clusters, but save them. With some heuristics it found that the content of the lost clusters looked like a directory, so it tried to recover them as a directory instead of a file. In fact it wasn't a directory, it was a random content of another file.

`scandisk` added a directory entry pointing to these lost clusters and decided that it had to restart checking the whole filesystem — so it did, found this new "directory" and started parsing its content. It added "." and ".." entries to the beginning; it removed entries that were obviously invalid — and it found one entry that looked like a subdirectory (but in fact it contained random bytes). It looked at this subdirectory's cluster (in fact a random number), found that it didn't have "." and ".." entries — so it "fixed" the "subdirectory" by adding these entries.

The result of this filesystem repair activity was that a random, completely correct file was corrupted by overwriting it with directory entries for "." and "..".

[1] This tool is called `fsck` in Unix environment and `chkdsk` in Windows environment.

[2] A lost cluster is an area on the disk that is marked as allocated in the File allocation table, but no directory entry is pointing to it.

A minor problem (lost clusters) caused a major problem (corruption of file) due to bad design of the filesystem checking algorithm. This is an example of how filesystem checking and repair shouldn't be performed.

A very strong requirement of a checking and repair algorithm is that even if part of the disk becomes corrupted or overwritten, the repair program never damages correct data.

In this chapter I will describe the design of this algorithm and how I tried to achieve this requirement.

8.2 Ext2 / Ext3

I will first describe the design of the filesystem-checking algorithm for Ext2 and Ext3 filesystems. The functionality of Ext2 fsck and a proposed improved design is published in [54]. Because Ext2 and Ext3 filesystems have the same data layout except that Ext3 supports journal and optional directory hash, the same utility `e2fsck` is used to check both filesystems.

The Ext2 filesystem has a rather simple structure making checking easy. Because inodes are at fixed places, the filesystem checking utility doesn't have to try to guess if a given piece of data is an inode or content of a file.

At first `e2fsck` tries to find a superblock. It tries to use a regular superblock at the beginning of the partition. If this superblock is damaged, `e2fsck` scans the partition for backup superblocks. Then, checking is performed in these 5 passes:

In pass 1, `e2fsck` scans all inodes and checks that their fields are valid and that their direct and indirect pointers point to valid data. In this pass, `e2fsck` treats directories just like ordinary files — i.e. it only checks that they contain valid blocks, but doesn't examine directory entries in the blocks. If cross-linked blocks are detected (i.e. two pointers pointing to the same block) this pass breaks into three additional passes:

- Pass 1B: Generating a list of duplicate blocks and inodes that contain them. Pass 1 only generated a map of used/unused blocks — it didn't tell which blocks are used by which inodes.
- Pass 1C: Walking the directories to get filenames of duplicate inodes. This is needed to be able to report meaningful error messages. The user needs to know what files are cross-linked in order to be able to examine them later and possibly replace them.
- Pass 1D: Actually duplicating the cross-linked blocks. One of the files will contain incorrect data after this action, but `e2fsck` has no way to tell which. It is up to the user to check the files manually.

In pass 2, `e2fsck` iterates over all directory inodes found in pass 1 and checks that they contain valid directory entries. Because pass 1 already resolved all cross-linked blocks, pass 2 can safely write to directories without the danger of damaging unrelated data (so the `scandisk` disaster described in the previous section can't happen).

In pass 3, `e2fsck` iterates over all directory inodes again and checks for lost directories (directories not connected to a valid parent), for loops in directory structure and for hardlinked directories. It fixes these problems and connects lost directories to

`/lost+found` directory. It also checks for a non-existing root directory at the beginning of this pass.

In pass 4, `e2fsck` fixes the link counts of all non-directory inodes and connects lost non-directory inodes to `/lost+found`.

In pass 5, `e2fsck` updates the bitmaps of free blocks and free inodes according to the information found in pass 1. It also updates statistic counters in superblock and group descriptors (counter of free inodes, free blocks and count of used directories). These counters are used in algorithms for inode and block allocation as described in section 6.

In reality, passes 3, 4 and 5 are very fast because they don't read much data from the disk. All information on inodes was collected in pass 1 and all information on directory entries was collected in pass 2. This information is kept in the memory to speed up the next passes. Keeping this information can be somehow problematic, because it can cause `e2fsck` to run out of memory with very large disks — the system administrator should create a large swap partition in this case.

The filesystem checking algorithm is broken into passes for good reliability. Each pass checks and fixes a subset of the metadata on the filesystem and further passes do their work while expecting that previous passes have fixed their data. If filesystem checking were done in just one pass, it would be prone to data corruption: it would be pointless to try to fix directory entries while cross-linked blocks are not resolved (because modifying a directory that seems to have invalid entries could damage a cross-linked file). It would be pointless to try to fix lost or cycled directories while directory entries are not known to be valid.

8.3 Checking the Spad filesystem

Now I will describe the algorithm that I designed and implemented for the Spad filesystem. This algorithm is implemented in the `spadfsck` utility. The Spad filesystem can allocate data and metadata at any place, so the algorithm is a bit more complicated — it cannot scan a predefined area of the disk and believe that it is an inode table.

8.3.1 Magic numbers

To differentiate between metadata and data blocks, spadfs uses technique known as magic numbers. Each metadata block contains a number that identifies the type of the metadata block. When checking the metadata, spadfsck checks the magic numbers on individual blocks. If the magic number mismatches, spadfsck considers the block as data block and doesn't attempt to correct it.

Spadfs has the option (turned "on" by default) to xor the sector number with the magic number. The resulting xored value is stored in the metadata block. The spadfs filesystem can contain another image of the spadfs filesystem in a file (for example, when using virtualized hosts). Xoring the sector number with the magic number allows `spadfsck` to differentiate between real metadata belonging to the top-level filesystem (that should be processed as metadata) and metadata belonging to the filesystem stored in a file (these metadata should be processed as a file content).

8.3.2 Metadata checksums

To alleviate data damage caused by broken chipsets or disks [55] [56], the Spad filesystem contains checksums on all metadata structures. An incorrect checksum prevents the kernel from touching a given structure because it could potentially cause more data damage — for example a flipped bit in a field describing a physical block of a file could cause damage to an unrelated healthy file. If the checksum happens to be incorrect, the kernel marks that the filesystem needs to be checked and `spadfsck` will scan it automatically on the next reboot. Alternatively, the administrator can unmount the filesystem and run `spadfsck` manually without rebooting the machine. `spadfsck` fixes the checksums and treats information in structures with checksum error as "less credible" — so that it tries to minimize damage to healthy structures.

8.3.3 Goals of the filesystem-checking algorithm

Let's specify the following important goals that need to be reached:
1. Memory consumption must be unrelated to filesystem size. With today's many-terabyte disk arrays it is well possible that the number of blocks on a filesystem exceeds the number of bits the in memory — keeping a bitmap of allocated and free blocks in the memory is thus not possible.
2. No damage to healthy data. If the filesystem contains some heavily damaged structures, the algorithm must in no way cause damage to unrelated correct structures. An example of a file that is being treated as a directory and damaged by `scandisk` given in section 8.1 clearly breaks this requirement.

8.3.4 The bitmap of allocated block

To achieve the first goal I use the following techniques:

- The bitmap of allocated blocks the in memory is compressed. The filesystem usually contains long runs of allocated and free blocks and I can use this knowledge to store the bitmap efficiently. I use the following compression algorithm: I store the bitmap in chunks of 32 bytes — each chunk describes 256 blocks on disks. I form a radix tree whose leaves are these 32-byte chunks. An internal node in the radix tree has 256 entries — i.e. it's size is 1024 bytes on 32-bit machines and 2048 bytes on 64-bit machines. If any subtree in the radix tree has only free or only allocated bits, the subtree is removed from the memory and the pointer to it is replaced by `(void *)0` (for free bits) or `(void *)1` (for allocated bits). Tree nodes are collapsed or expanded as the algorithm proceeds. This compression algorithm can consume at most 9/8 of

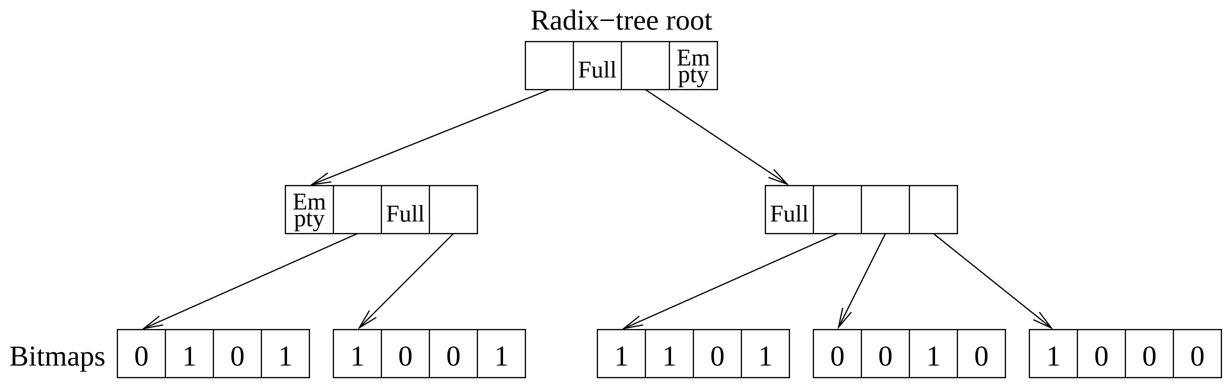

Figure 8.1: Bitmap compression using radix tree

the actual bitmap size (or 10/8 on 64-bit machines) and can compress runs of arbitrary length efficiently. Accessing or modifying any bit is fast.

- If the bitmap can't be compressed efficiently and `spadfsck` wants to use more than the specified amount of memory, the bitmap is swapped out on disk into the filesystem being checked. As I described in section 5, apages have two versions of data — one valid in the case of a crash and the other currently used. Because `spadfsck` runs on an unmounted filesystem, the working part of apages can be freely used. When creating the filesystem, `mkspadfs` creates so many apages that they can contain allocation data for the case of worst possible fragmentation — so `spadfsck` always has enough room to store its bitmaps there. `spadfsck` uses a LRU-caching algorithm to keep frequently accessed bitmaps in cache.[3]
- If the apages on the filesystem are damaged, there is no place to swap the bitmaps. In this case, the administrator can specify an external file or partition for swapping with `--swapfile` option.
- During operation, `spadfsck` doesn't store any other information than addresses of structures from the root directory to the current block. Even in the case of a large directory with many entries, memory consumption is only logarithmically dependent on the directory size (i.e. linearly dependent on tree depth). The checking algorithm works node-by-node and doesn't need information about the whole directory at one time.
- Information about all files with hardlinks (stored in fixed fnode blocks) must be kept in the memory all the time, so that their reference counts can be properly checked. However, information about files embedded in directories is not kept — files are removed from the memory after they are checked.

[3] Note that because of this feature, it is strongly discouraged to run `spadfsck` on a mounted filesystem — even if `spadfsck` doesn't find any errors, it can swap its bitmaps to the filesystem into an area that is possibly used by the kernel. Running `spadfsck` on a mounted filesystem with `-n` flag (meaning read-only access) is allowed because it disables swapping to the filesystem.

8.3.5 Using the bitmap to mark healthy data

Now I will describe how to achieve the second goal — not damaging healthy data. `spadfsck` keeps a bitmap of allocated blocks as described in the previous paragraph, and any filesystem repair activity never modifies blocks that are marked as ”allocated”. Filesystem repair generally works in these steps:
1. Find a new metadata block that is pointed to by some valid block.
2. Validate the new block, possibly fix errors.
3. Mark the new block as ”allocated” in an in-memory bitmap.

`spadfsck` delays processing of pointers from damaged blocks. Imagine a case where one subdirectory pointer has incorrectly flipped bit that makes it point to an already valid directory. If `spadfsck` walked the incorrect pointer first, it would link the correct directory that belonged elsewhere to an incorrect place — damaging healthy data. To prevent this scenario from happening, when `spadfsck` finds some damage in a structure (for example a bad checksum), it delays processing pointers from this structure until all healthy structures are processed. Thus, in the above case with a flipped bit, when `spadfsck` finds a damaged structure, it delays its processing, later finds the healthy directory with correct pointer, and marks the subdirectory as allocated. When it finishes processing all healthy structures, it returns to the damaged structure, finds a pointer to an already allocated space and deletes the pointer. The healthy subdirectory is not damaged.

When `spadfsck` finds cross-linked blocks, it switches to a two-pass walk of the directory tree. In the first pass it marks all metadata as allocated and in the second pass it marks all data as allocated. It also collects paths and filenames of cross-linked objects in order to give meaningful error messages to the user (just like `e2fsck` does). The purpose of this two-pass walk is this: If we have a file that contains an area cross-linked with many metadata structures, it would mark the area as allocated and prevent access to the metadata, causing a large data loss.

8.3.6 Scanning for lost data

When `spadfsck` finishes scanning of the tree, it searches for lost data. Lost data are data that are marked as allocated in apages, but are not connected to the directory tree. `spadfsck` scans all lost blocks for magic numbers. If magic number is found and if the data structure seems consistent, `spadfsck` retains record of the structure in memory and it remembers all the other blocks this structure points to. `spadfsck` builds an oriented graph of all lost metadata structures.

`spadfsck` attempts to extract the tree structures from the oriented graph. It finds metadata structures that no other structures point to and selects them as a possible tree roots. It sorts these roots according to the size of tree underneath the root and attaches them to `LOST+FOUND` directory sequentially from the largest tree to smallest. It checks them (this activity marks the trees as allocated). Then, it finds remaining lost metadata (that did not belong to just recovered trees) and repeats this process until some lost metadata remains.

Scanning for lost data consumes amount of memory that is linearly dependent on the amount of metadata.

8.3.7 Description of spadfsck

First, `spadfsck` checks the superblock and txblock. If the superblock is not found, `spadfsck` tries to load a backup superblock from the end of the device. If the txblock is invalid, it will be recreated at the end.

Then, `spadfsck` checks the crash count table. If some words are invalid, they are set to `0x7fffffff`. If some words past `txblock.cc` are invalid, they are set to `0`. If too many errors are found in a sector, the whole sector is skipped.

Then, `spadfsck` checks the apage index and apages. It doesn't store the information about apage content anywhere, it just checks that the apages contain valid data. It also prepares pointers to unused parts of apages for possible future use for bitmap swapping.

Then, `spadfsck` walks the directory tree (if it finds cross-linked data, it performs two walks as described above). If it finds a corrupted file or directory, it remembers it and doesn't attempt to fix it now. `spadfsck` maintains an in-memory list of files and directories that were skipped and must be recovered. When it finishes walking the tree, it recovers all corrupted directories that were previously found, and walks the tree from them again.

Then, `spadfsck` scans the device for lost blocks. If lost blocks are found, they are added to the tree under `LOST+FOUND` directory and they are checked again.

Then, `spadfsck` fixes all reference counts on hardlinked files.

Then, `spadfsck` recovers all corrupted files found in previous passes.

Next, `spadfsck` creates the crash count table and the txblock if they were corrupted.

Next, `spadfsck` walks apages again and compares them with the allocated block bitmap that it created during previous passes. If it finds differences (or if apages were corrupted), it rebuilds the apages[4].

Finally, error flags in the txblock are cleared.

8.3.8 A proof that the directory checking always creates a tree

In this section I present a proof that the directory checking algorithm in `spadfsck` always creates a directory tree.

The algorithm can be defined in the following way: The device contains metadata nodes, each node points to zero, one or more other metadata nodes. The metadata nodes form a directed graph. In the beginning, nodes point to each other arbitrarily. In the end, I show that all nodes form a tree.

The nodes can belong to three sets, A, B, C. Each node belongs to exactly one set. $N \rightarrow M$ means that the node N points to the node M.

[4] Rebuilding all the apages is noticeably simpler than trying to modify them to match an in-memory bitmap — so they are rebuilt anyway even if small differences are found.

The sets are implemented in the following way: nodes in the A and B set are marked as used in `spadfsck` bitmap. The B set contains nodes in the `todolist` variable in `spadfsck`. The nodes not marked as used in the bitmap belong to the set C.

The simplified algorithm can be described as following:

```
• 1. put the root node to the $B$ set and all the other nodes to the $C$ set
• 2. while $B \cup C \neq \emptyset$; do begin
•      3. while $B \neq \emptyset$; do begin
•           4. take node $N$ such that $N \in B$
•           5. for each node $X$, where $N \to X$
•                6. if $X \in A \cup B$
•                     7. delete the pointer from $N$ to $X$
•                8. if $X \in C$
•                     9. move $X$ from the $C$ set to the $B$ set
•           10. move $N$ from the set $B$ to the $A$ set
•      11. end
•      12. if $C \neq \emptyset$; then begin
•           13. take node $N$ such that $N \in C$
•           14. create a pointer from the root node to the node $N$
•           15. move the node $N$ from the $C$ set to the $B$ set
•      16. end
• 17. end
```

In step 1, if there is no root node or root node is severely corrupted, an empty root is created and the algorithm follows normally. Steps 3–11 are normal directory check, steps 12–16 represent scanning for lost data (if some lost data exist) and attaching lost data to the main tree.

The following invariants hold across the whole algorithm.

- 1. for each node $N \in A$, for each node M such that $N \to M$: $M \in A \cup B$

Proof: we add node N to the A set only in the step 10. In steps 5–9 we checked all the nodes that N points to; we deleted pointers to nodes in A and B sets and converted C nodes to the B nodes. Thus, in the step 10, the node N points only to nodes in the B set. The algorithm moves nodes only from C to B and from B to A, so once we verified that a new node N added to A points only to nodes in B, in the future, this node will point to nodes in A or B. This proves that every A node points only to A nodes or B nodes.

- 2. for each node $N \in A \cup B$, N is not root: there exists exactly one node $M \in A$ such that $M \to N$

Proof: we add nodes to the B set in steps 1, 9, 15. In the step 10 we move a node from the B set to the A set.
In step 1, we add the root node (which is excluded in this invariant).
In step 9, we take a node X from the C set and add it to the B set. When X was in the C set, there was no node in A that pointed to X (invariant 1). If we add node X to the B

set (step 9), and N to the A set (step 10), we create exactly one pointer $N \to X$ for node X.
In step 15, we take a node X from the C set and add it to the B set. When X was in the C set, there was no node in A that pointed to X (invariant 1). We add X to the B set and create exactly one pointer in the step 14.
We proved that when we added a node X to the B set, there was exactly one pointer from the A set to the node X. Now we must show that the pointer won't be deleted or another pointer to the node X won't be created:
The algorithm only adds a pointer in step 14 (in this case, the target of the pointer is not in $A \cup B$). The algorithm only deletes a pointer in step 7 (in this case, the origin of the pointer is not in the A set).

- 3. there is no node $N \in A$ such that $N \to root$

Proof: This invariant holds in the beginning (because the A set is empty).
The root is the first node that is added to the A set (this can be seen by tracing the first run of the algorithm from steps 1 to 10).
Now we show that the pointer to root from some node in the A set cannot be added:
The algorithm only adds a pointer in step 14. In this case, the target of the pointer is not in A.
The algorithm only moves a node to the A set in step 10. In this case, the node has no pointers to some node in the A set (if there were some, they were deleted in steps 5–9).

- 4. there exists a path from the root to every node in the A set

Proof: This invariant holds in the beginning (because the A set is empty) and holds when the root is added as the first member of the A set.
Now, we assume that the invariant holds for the A set and prove it inductively:
We only add a node N to the A set in step 10. The node N was previously in the B set. From invariant 2 we know that there exists exactly one node $M \in A$ such as $M \to N$. There exists a path from root to the node M (this is inductive assumption) and there exists a pointer $M \to N$. Therefore, there exists a path from root to the node N. The node N is just added to the set A. Thus, we prove that invariant 4 holds for the set A.

The algorithm is finite.
Proof: The loop 5–10 is finite because there is a finite number of nodes X where $N \to X$. In both bodies of loops 3–11 and 2–17, the algorithm will either move a node from the set B to the set A or move a node from the set C to the set B. It doesn't perform any other moves. Thus, $2|C| + |B|$ can be seen as a value that always decrements by at least one when bodies of these two cycles are executed.

At the end of the algorithm, sets B and C are empty. Thus, from invariants 2, 3, 4, we can conclude that the nodes in the set A form a tree.

8.4 Conclusion

In this section I outlined general requirements for the filesystem checking utility. I described the algorithm used in Ext2 and Ext3 filesystems and I described the algorithm for checking the Spad filesystem. The algorithms for filesystem checking are hardly ever published.

`spadfsck` performance is linearly dependent on the amount of metadata it processes. `spadfsck` memory consumption is not dependent on the size of the filesystem if compression or swapping is used. `spadfsck` memory consumption is linearly dependent on the size of the filesystem if swapping could not be used or when scanning for lost metadata.

9. The system architecture and filesystem features

In this chapter I will present a summary of features of the filesystems that will be experimentally evaluated: The implementation of the SpadFS filesystem on two operating systems (Linux and Spad) as well as other popular Linux filesystems — Ext2, Ext3, Ext4, ReiserFS, XFS and JFS. Spad is a new experimental kernel, some of its architectures is published in [57]. An architecture and experimental evaluation is published in [58].

9.1 System architecture of the VFS

Today's operating systems support many filesystems. Filesystem code is split into a filesystem-dependent part (i.e. filesystem driver) and a filesystem-independent part (often called VFS — Virtual File System) to make development and maintenance easier. VFS is specific to a given operating system and it is used by all filesystem implementations running on that system. Over the past years, more and more functionality has been moved from the filesystem-dependent part to VFS to process data in a unified way [59] [60]. Current operating systems have a file and directory cache in VFS and do physical I/O in VFS. The filesystem-dependent part only maps logical blocks in a file to physical blocks on the disk and any further reading or writing is done by VFS without any interaction with the filesystem driver.

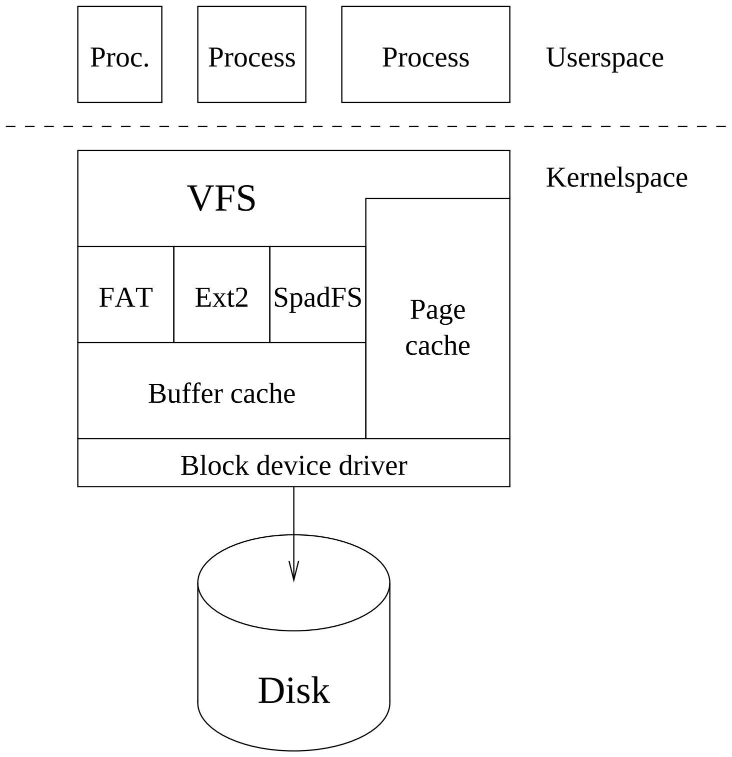

Figure 9.1: System architecture

Thus, the performance of VFS becomes even more important than the performance of the filesystem driver. I benchmarked filesystems with the same data layout (SpadFS, Ext2) on two operating systems' VFS implementations — Linux 2.6 and Spad, to show the impact of VFS on filesystem performance.

The Spad is a kernel written from scratch, allowing novel features to be implemented in it. I implemented fully delayed allocation and cross-file readahead in the VFS of the Spad kernel. The Spad VFS uses extents consistently when passing information between VFS and the filesystem driver, so it doesn't suffer from the performance penalty caused by small block size. These features would be hard or impossible to implement in the Linux kernel because Linux supports many filesystems and making changes to VFS would result in modifications of all the filesystem drivers.

9.2 Filesystem features

In table 9.1 you can see individual features of each tested configuration.

- **Fast recovery** — The possibility to quickly recover after a crash without scanning the whole filesystem. On most filesystems this is accomplished by using a journal. SpadFS implements a different method — crash counts.
- **Indexed directories** — The ability of a filesystem to quickly find a file in a large directory without scanning the whole directory. In most filesystems BTrees or B+Trees are used. They may be used either directly with the filename as a key (JFS) or in such a way that the filename is hashed and the hash value is used as a key to BTree (Ext3, ReiserFS, Reiser4, XFS). SpadFS uses a different method — a slightly modified variant of Fagin's extendible hashing.
- **Extents** — The filesystem stores file allocation information in triplets (*logical block, physical block, size*) rather than having a list of all physical blocks forming a file. Extents significantly reduce the space needed for metadata of large files. The effect of extents on filesystem performance has been experimentally evaluated by Z. Zhang, K. Ghose [61].
- **Optimized small files** — The filesystem can efficiently store a large amount of small files. ReiserFS and Reiser4 can store file content directly in the tree so that it consumes exactly the required amount of space; no padding to a block size is done. In Spad VFS I made a different design decision to achieve efficiency for small files — I allowed the filesystem to run with a block size of 512 bytes — thus small files are padded up to 512-byte blocks, unlike the 4096-byte blocks most commonly used in Linux filesystems. Unlike Linux, the Spad VFS uses extents internally in the memory; there is no structure having lists of a file's blocks — so the Spad kernel performs well regardless of a filesystem block size. When SpadFS is created on the disk with 4096-byte sectors, 4096-byte sectors are used. When SpadFS runs on the top of the Linux kernel, it is recommended to use 4096-byte blocks because the Linux kernel page cache handles small blocks inefficiently.
- **Delayed allocation** — Most filesystems allocate space on disk when syscall `write()` is called. This makes the syscall slow. An obvious improvement is to allocate space on disk when the cache is about to be flushed, not when the data were put in the cache with `write()` syscall — this is called delayed allocation. Delayed allocation has two advantages: it reduces the time of cached writes significantly (as will be seen in

benchmarks in the next section) and it reduces file fragmentation, because continuous space can be allocated for file size exactly. The XFS has delayed allocation for file data. Ext4 has delayed allocation in experimental patches that have not yet been committed to an upstream kernel (the upstream version of Ext4 was benchmarked, so the effect of delayed allocation is not seen in the benchmarks).

Reiser4 has delayed manipulation of a tree — tree entries are created at the time of syscall but the nodes are allocated when the filesystem is flushing the cache. Unfortunately Reiser4 gets this (and other) features by reimplementing a lot of work that is expected to be done in VFS, which lead to the refusal of Linux kernel developers to include it in the standard kernel [100].

In Spad VFS I took this approach further and moved the delayed allocation framework from the filesystem-dependent driver to the filesystem-independent VFS. Spad VFS can delay almost all operations — it has delayed allocation of file data, delayed adding of a file's directory entry to a directory, delayed deletion of files and delayed allocation of whole directories — these features are integrated in Spad VFS and thus any filesystem on top of it automatically uses them. Delayed allocation of whole directories enables another novel feature: continuous allocation of a directory. When the filesystem is about to allocate space on the disk, it sums the size of all new files in a directory and allocates continuous space for all of them. This improves efficiency of another feature — cross-file readahead.

- **Cross-file readahead** — The ability to read ahead across boundaries of individual files. Read ahead within files is implemented in all current operating systems. I widened this principle, doing read ahead when sequentially reading directories containing many small files. As with delayed allocation, this feature is integral part of Spad VFS and thus all filesystems on top of it benefit from it.

  Aggressive readahead has already been envisioned [62] [63] [64] [65] [66]. It is based on the fact that the transfer speed of devices increases more and more while seek time remains roughly the same; thus the cost of readahead decreases. My algorithm works in a simple way: if the file being read is smaller than 32KiB, I read data up to the next 32KiB boundary past the file end. If access to some other file hits the prefetched area, I read ahead following 64KiB data with two separate 32KiB requests (an experiment showed that reading more doesn't increase performance — that experiment was done by running `grep -r` command on a Linux kernel source tree; I don't present the results of that experiment because the setup where this requirement was observed is gone — the disk died and was replaced and the computer was upgraded with faster CPU, more memory and better a IDE controller).

  On the Linux version of SpadFS I implemented only read ahead of metadata. Read ahead of file data would need redesign of the Linux page cache.

  Reiser4 has a different kind of cross-file read ahead — it attempts to read the tree in advance if it detects sequential access to it. It does not attempt to read ahead file data though.

	Fast recovery	Indexed directories	Has extents	Optimized small files	Delayed allocation	Cross-file readahead
Linux/Ext2	No	No	No	No	No	No
Linux/Ext3	Yes	Optional	No	No	No	No
Linux/Ext4	Yes	Optional	Optional	No	Experimental	No
Linux/ReiserFS	Yes	Yes	No	Yes	No	No
Linux/Reiser4	Yes	Yes	Yes	Yes	Yes	Tree only
Linux/XFS	Yes	Yes	Yes	No	File content	No
Linux/JFS	Yes	Yes	Yes	No	No	No
Linux/SpadFS	Yes	Yes	Yes	No	No	Metadata only
Spad/SpadFS	Yes	Yes	Yes	Partial	Yes	Yes
Spad/Ext2	No	No	No	No	Yes	Yes

Table 9.1: Filesystem features

10. Experimental evaluation of filesystem performance

In this chapter I will present the experimental evaluation of the SpadFS implementation on two operating systems (Linux and Spad) as well as evaluation of other popular Linux filesystems — Ext2, Ext3, Ext4, ReiserFS, XFS and JFS.

Reiser4 was initially meant to be benchmarked; however, a serious performance bug in Reiser4 was found — Reiser4 corrupts the limitation of a number of dirty pages in the kernel. When Reiser4 filesystem was mounted, the kernel stopped enforcing the limit specified in the file `/proc/sys/vm/dirty_ratio` and allowed the whole memory to be consumed by dirty pages (as could be seen in `/proc/meminfo`). What's worse, when Reiser4 was unmounted, this corruption persisted until reboot; any other filesystem mounted after Reiser4 had its amount of dirty pages not properly limited. Using Reiser4 damaged performance of any other filesystem used, therefore it was dropped from the benchmarking and I don't recommend using Reiser4 in a production environment.

I benchmark different filesystems on the same operating system kernel as well as the same filesystem on different operating systems. Thus I can compare the impact of physical data layout and kernel interface design on the performance of various filesystem operations. SpadFS and Ext2 were benchmarked twice — once on the Linux kernel and once on the Spad kernel. The filesystems have the same layout of disk structures on both operating systems, but different code was driving it.

10.1 Benchmarking methodology

With rising memory sizes, a lot of data is kept in cache. Thus, the performance of cached operations is often even more important than performance of raw disk I/O speed — so both cached and uncached operations will be benchmarked.

For server applications running many processes, I/O throughput is not the only factor that matters. Another important feature is the CPU consumption — i.e. how much CPU time the process performing filesystem I/O consumes and how much CPU time it leaves to other processes. Both throughput and CPU consumption will be measured. Operating system tools such as "top" or "time" commands or `clock()` function are not precise enough to measure CPU consumption (they work by looking at a process state in 10ms intervals and computing approximate CPU consumption) so, instead, I run two threads, the first thread performing I/O and the second thread spinning in a loop and incrementing single variable. From the number of iterations done by the second thread I can compute with almost CPU tick-level granularity how much time the first thread consumed. The second thread has priority lowered to minimum, so that it does not take processing power from the main thread when doing a CPU-intensive workload. I disabled hyperthreading so that the measuring is accurate. CPU consumption measured this way represents the total amount of CPU time consumed by all kernel code.

In some other benchmarks CPU consumption was measured using the oprofile program. This includes only CPU spent in the filesystem driver. Oprofile issues periodic interrupts at a rate of 1000 per second and records the module that was being executed.

CPU consumption is measured only for raw uncached I/O benchmarks. For cached benchmarks, CPU consumption is obviously 100% and disk utilization is 0% because the benchmark reads all the data from the cache. I do not report CPU consumption as percentage, but rather as seconds of time consumed. Percentage reporting is misleading — the filesystem that can finish the task faster would report a higher percentage of CPU time consumed.

10.1.1 Hardware and software configuration

Benchmarks were performed on a Pentium 4 processor running at 3GHz with 16kiB L1 cache, 2MiB L2 cache and 1GiB RAM. The computer has ASUS mainboard with ICH-7 chipset, dual DDR-2 533 memory modules and four hard disks (160GB Wester Digital Caviar SATA disk, 80GB Maxtor SATA disk, 80GB Maxtor IDE disk and 80GB ExcelStor IDE disk).

Linux benchmarks were done on SUSE Linux 10.0 with ReiserFS root partition with generic kernel 2.6.26.3. Spad benchmarks were done on the current development version with SpadFS root partition. Root partitions were placed on Western Digital disk.

Tests were done on a clean 20GB partition located at the beginning of Maxtor SATA disk, reformatted to a particular filesystem before each set of tests.

Multi-disk benchmarks were performed on one to four disks in RAID0 array with stripe size 64kiB. These partitions were used in this order:

- 20GB partition at the beginning of ExcelStor IDE disk.
- 20GB partition at the end of WD SATA disk.
- 20GB partition at the end of Maxtor IDE disk.
- 20GB partition at the end of Maxtor SATA disk.

Each disk has a transfer rate of approximately 40-50MB/s at the place where the test partition is located. The ExcelStor disk is slower than the other disks, so the partition was deliberately placed at the beginning to compensate for this.

All filesystems were created with default settings, except that SpadFS metadata checksums were disabled (all the other filesystems don't have metadata checksums, thus I did not want to give SpadFS an unfair disadvantage). Default setting for SpadFS is a 512 byte block size on Spad and a 4096 byte block size on Linux — these default values were used. This distinction comes from the fact that Spad VFS works with extents everywhere, thus increasing block size has no performance advantage and only wastes disk space. Linux VFS, on the other hand, works with lists of blocks and increasing block size lowers CPU usage tremendously.

10.1.2 Benchmarks performed

I performed the following microbenchmarks showing the performance of particular filesystem functions:

- reading a small file from the cache

- writing a small file to the cache
- opening a file with varying directory level depth
- uncached read of large file
- uncached write of large file
- uncached rewrite of a large file (i.e. writing a file that is already allocated)
- uncached random read in a file
- uncached random write in a file

These standard benchmarks were run:

- postmark 1.51
- ffsb 5.2.1

I performed some benchmarks consisting of real-workload:

- extraction of a tar archive
- copying a directory tree
- reading a directory tree
- comparing two directory trees
- deleting a directory tree

All the benchmarks were performed 5 times and the average value is reported.

Cached microbenchmarks were performed repeatedly for 5 seconds and the average time is reported. Before measuring, the benchmarked action was performed twice to fill disk caches and CPU caches, so that cache filling doesn't impact the reported value.

Uncached write/rewrite/read of a large file ran for about three minutes — long enough that small delays due to input from network couldn't affect the overall value.

10.2 Results

10.2.1 Cached read

In figures 10.1 and 10.2 we can see the time required to read from the cache. Because reading from the cache does not depend on filesystem code at all and depends only on VFS code, we can see the only difference between Spad and Linux VFS's, not between actual filesystems. For small files Spad is 13% faster than Linux. We can clearly see the performance degradation when a 2MiB CPU cache fills up. From this point on, the performance does not depend on the operating system at all.

The difference between the values in the individual runs and the average was in most cases less than 1%, in a few isolated cases it was up to 20%.

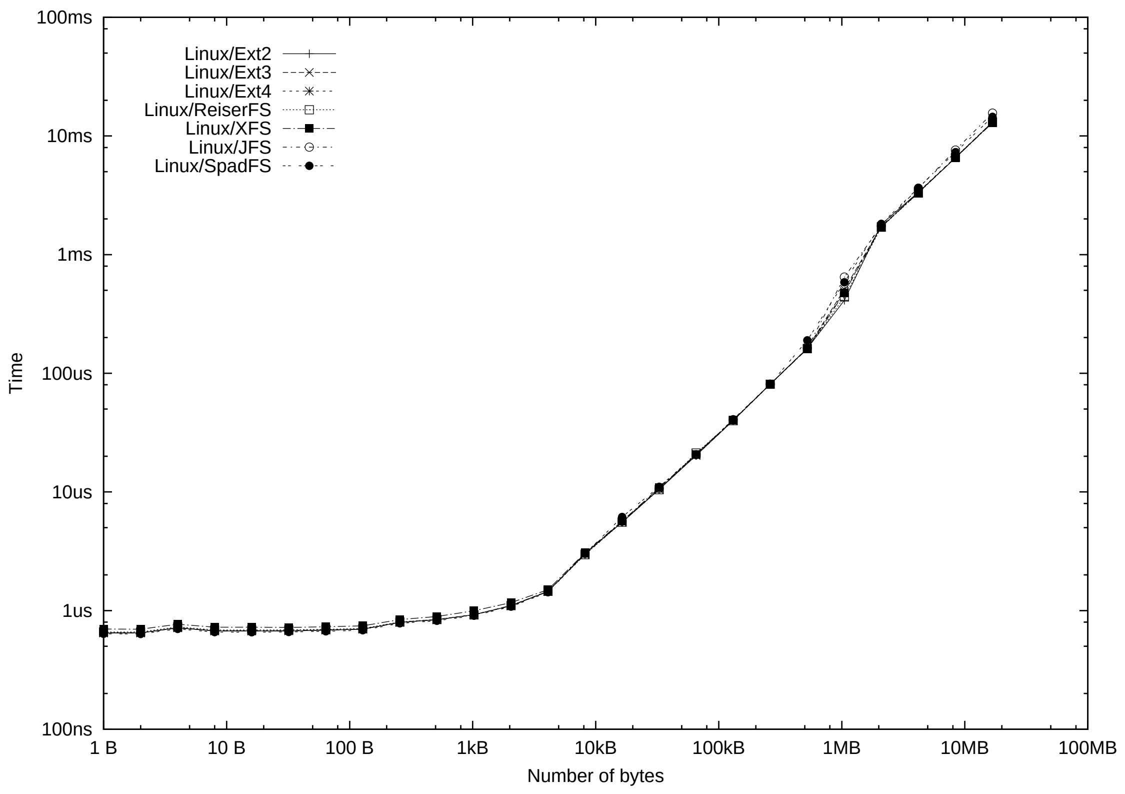

Figure 10.1: Time to read a file from cache depending on file size

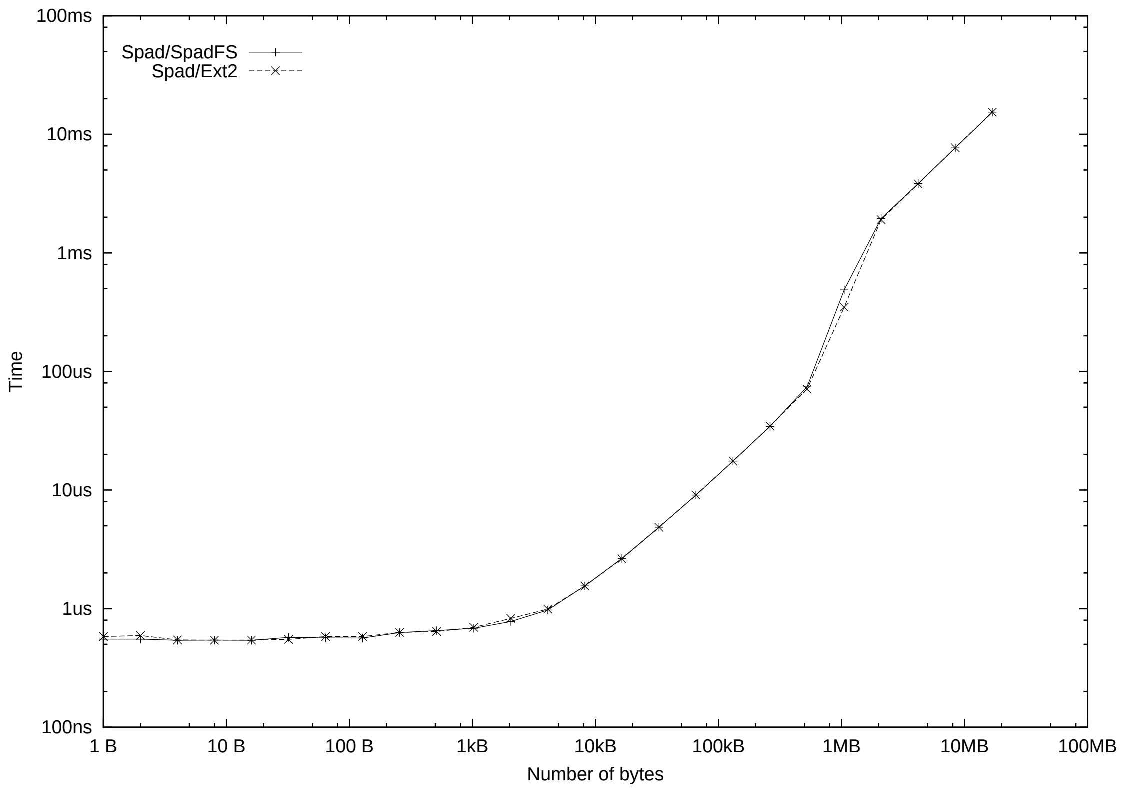

Figure 10.2: Time to read a file from cache depending on file size

10.2.2 Cached write

Figures 10.3 and 10.4 present a similar benchmark — time to write to the cache. Most filesystems do allocation of blocks at this stage, so the time is dependent on actual filesystem implementation — it depends on the speed of its block allocator.

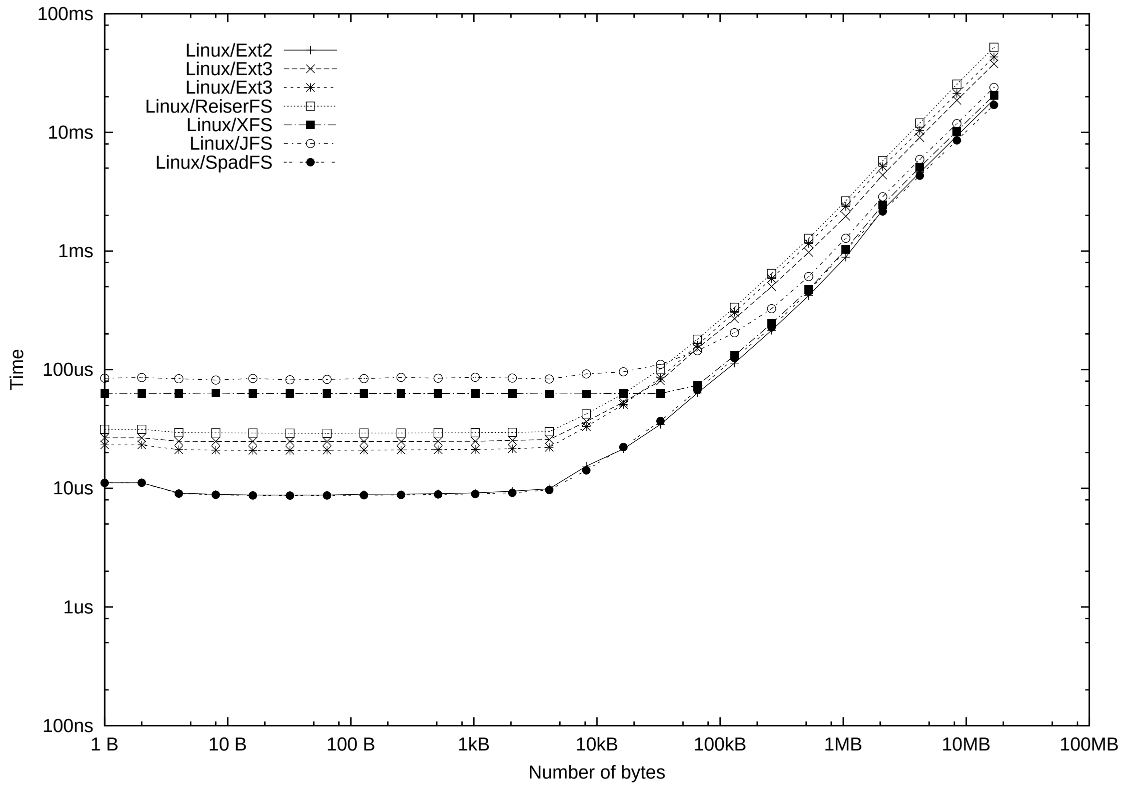

Figure 10.3: Time to write a file to the cache depending on file size

The difference from the average was in most cases less than 1%, Ext3 showed a difference of 4%, JFS 5%, XFS 3%.

On smaller file sizes Spad VFS clearly wins, being 8 times faster than the fastest Linux filesystem, Ext2. It is because Spad does not do any block allocation at this stage; it does delayed block allocation when physically writing data.

As expected, the performance of Spad/SpadFS and Spad/Ext2 is the same because Spad VFS does not call any filesystem-specific code in this scenario.

The XFS has delayed allocation, too, but it scores poorly at this benchmark.

Again, once the L2 cache is filled, the benchmark does not depend on filesystem or operating system at all.

10.2.3 Cached open

The last cached benchmark (Fig. 10.5 and 10.6) — I measured the time to walk the directory tree when opening a file. A directory structure of a depth of up to 60 subdirectories was created and a file in this structure was opened.

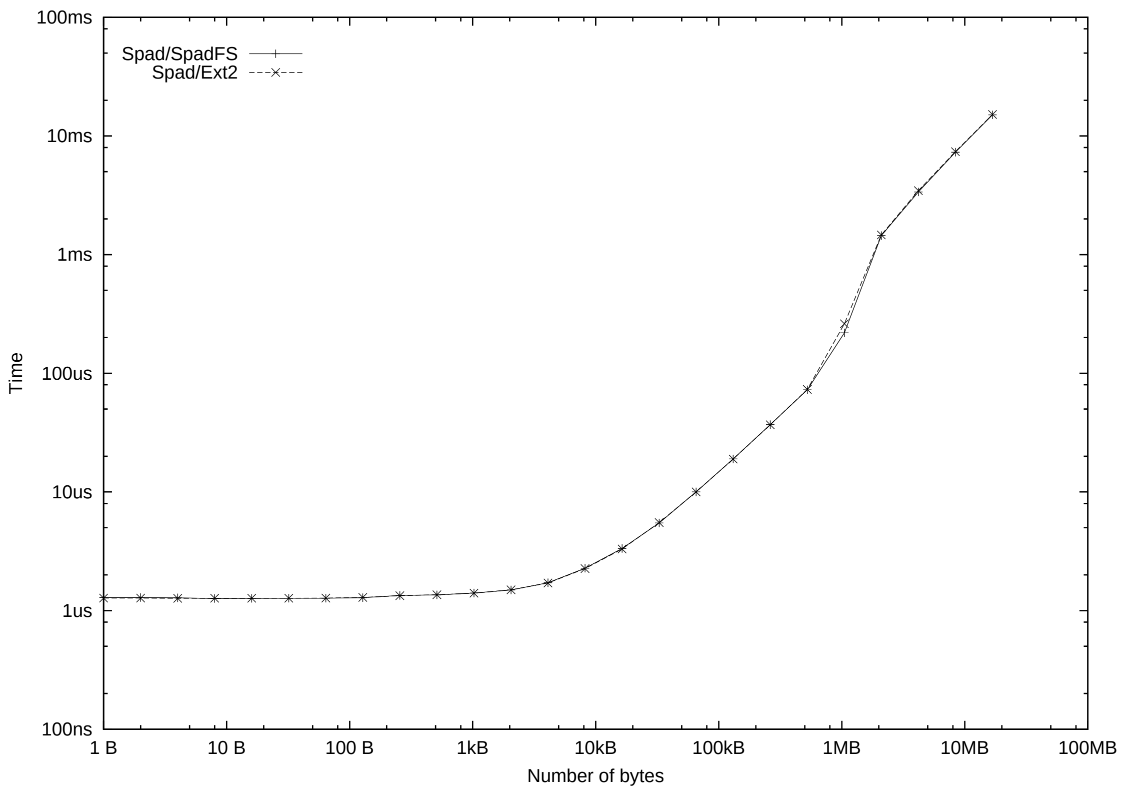

Figure 10.4: Time to write a file to the cache depending on file size

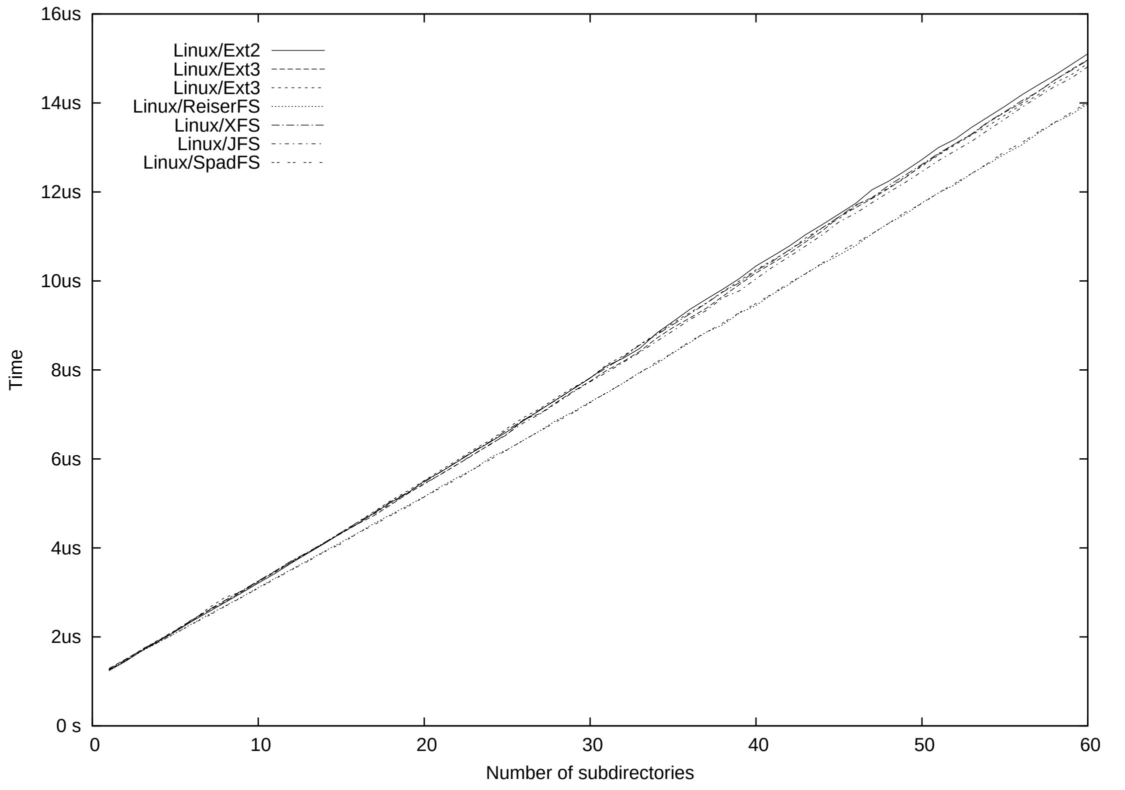

Figure 10.5: Time to open a file depending on the number of subdirectories in the path

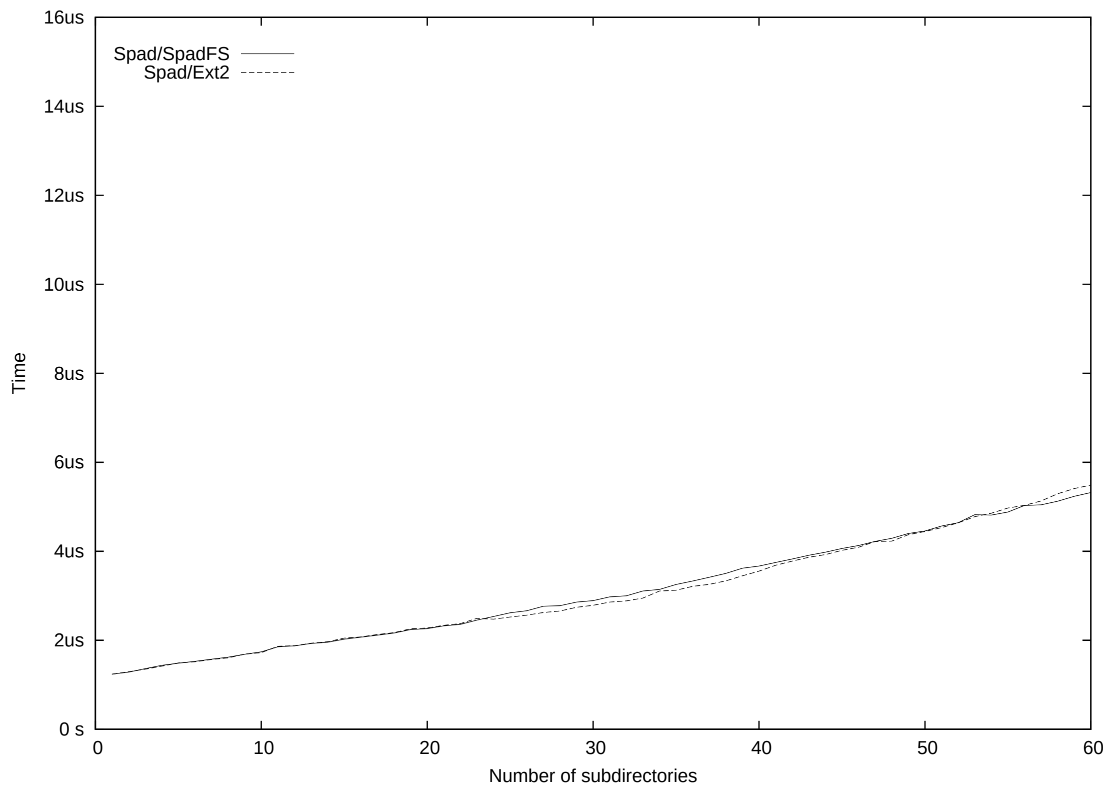

Figure 10.6: Time to open a file depending on the number of subdirectories in the path

The usual difference from the average was 1% on the Linux kernel and 2% on the Spad kernel.

This is, again, a cached operation, so it does not depend on the layout of data on the disk.

As expected, all Linux filesystems have the same speed. Spad VFS is 2.8 times faster than Linux VFS.

10.2.4 Raw I/O throughput

Next, I measured uncached operations (Fig. 10.7, 10.8, 10.9, 10.10) — I created a 8GiB file, rewrote the file sequentially without truncating it and read it sequentially. To test random access performance, I performed read and write on the 8GiB file at random positions; 16 threads were performing requests each having 64kiB; each thread performed 1024 requests total.

The throughput of filesystems can be seen on these charts. The charts represent throughput for one disk, a two-disk RAID0 array, three-disk RAID0 array and four-disk RAID0 array with 64kiB chunk size.

The difference from the average value was in most cases about 1%, in a few isolated cases up to 15%. Ext2/Spad showed up to a 35% difference on a raid array.

Ext2, Ext3 and ReiserFS clearly lose on rewrite operation — it is because they contain a table of all allocated blocks instead of a list of extents and they need to read extra metadata to determine locations of blocks that need to be rewritten. Spad VFS shows a slightly lower write throughput compared to the same data layout on Linux VFS (12% lower on one disk and 2% lower on the 4-disk array).

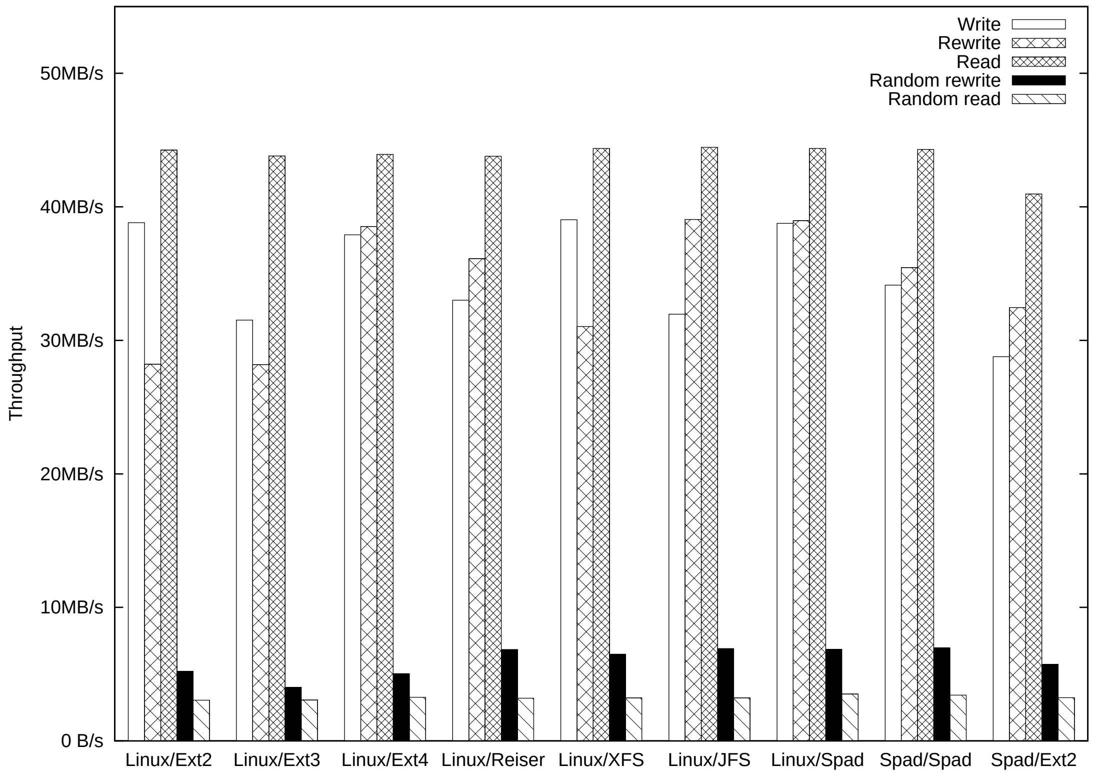

Figure 10.7: RAW I/O throughput of filesystems on one disk

However, raw throughput is not the only important factor. Another issue is CPU consumption. CPU consumption of the same operations is shown in figures 10.11, 10.12, 10.13, 10.14. We can see Ext3, Ext4 and ReiserFS clearly losing for writing. Spad VFS and SpadFS have the lowest consumption. It can be seen that CPU consumption during reading depends more on the VFS layer than on the filesystem itself — it is because in all modern operating systems reading is done by the kernel with very little interaction with the filesystem driver — the filesystem driver only passes block numbers to the kernel.

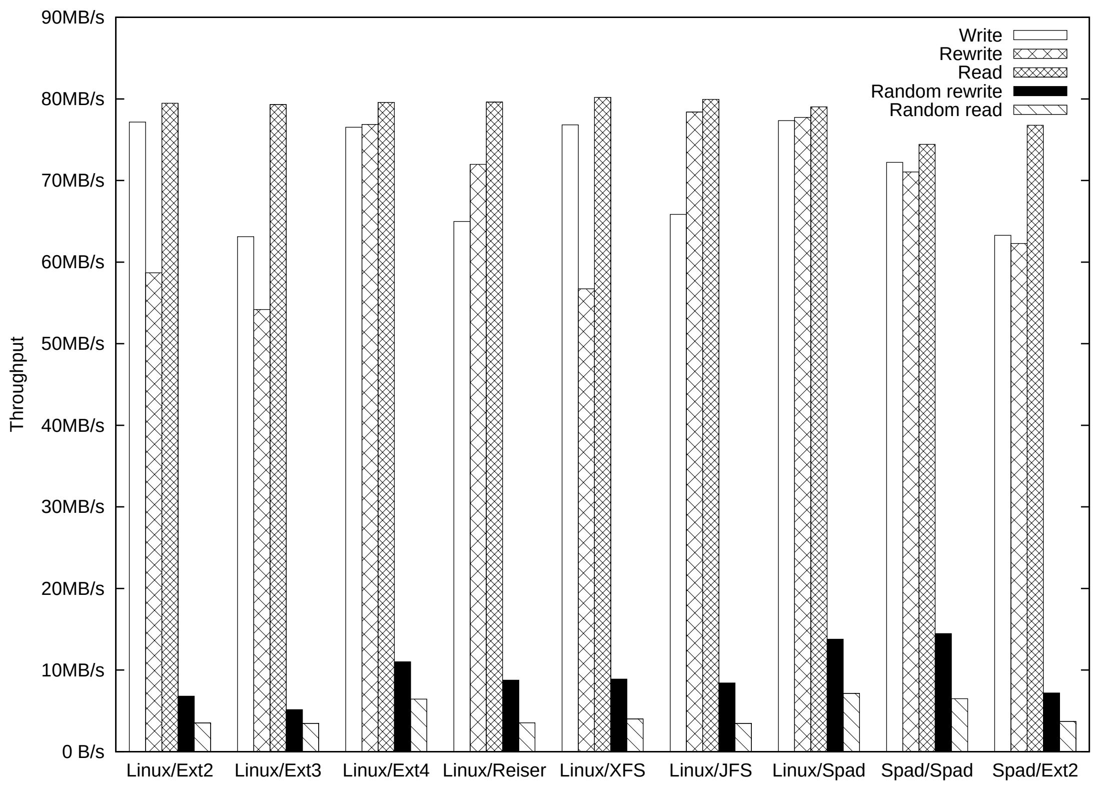

Figure 10.8: RAW I/O throughput of filesystems on a two-disk array

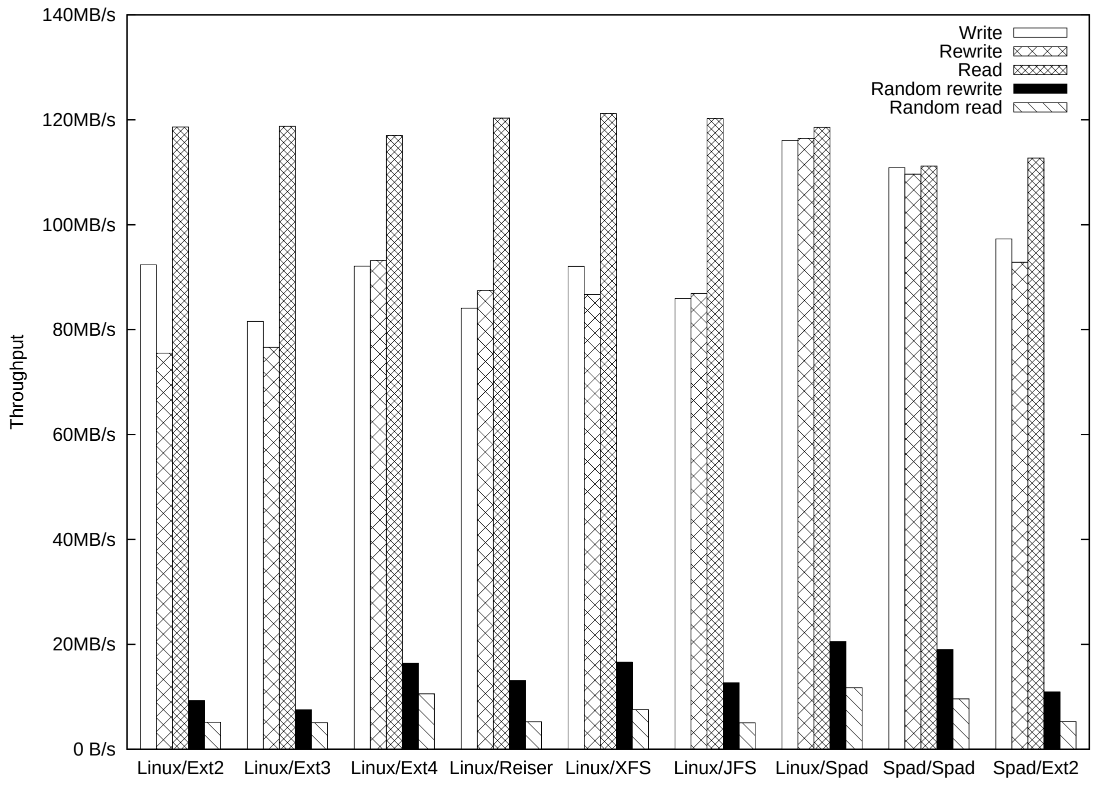

Figure 10.9: RAW I/O throughput of filesystems on a three-disk array

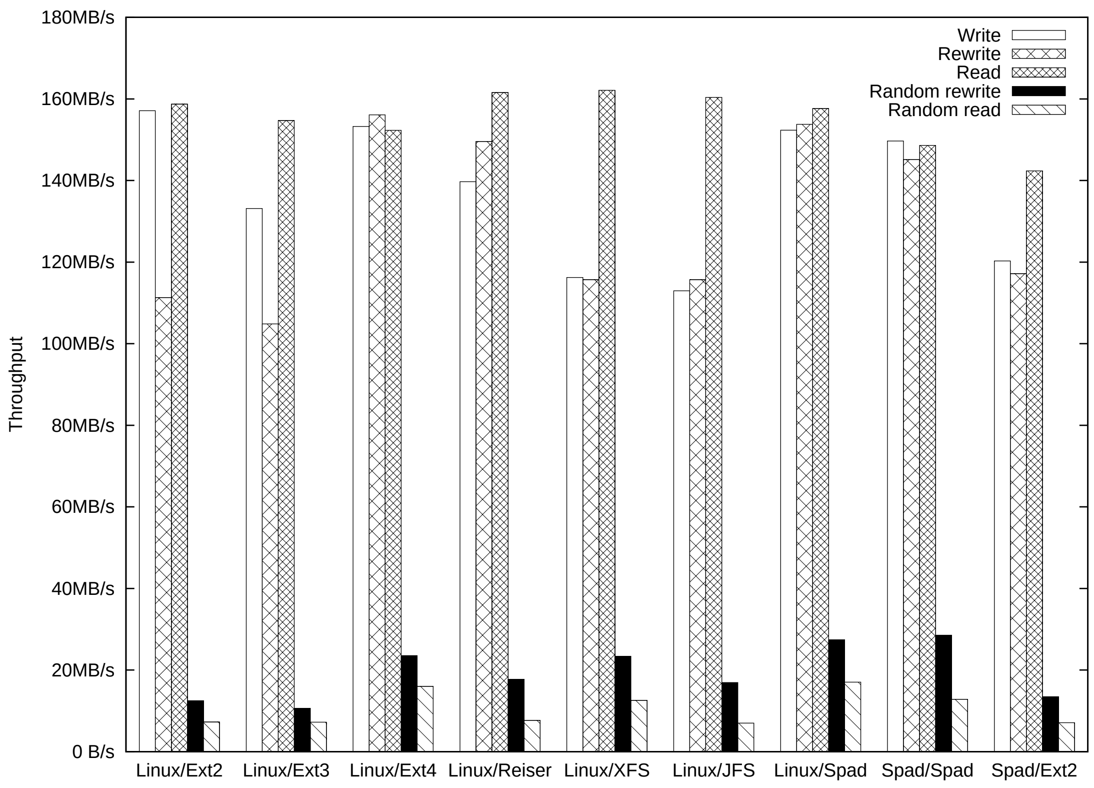

Figure 10.10: RAW I/O throughput of filesystems on a four-disk array

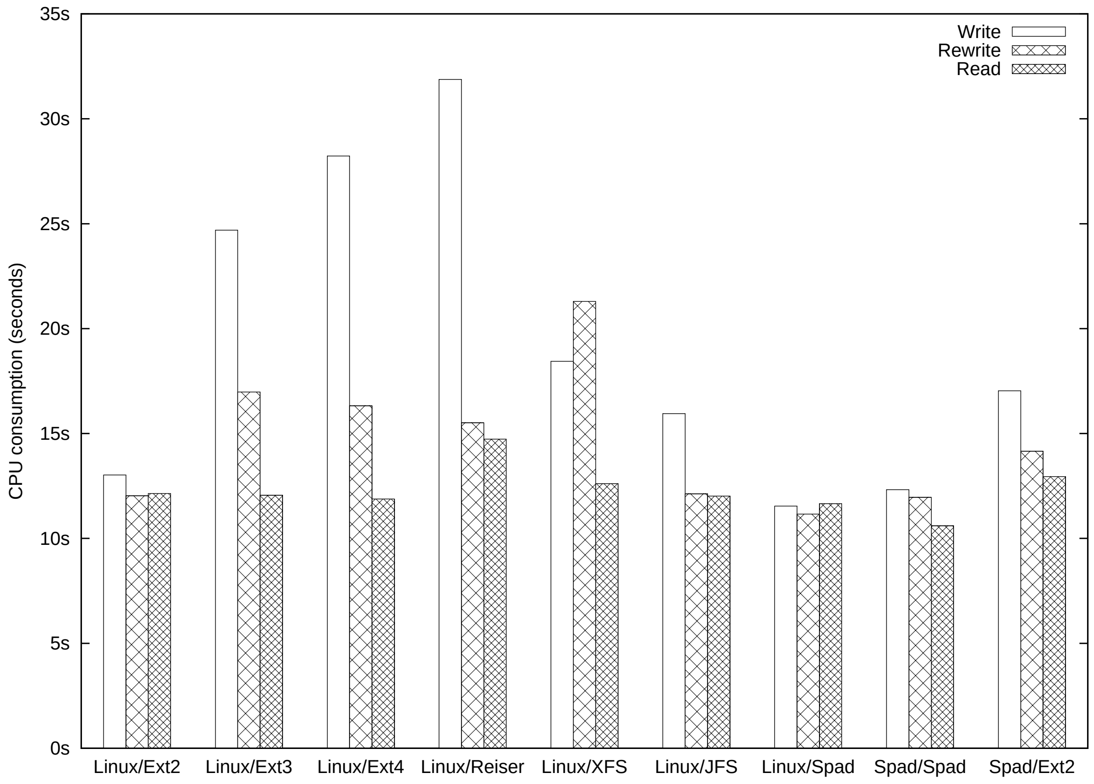

Figure 10.11: CPU consumption of filesystems on one disk

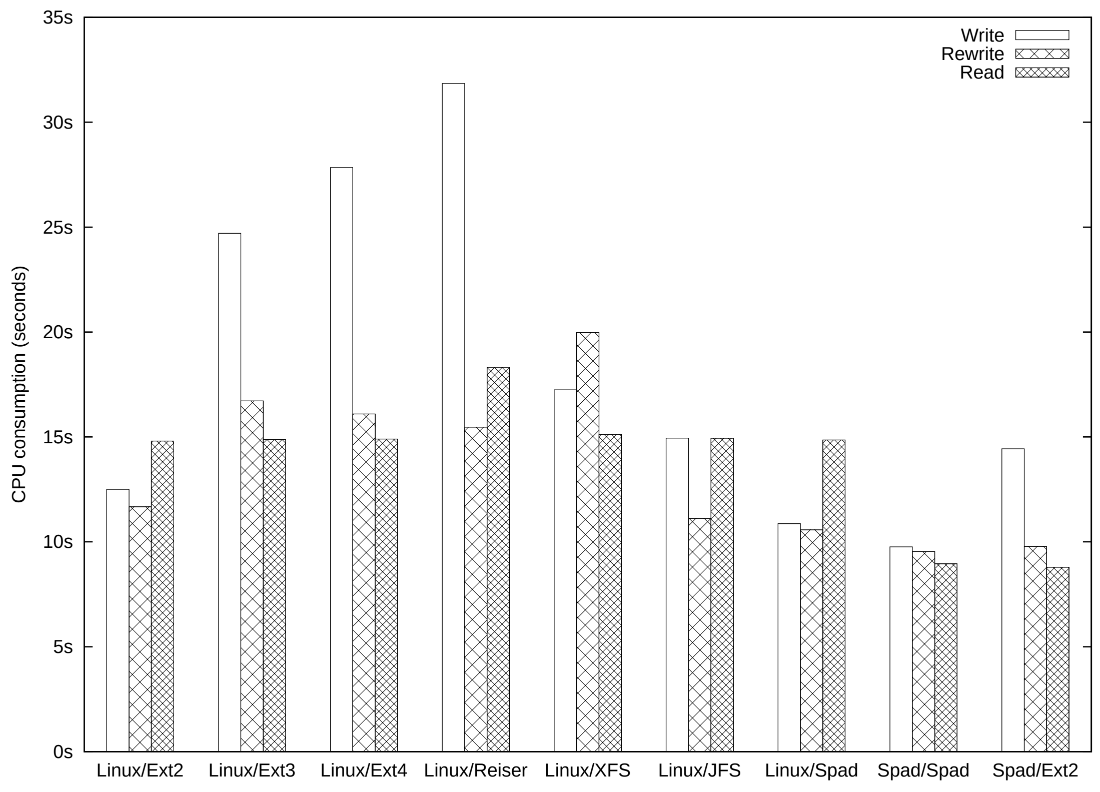

Figure 10.12: CPU consumption of filesystems on a two-disk array

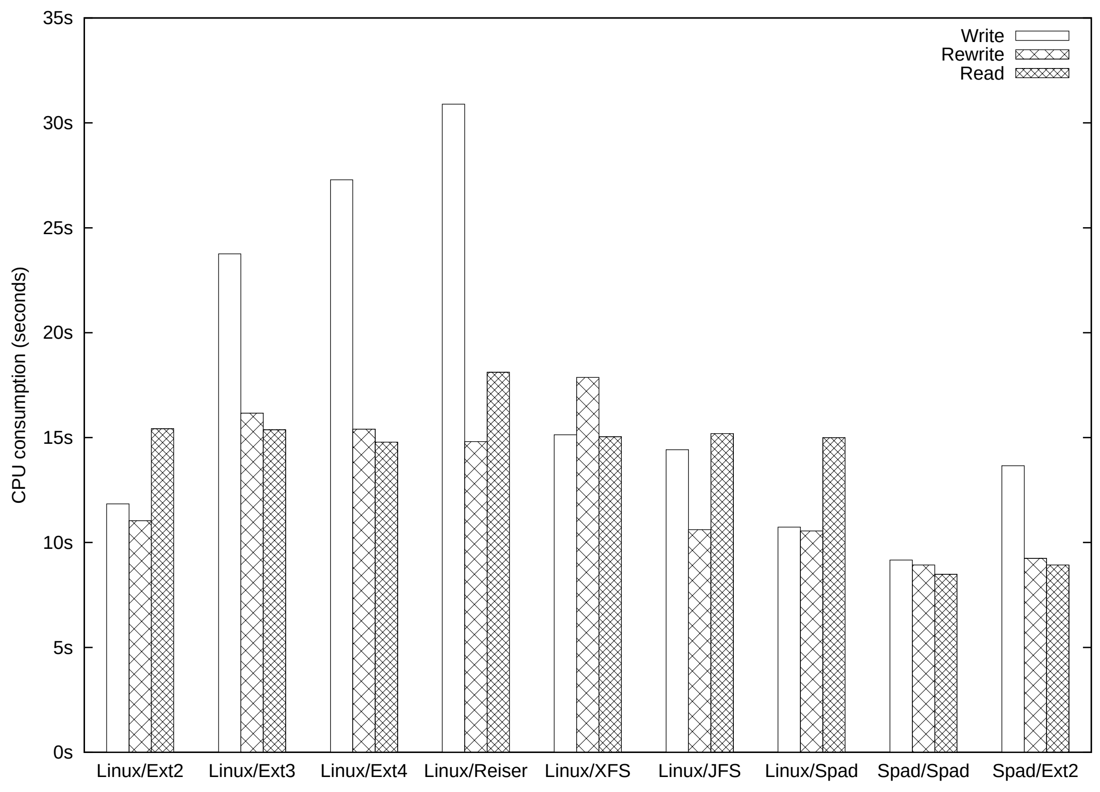

Figure 10.13: CPU consumption of filesystems on a three-disk array

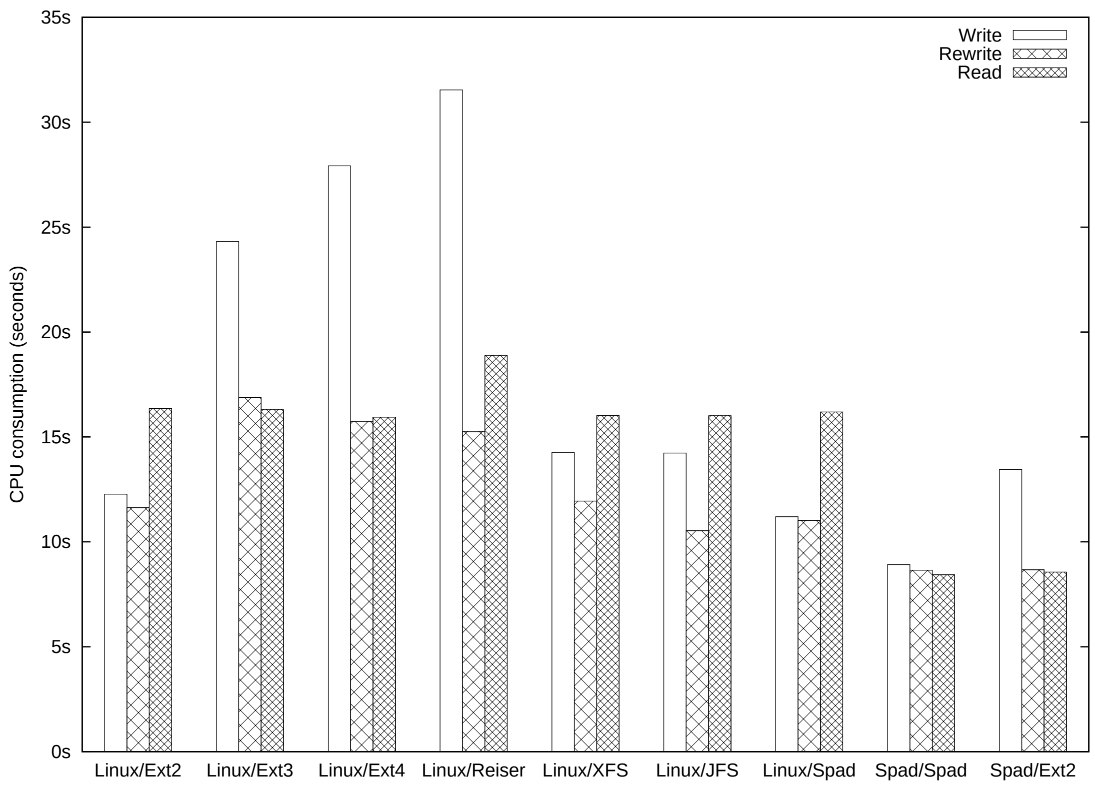

Figure 10.14: CPU consumption of filesystems on a four-disk array

10.2.5 CPU consumption benchmarks

To show the scalability of filesystems with respect to increasing device speed, I performed write/rewrite/read benchmarks (as in section 10.2.4) on devices with increasing speed and with increased file size. The devices were constructed from 1 to 4 20GB partitions on separate disks in RAID0 with a 64kiB chunk size. The transfer speed of the devices was 40, 80, 120 and 160MB/s (as can be seen in section 10.2.4). In this benchmark, the files had a size of 8GiB, 16GiB, 24GiB and 32GiB bytes (this differs from the benchmark in section 10.2.4 where all the files had 8GiB size).

CPU consumption was measured with the oprofile tool. All the filesystems were compiled as modules and oprofile showed how much CPU time was spent in each module. Oprofile makes a sample each 1ms. The reported value is the number of samples that were sampled in a given module. For ext3, the times for "ext3" and "jbd" modules were summed (because the filesystem driver consists of these two modules); for ext4, the times for "ext4dev" and "jbd2" were summed.

Figure 10.15 shows the time spent in the filesystem driver when writing files; figure 10.16 shows the time spent when rewriting files and figure 10.17 the time spent when reading files.

In the write test, the difference from the average value was less than 18%. In the rewrite test it was less than 9%. The read test is the most unstable — the difference was as big as 50%.

The Spad filesystem has, because of its simplicity, the lowest CPU consumption. It has 6 times lower consumption than Ext2 for writing and 3.7 times lower consumption than Ext2 for rewriting. For the reading benchmark, SpadFS shows 1.14 – 2.3 times lower consumption than the next filesystems, Ext4 and JFS.

10.2.6 Directory access counting

To show efficiency when accessing large directories, I used blktrace to count the number of IO requests. A directory was created with 100 to 10 000 000 files. The filesystem was unmounted and remounted (so that there were no cached data left) and an operation was performed: open a file, create a file and sync, or delete a file and sync. blktrace was used to measure the number of disk accesses for this operation.

The number before the slash denotes read requests; the number after the slash denotes write requests.

The average number of disk accesses is reported in the table. The difference in successive runs was no more than one request in most cases. In a few isolated cases it was up to 3 requests.

SpadFS clearly wins for both read and write requests.

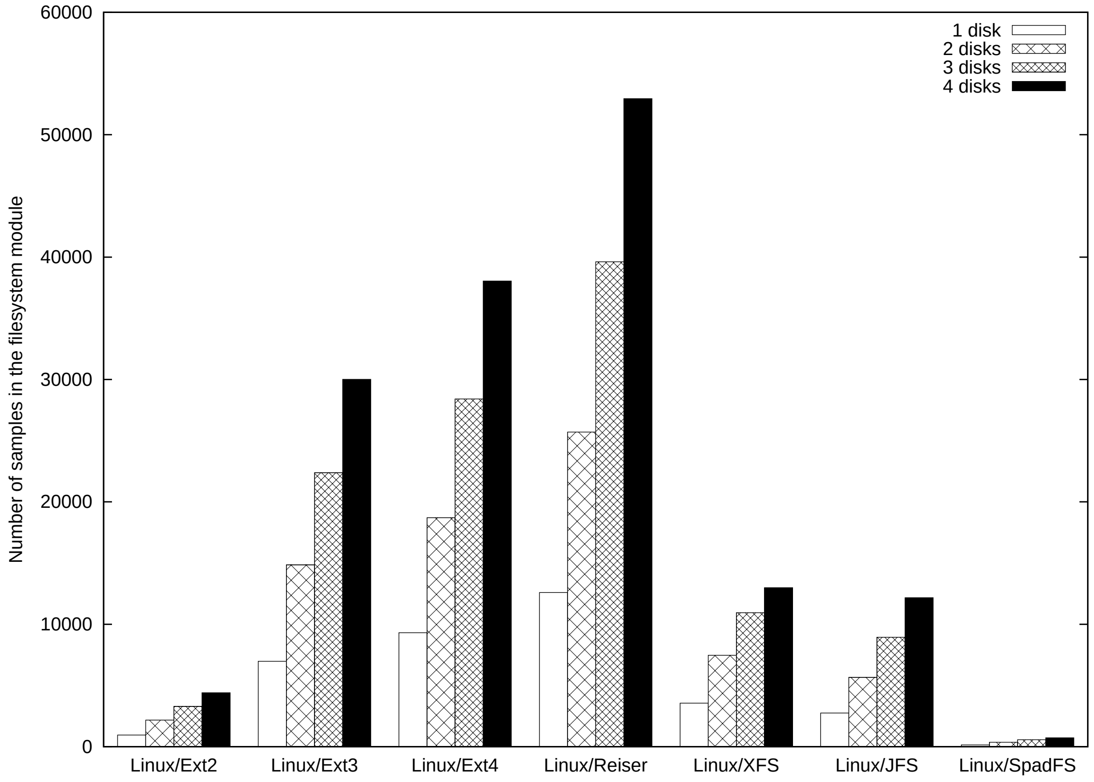

Figure 10.15: CPU consumption when writing files

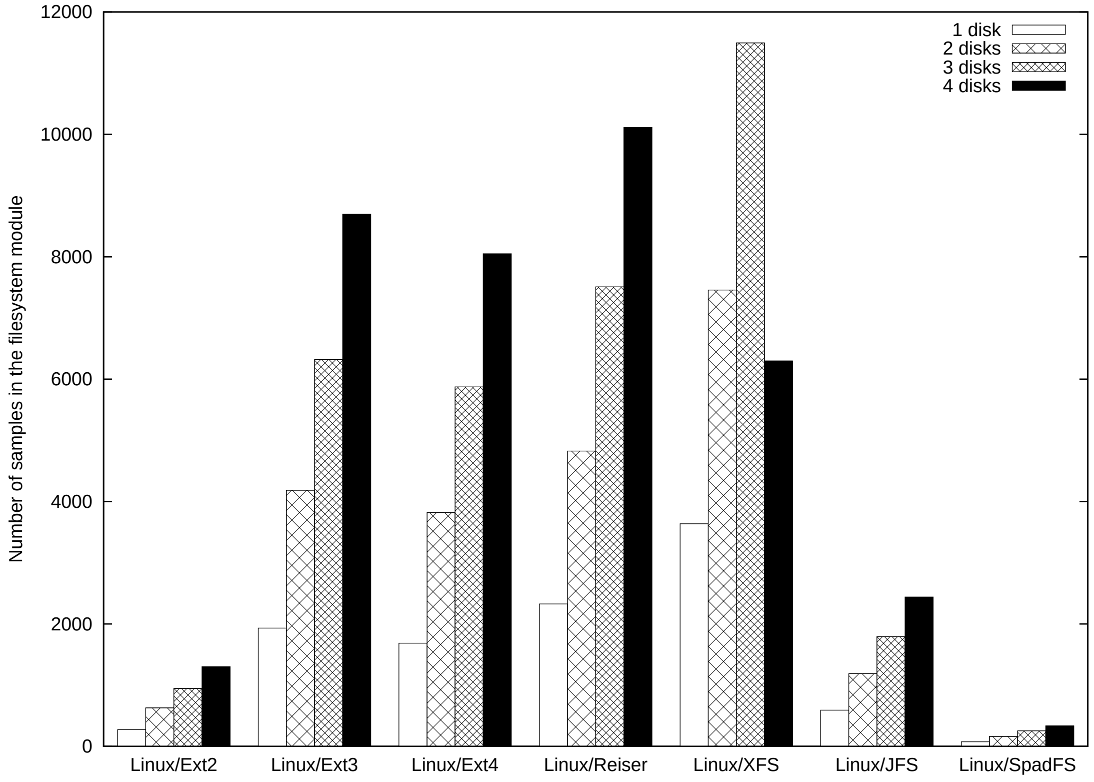

Figure 10.16: CPU consumption when rewriting files

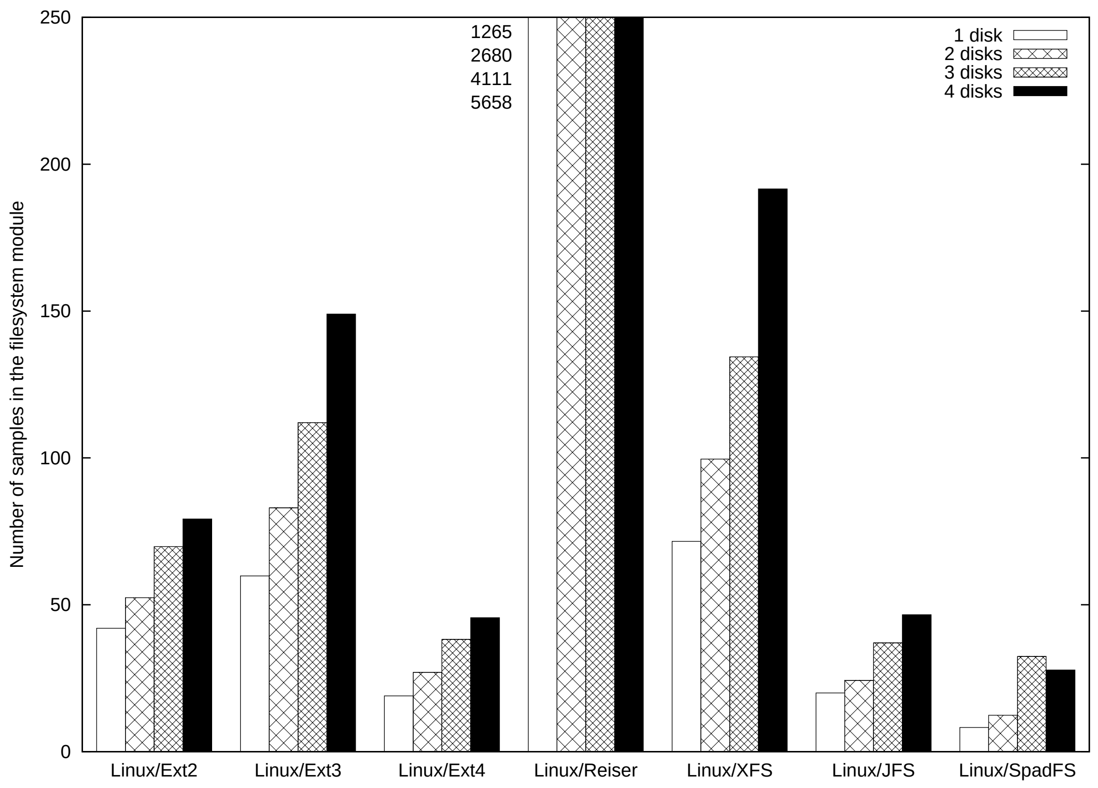

Figure 10.17: CPU consumption when reading files

	Ext3	Ext4	ReiserFS	XFS	JFS	SpadFS
100	3.6	3.6	1	1.6	4	1
1000	5	5	1.6	4	4.8	1.8
10000	5.8	5	2	5	5.8	2
100000	7	6	4	8	7	2
1000000	9	6	4	16.4	7	3
10000000	9	7	5.8	113.2	9	3

Table 10.1: Number of disk accesses to open a file in the directory

read / write	Ext3	Ext4	ReiserFS	XFS	JFS	SpadFS
100	5 / 7	5 / 7	0.8 / 8.2	3 / 5	6 / 7.2	1 / 2
1000	6 / 7	6 / 7	2 / 10.2	6 / 5	8 / 8.8	1.8 / 2
10000	6.8 / 7	6 / 7	2.2 / 12	7 / 5	11 / 10	2 / 2
100000	8 / 8	7 / 8	2.8 / 12.4	17 / 5	12.2 / 10.6	2 / 2
1000000	9.6 / 8	7 / 8	3.4 / 13.8	82 / 5	14.4 / 10.2	3 / 2
10000000	10 / 8	8 / 8	5.6 / 18.6	145 / 5	17.4 / 10.4	3 / 2

Table 10.2: Number of read/write accesses to create a file in the directory

read / write	Ext3	Ext4	ReiserFS	XFS	JFS	SpadFS
100	4.8 / 7.2	4.6 / 6.6	1.4 / 8	2.6 / 4.6	5 / 4.4	1 / 2
1000	6 / 7	6 / 7	1.4 / 9.8	5.4 / 5	6.6 / 6.2	2 / 2
10000	6.8 / 7	6 / 7	2 / 12	6.8 / 5	8.4 / 7.6	2 / 2
100000	8 / 8	7 / 8	3.8 / 12.8	11 / 5	11.2 / 9.6	2 / 2
1000000	9.6 / 7.8	7 / 8.6	4 / 12.8	19 / 5	13 / 9.4	3 / 2
10000000	10 / 8.4	8 / 8.4	5.8 / 13	117 / 5	16.6 / 9.8	3 / 2

Table 10.3: Number of read/write accesses to delete a file in the directory

10.2.7 Postmark

Postmark [67] was originally developed for simulating the load of mail servers. Postmark is a single-threaded benchmark; it creates a set of files and performs transactions on them, each transaction consisting of two operations:

- read or append (chosen at random) of a random file
- create or delete a random file

All the default parameters were used, except that the number of transactions was increased to 1000000 (so that the results were more accurate) and buffered libc IO was disabled (so that postmark performs direct kernel syscalls — because I need to measure kernel performance and not libc performance).

When running postmark, the fileset fit into a 1GiB system memory, thus this benchmark shows the performance of cached operations.

The results are in figure 10.18.

The difference from the average value was no more than one second, except for JFS, where it was 4 seconds and for XFS 20 seconds.

The Spad kernel wins the benchmark as it is twice faster than the SpadFS implementation on the Linux kernel. Spad shows the same performance for both its filesystems, Ext2 and SpadFS — the reason is that Spad has the write cache above the filesystem, so the benchmark hit the cache and didn't get to the filesystem driver at all. XFS greatly loses on this benchmark; the reason is that it can't batch a high amount of transactions into one disk I/O. So it performs at a rate of only 313 transactions per second.

This is a metadata-intensive in-memory benchmark, so I ran one benchmark on SpadFS on the Linux kernel with metadata checksums enabled to see the effect of metadata checksumming on performance. The result without checksums was 38.8 seconds, with checksums 41.2 seconds. The Linux implementation always checks the checksum when accessing the buffer; the Spad kernel marks the matched checksum with a bit in the buffer structure (so that the checksum isn't rechecked if the buffer is read multiple times from memory), so there is no difference on the Spad kernel when checksums are enabled or disabled on cached operations.

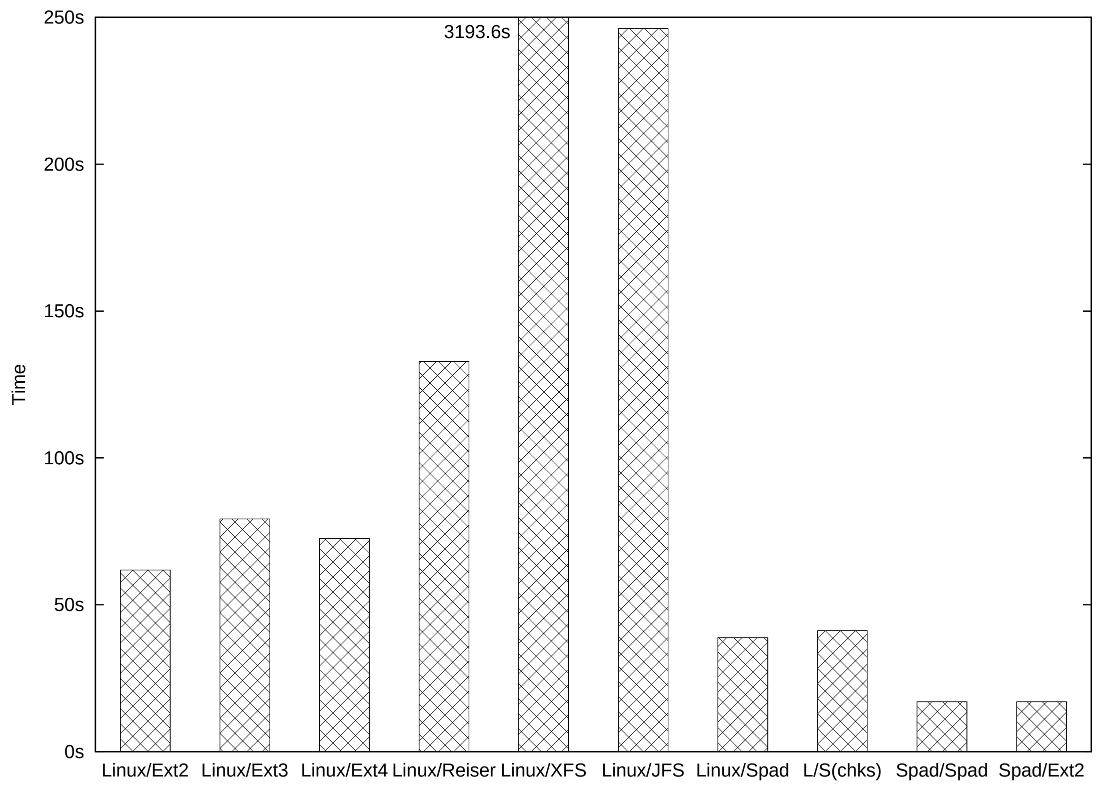

Figure 10.18: The duration of the postmark benchmark for 1000000 transactions

10.2.8 FFSB

FFSB stands for "Flexible filesystem benchmark". It is a configurable benchmark that can perform creating, deleting, reading and writing files in multiple threads. The benchmark runs for a fixed amount of time (300s) and the result is the number of transactions performed in this time. For the benchmark, I selected standard `profile_mixed_workload` script in `examples` directory. This script works by creating a total of 100 files randomly scattered in 100 directories and performing transactions on the files, each transaction is either 4kiB read (with 4/5 probability) or 4kiB append (with 1/5 probability). The generated fileset didn't fit into the memory, so the benchmark shows an uncached operation. The results are in figure 10.19.

The difference from the average value was 2.5%, except for SpadFS/Spad where it was 10%.

FFSB can also measure the percentage of CPU usage. I recalculated these numbers to seconds of CPU usage per transaction (so that the filesystem that performs a low number of transactions wouldn't show the unfair advantage of having low CPU usage) and displayed them in figure 10.20. The percentage of CPU usage wasn't measured on the Spad kernel, because the Spad kernel doesn't have syscalls for measuring it.

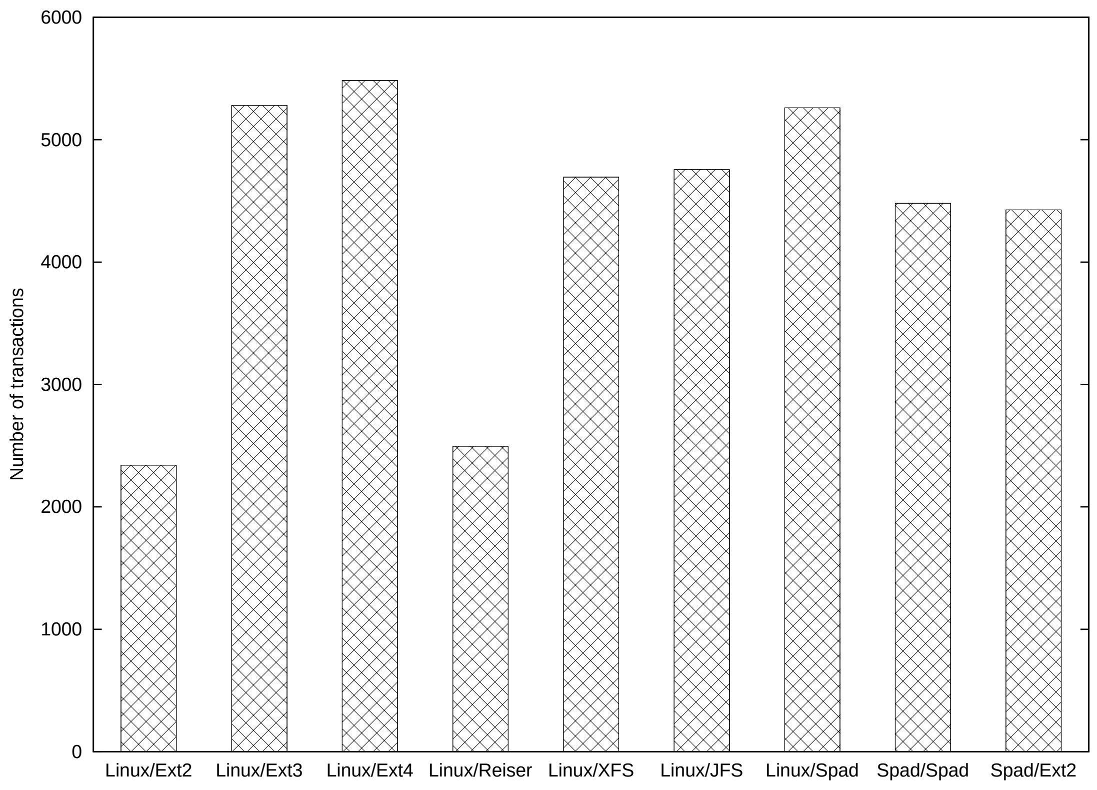

Figure 10.19: The number of transactions in 5 minutes for the FFSB benchmark

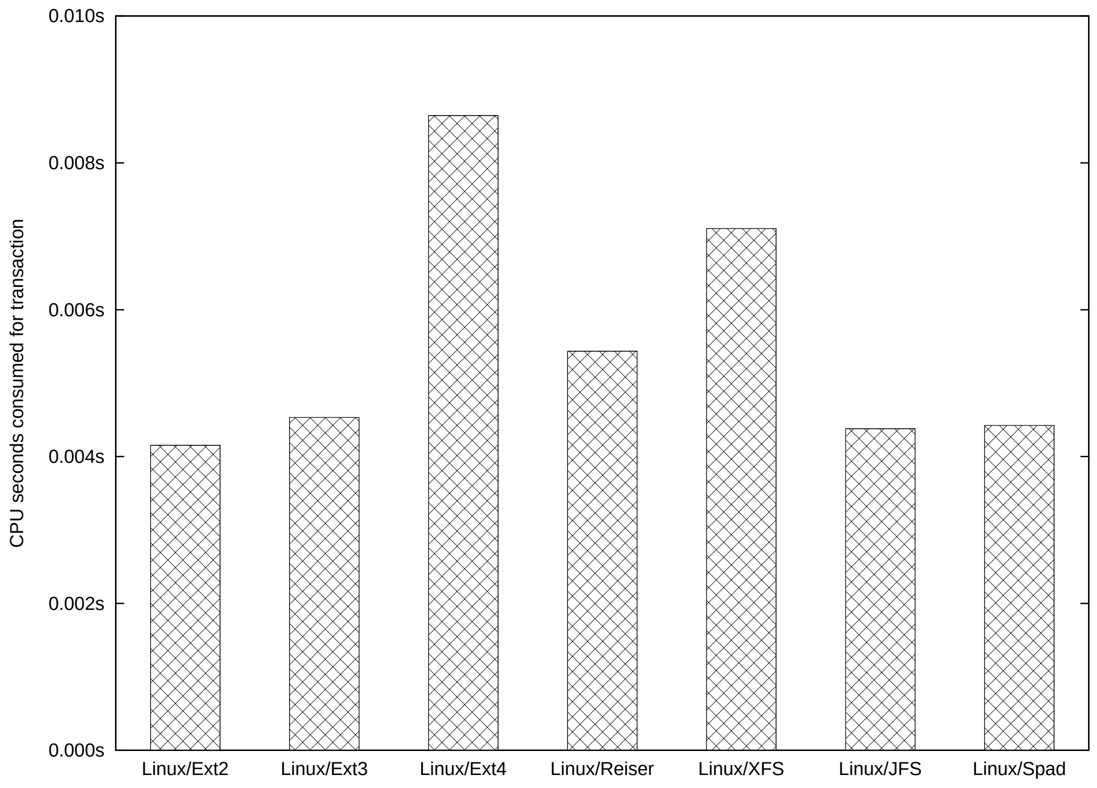

Figure 10.20: CPU seconds consumed per transaction for the FFSB benchmark

10.2.9 Postmark and FFSB on different computers

I performed postmark and FFSB benchmarks on some different computers:

- AMD K6-3 550MHz, 256kB L2 cache, 512MiB RAM, 36GB 10k RPM Compaq / Seagate SCSI disk. The test partition was in the first half of the disk.
- UltraSparc IIi 440MHz, 2MiB L2 cache, 1GiB RAM, 18GB 10k RPM Seagate SCSI disk. The test partition was in the first half of the disk.

The configuration of the benchmarks was the same as on the Pentium 4. The same Linux kernel version 2.6.26.3 was used on these computers. The Spad kernel was tested only on K6-3 because it doesn't run on Sparc64 architecture.

The results are shown in figures 10.21, 10.22, 10.23, 10.24.

The individual runs had no more than 10% variation from the average.

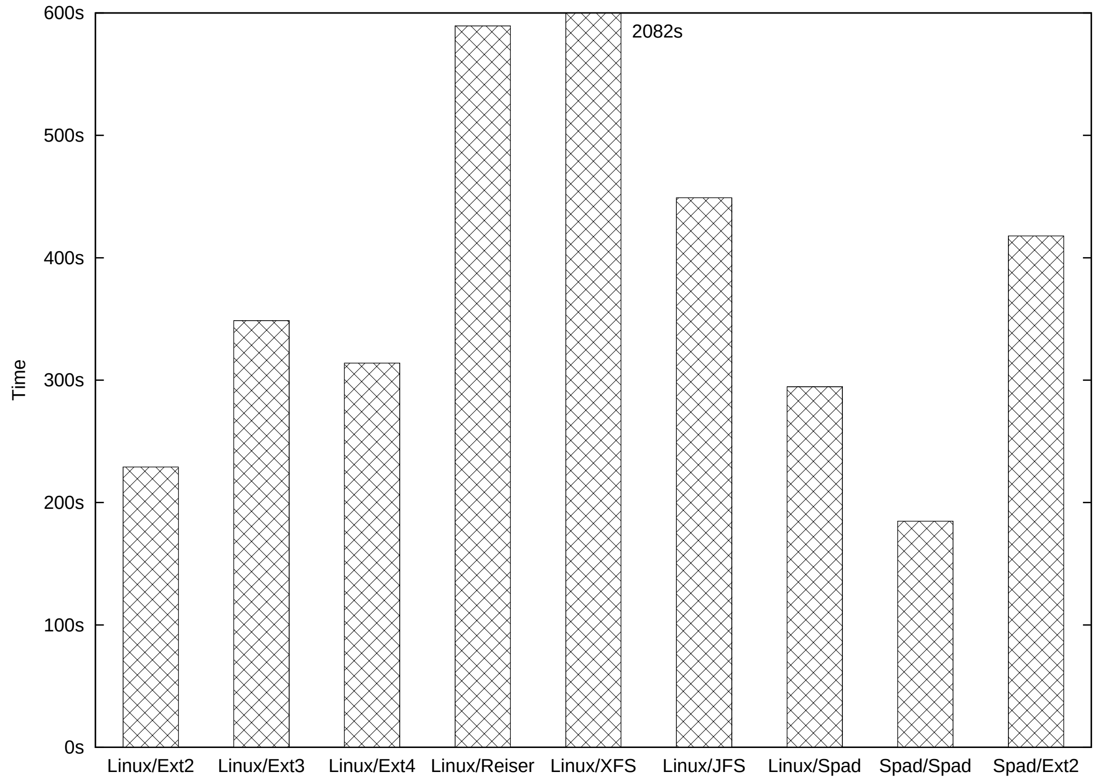

Figure 10.21: The duration of the postmark benchmark on AMD K6-3

We can see that the scaling of performance on different computers is not perfectly linear.

For example, in the postmark benchmark, we can see that SpadFS/Linux is 37% faster than Ext2/Linux on Pentium 4, SpadFS/Linux is 28% slower than Ext2/Linux on K6-3 and SpadFS/Linux is 20% faster than Ext2/Linux on UltraSparc.

Another great nonlinearity can be seen in the postmark between JFS and ReiserFS — ReiserFS is almost twice faster than JFS on Pentium 4, but JFS is twice faster than ReiserFS on UltraSparc; on K6-3 JFS is 24% faster than ReiserFS.

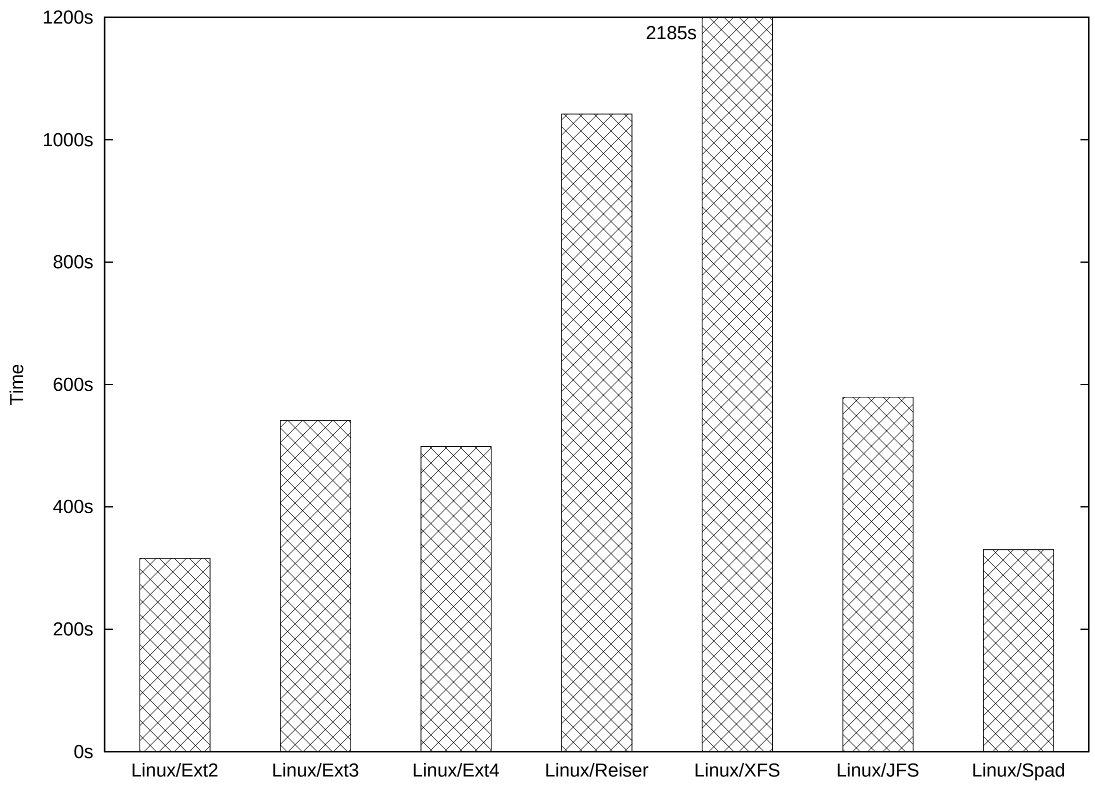

Figure 10.22: The duration of the postmark benchmark on UltraSparcIIi

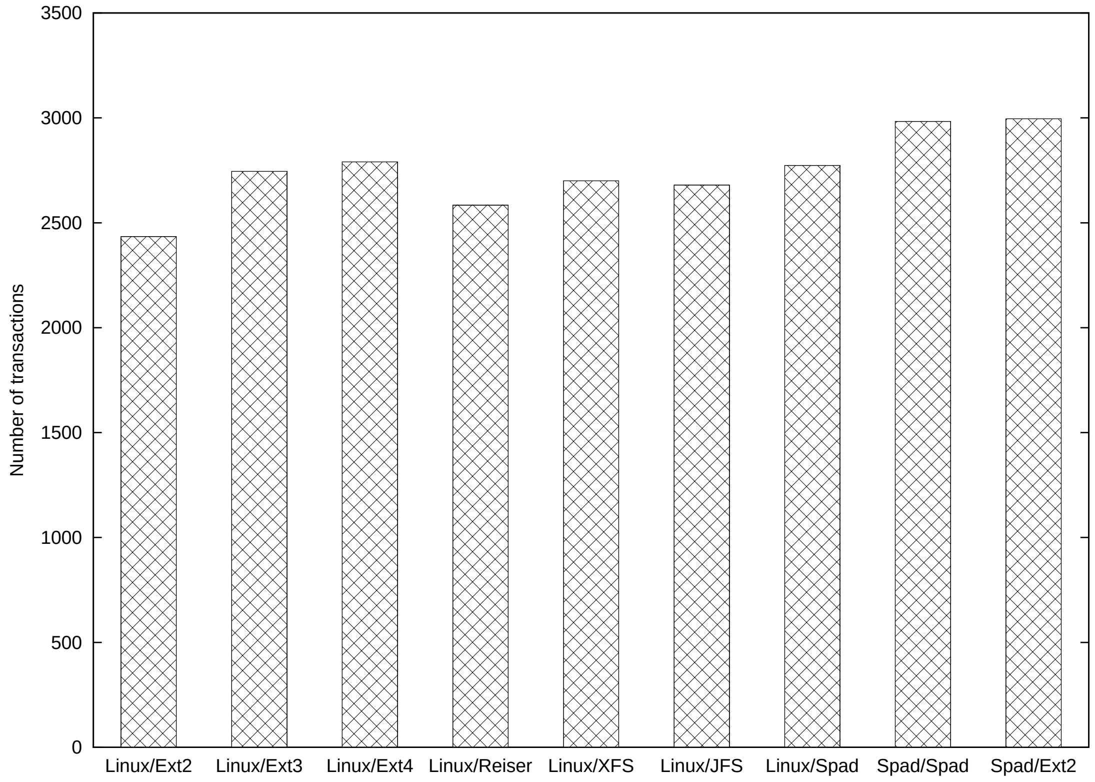

Figure 10.23: The number of transactions for the FFSB benchmark on AMD-K6-3

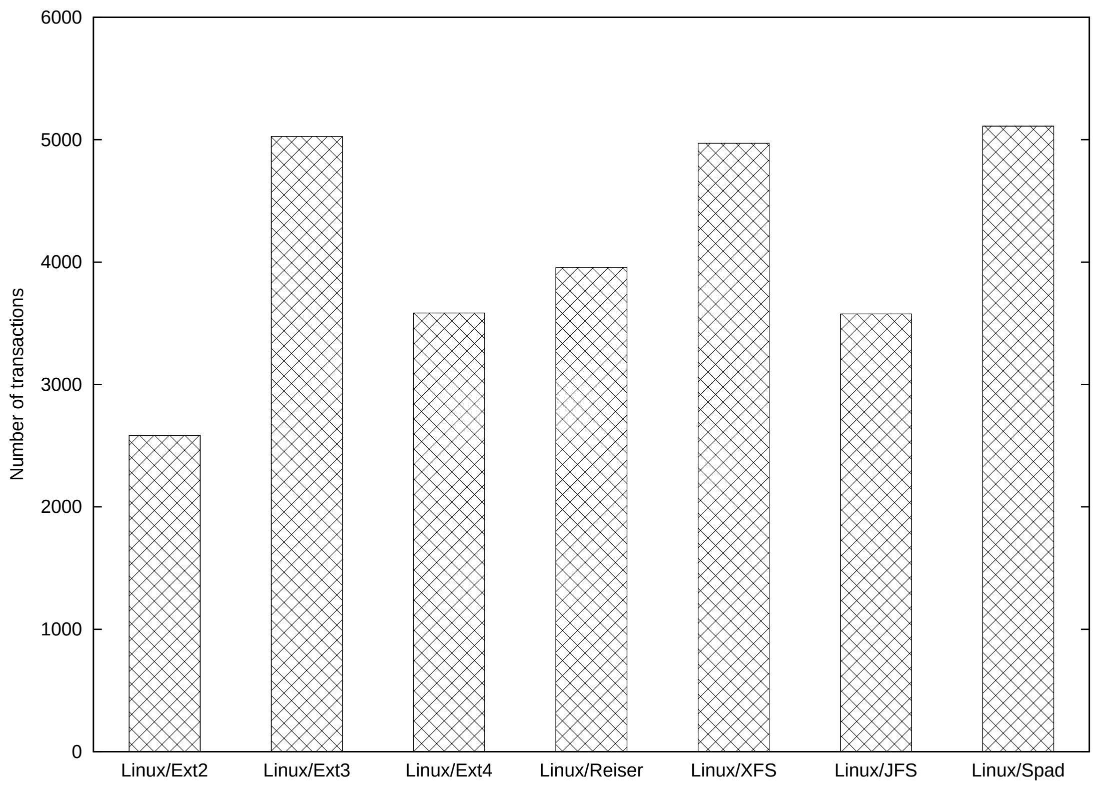

Figure 10.24: The number of transactions for the FFSB benchmark on UltraSparcIIi

In FFSB benchmarks nonlinearities happen, too. On Pentium 4, the Spad kernel is 22% slower than the fastest Linux filesystem, Ext4. But the Spad kernel is 6% faster than Linux Ext4 on K6-3.

Note that postmark is mostly cached, stressing the CPU and memory system. FFSB is uncached, thus stressing mostly the disk.

The performance of parts of the computer (CPU, L1 cache, L2 cache, memory, disk) does not scale linearly between computers. Because different software stresses these parts differently, performance of the software does not scale linearly either.

Therefore, any strong conclusion in the form "Filesystem A is X% faster than filesystem B" cannot be made because the number X depends on the computer being used.

10.2.10 FFSB on a flash memory

I ran the ffsb benchmark on a flash memory. The memory is Transcend TS64GSSD25-M connected to an AMD K6-3 computer with 512MB RAM. The tests were run on an 8GB partition located near the beginning of the memory. The results can be seen in figure 10.25. The difference from the average value was no more than 3.5%, except for JFS where it was 10%.

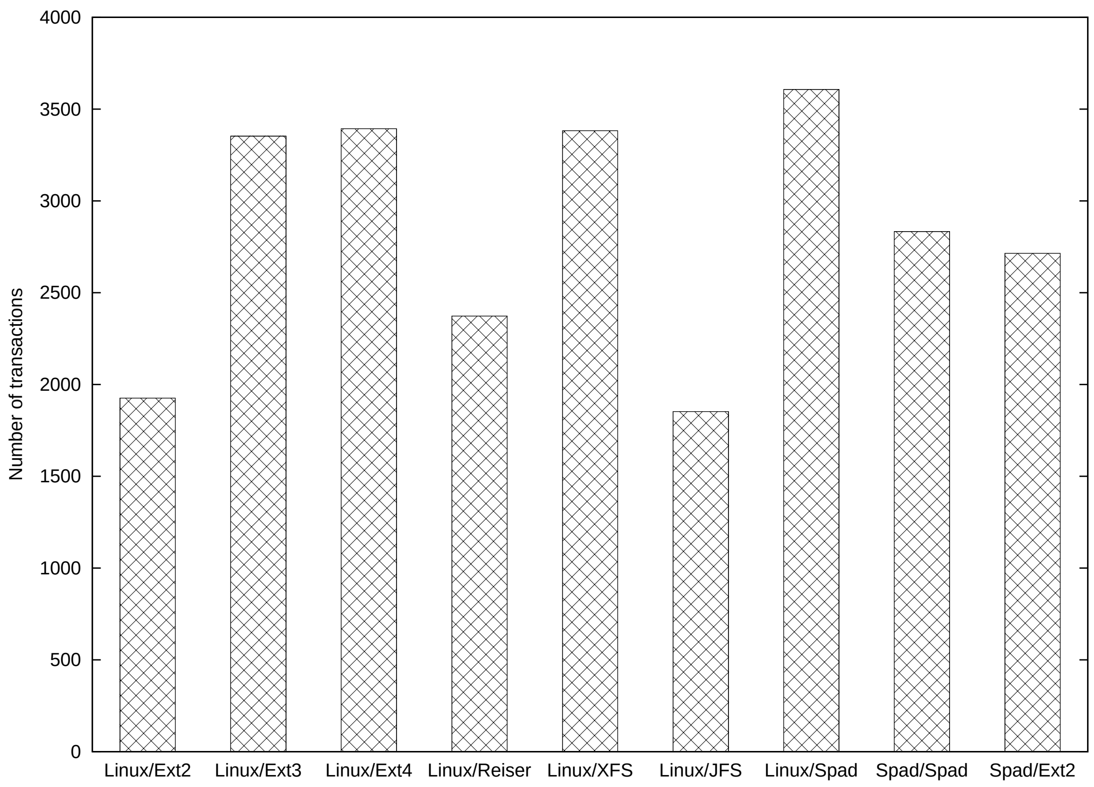

Figure 10.25: The The number of transactions for the FFSB benchmark on the flash memory

10.2.11 Performance depending on block size

In figure 10.26 we can see the duration of the postmark benchmark on the SpadFS filesystem, depending on the block size. The test was done on UltraSparc IIi.

10.2.12 The time to execute fsck

In figure 10.27 we can see the time to execute fsck for various filesystems. The test was done on UltraSparc IIi on 36GB 10k RPM Seagate SCSI disk. The 4GB test partition was at the end of the filesystem. The test partition was filled with four copies of the Linux kernel 2.6.38 source.

JFS was not tested because its fsck program requires unaligned accesses and thus doesn't work on Sparc architecture.

10.2.13 Operations with directory tree

Next, some real-workload benchmarks were performed. I downloaded sources of the OpenSolaris operating system `on-src-20060814.tar.bz2` from `www.opensolaris.org`. This file was selected because it is one of the largest software archives on the Internet. I

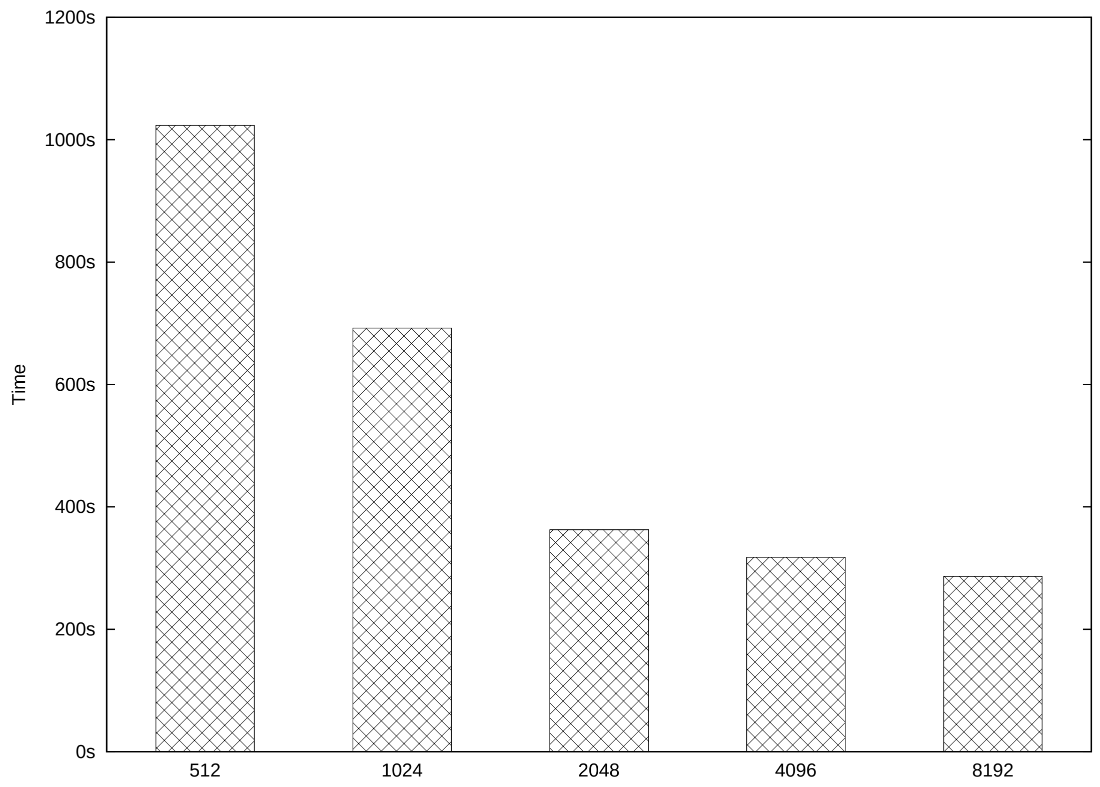

Figure 10.26: The duration of the postmark benchmark depending on the block size

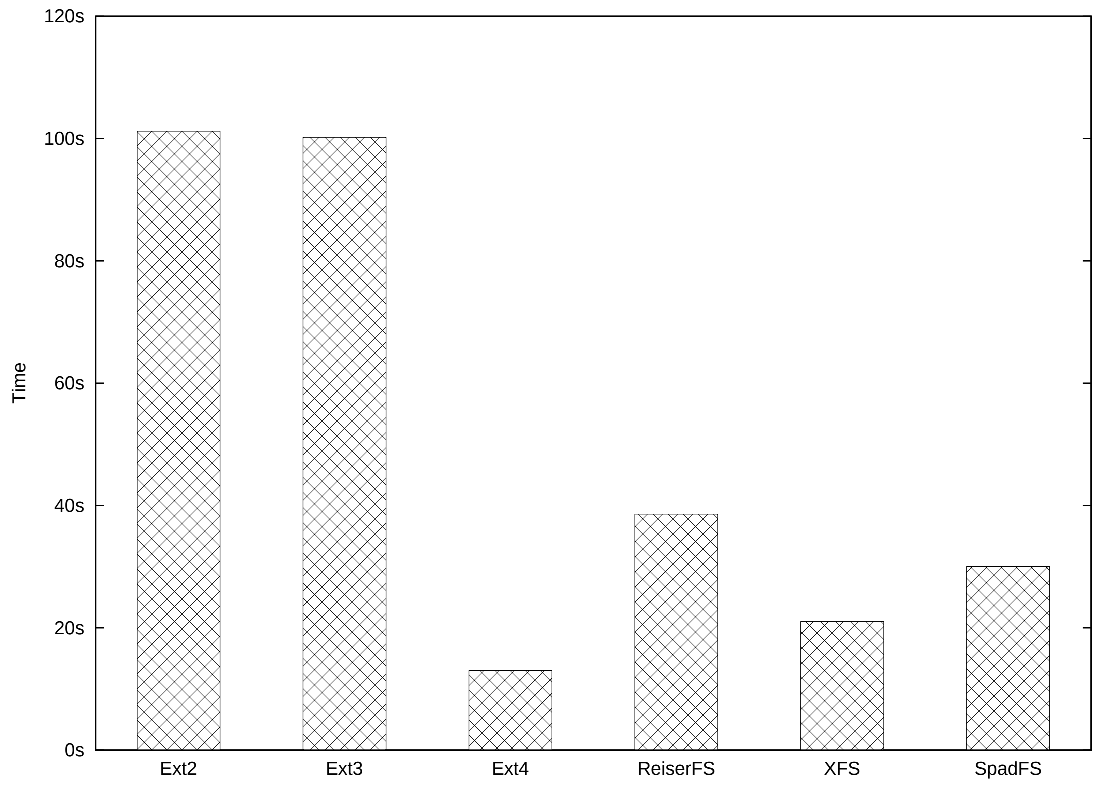

Figure 10.27: The duration of fsck for various filesystems

decompressed it, creating a 365MB `.tar` file containing 6275 directories and 34411 files and performed the following operations with it:

- I unpacked the file with `tar xf` command
- I copied the unpacked directory tree with `cp -a` command
- I read the whole directory tree with `grep -r` command
- I scanned the whole directory tree with `find` command
- I compared both copies with `diff -r` command
- finally I deleted the directory tree with `rm -rf` command

Cache was invalidated between all these tests.

The time of these operations is shown in figures 10.28 and for diff operation in 10.29. Columns filled with pattern show the time until the command completed; the empty columns above show the time until the subsequent cache flushing completed.

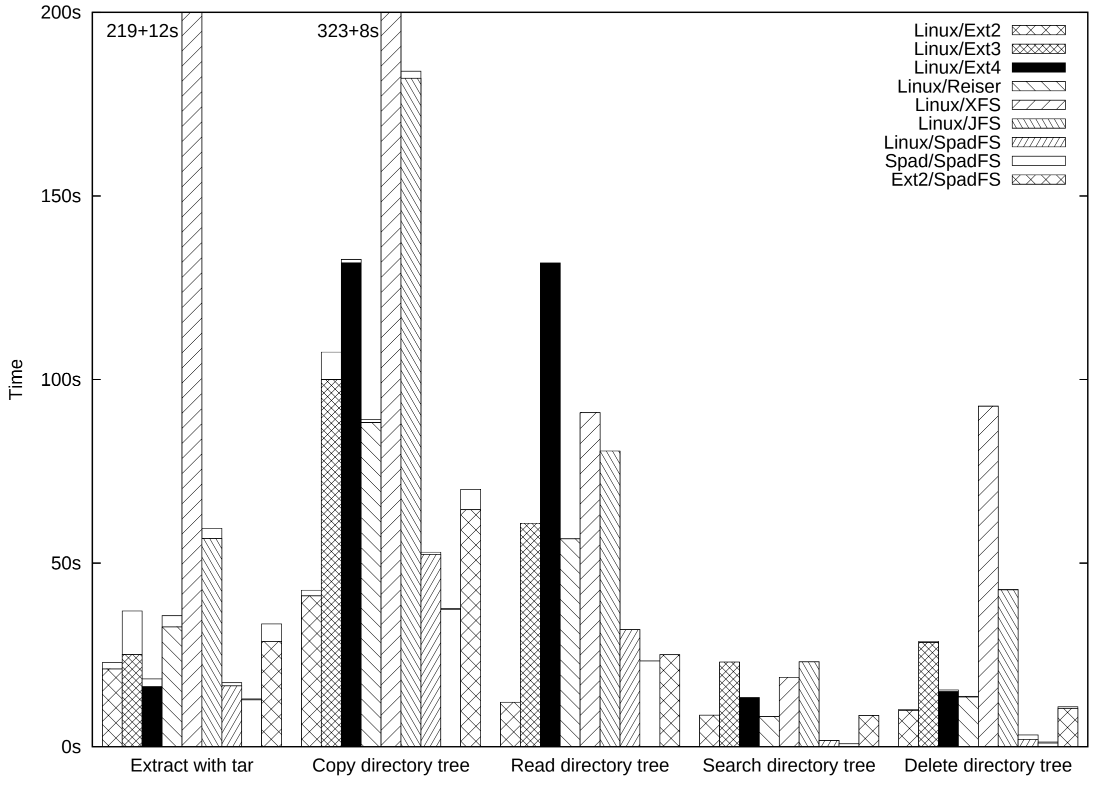

Figure 10.28: Time to do operations with directory tree.
The upper unfilled bar represents the time needed to flush cache.

Ext2, despite being the oldest filesystem, performs exceptionally well, winning the directory read benchmark and being second in the directory copy benchmark. Ext2 simplicity does not damage performance in this benchmark — this benchmark consists of many small files (so extents do not cause an advantage) and directories with few entries (so the directory index will likely hurt the performance). The simplicity of Ext2 makes it consume a small amount of CPU time and win.

SpadFS on Spad wins extract, copy, diff and delete benchmarks. Extract is won by 25% over Ext4; copy is won by 14% over Ext2.

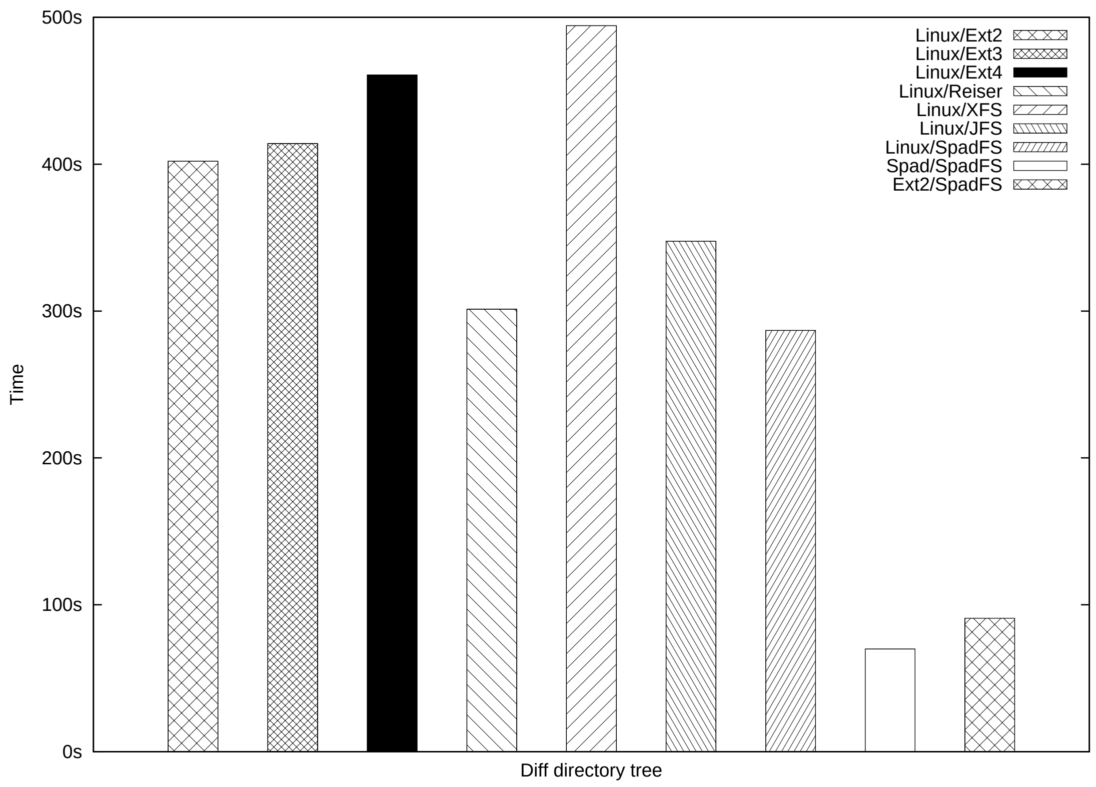

Figure 10.29: Time run diff command on a directory tree.

SpadFS on both kernels (Spad and Linux) greatly wins the find and delete benchmark because SpadFS stores file information directly in the directory, so there is no need for an additional seek to an inode. Additionally, SpadFS stores metadata in an extra zone, allowing easy prefetch of metadata. SpadFS on Spad is 8 times faster than Ext2 on Linux. SpadFS on Linux is 3 times faster.

The Spad kernel with both its filesystems (SpadFS and Ext2) greatly wins the diff benchmark — it is caused by the cross-file readahead feature. The diff test makes the disk head seek vigorously from one tree to the other; filesystems without cross-file readahead would read one file from one tree (having a few kB size) and seek to the other tree while filesystems with cross-file readahead can read more files in advance.

10.3 Testing correctness of the filesystem

In this section I describe the testing of the filesystem that was done. The purpose of testing is not to give some numbers that can be compared but to check the implementation for correctness.

To test the correctness of the kernel driver, I used two programs: FSX [101] and FSSTRESS.

FSX was developed by Avadis Tevanian. It is a single-threaded program working on a single file. It performs reads and writes (via `read()/write()` syscalls or via mapped memory) and truncates and extends. FSX stores the image of the file in the memory and checks that the data read from the file corresponds to the data that should be there.

FSSTRESS is a program developed by SGI for testing XFS. FSSTRESS is a multi-threaded program; it spawns a configurable number of threads each performing tests in its own directory. FSSTRESS tests all directory filesystem operations, including manipulation of directories or creation of device nodes.

FSX and FSSTRESS tests were run on both the Spad and Linux kernel.

FSSTRESS together with LVM2 snapshots were used to test correctness of the crash count algorithm. For this test, FSSTRESS was run and simultaneously, another program was repeatedly creating a snapshot, checking the snapshot with spadfsck and deleting the snapshot. This test showed that the filesystem is consistent at each point in time and that it can maintain consistency across system crashes.

Spadfsck program was tested with FSFUZZ program. FSFUZZ takes a file and changes random bytes in a predefined range. For this testing, the filesystem image with a size of 768MiB was created and two Linux kernel source trees (partially hardlinked) were extracted on it. FSFUZZ was instructed to change bytes in a range from 8MiB to 16MiB (where the metadata zone resided). To check memory correctness of Spadfsck, the calls `malloc` and `free` were wrapped in such a way that a control block with a magic number and a file and line of the allocation was added before each allocated chunk and "redzone" byte was added after the allocated chunk. The `free` function wrapper checked the magic and the redzone. Finally, at the end of spadfsck, the memory was checked for memory leaks. Spadfsck was run on the resulting file, first with the argument "`-fy`" to fix all data and the second time with the argument "`-fn`" to check that the filesystem is really correct. The script was run repeatedly and checked if any of the runs ended abnormally (for example with a segmentation fault) or for the error condition: if the first run reported that it fixed all the errors and the second run found out that there were still errors on the filesystem.

FSFUZZ was also used to test error handling of the kernel driver. FSFUZZ created a filesystem, changed random bytes and tried to mount and access the modified filesystem. This test can uncover bugs in error handling in the kernel driver.

10.4 Code size

In this section I present the code size of the filesystem drivers for the Linux and Spad kernel.

The code size may be related to complexity (i.e. how hard it was to write and maintain the code) but some people may argue that the relation is meaningless (for example, writing more comments into the code will make it more maintainable although it increases code size). I will present the code size anyway.

The code size is calculated from all files with extensions `"c"` or `"h"` in the filesystem directory and its subdirectories (except `"utils"` subdirectory for SpadFS because it contains userspace tools and no kernel code). Some filesystems on Linux have some include files in `include/linux`; if this is the case, the include files were added to the filesystem size. Some filesystems are composed of multiple modules: for Ext3, the sum of sizes of modules `ext3` and `jbd` was taken; for Ext4, the sum of `ext4` and `jbd2` and for FAT, the sum of `fat`, `msdos` and `vfat`. The CD filesystem is called `ISO9660` on Linux and `CDFS` on Spad, but it is really the same filesystem.

Binary size is the size of the module file compiled for x86 with optimization and without debugging.

OS	Filesystem	Lines of code	Bytes of code	Binary size
Linux	Ext2	9523	266059	81522
	Ext3	25347	724179	205606
	Ext4	36553	1040093	305412
	ReiserFS	30794	936170	254692
	XFS	105178	2990989	611366
	JFS	32992	845506	205237
	FAT	6623	170111	90159
	HPFS	6036	170749	86962
	ISO9660	3992	102789	36913
	SpadFS	11671	316850	109421
Spad	SpadFS	5804	205204	75568
	Ext2	3211	103374	27948
	FAT	2123	64755	26428
	HPFS	3789	128936	40312
	CDFS	1398	44572	15900

Table 10.4: The sizes of various filesystem drivers

10.5 Notes about benchmarking

This section contains some interesting facts that I discovered during development and benchmarking.

For benchmarking, it was crucial to properly flush the cache. I found that making a subroutine that allocates all available memory, writing and reading it twice, and forcing the operating system to swap, is a reliable way of flushing the filesystem cache. Note that reading many unrelated files does not always flush the cache because virtual memory algorithms are protected against sequential reads.

When implementing continuous allocation of directories and cross-file readahead, I found a rather important requirement: sort the files alphabetically before allocating them. My `readdir` implementation returns files in sorted order and if they were allocated on disk in a different order, cross-file readahead actually hurts. The file `ON-SRC-20060814.TAR` contains unsorted filenames, which made this problem initially appear.

During the development of cross-file readahead, I discovered a small but important fact: first start the readahead, then copy the data from the cache to userspace — so that reading from the disk and copying the data overlaps. It was noticeably slower when these two actions were swapped.

Ext2 and Ext3 have the same on-disk format but Ext3 has a significantly slower reading of the directory tree with `grep`. The reason is the directory index. Ext3 without the directory index actually performed like Ext2 in `grep` test. Without the directory index, the directory entries are stored in the directory in the order in which they were added. With the index, this order of directory entries is violated, but inodes are still allocated in the order in which they were created — as a result, accessing the directory entries sequentially will access inodes in a non-sequential way[1]. While many users think that the directory index improves performance, it might not always be true.

XFS performance can be improved if write barriers are turned off (but turning barriers off causes risk of data damage if the hardware write cache is enabled and power failure happens). Because barriers are used on other filesystems, too, default settings (enabled barriers) for XFS were retained.

Spad VFS flushes the cache very aggressively on writes and discards cached data on sequential reads (it can be seen in figure 10.28 — it has nearly zero times to perform sync). The benchmarks could be improved even more by disabling this functionality and aggregating more data in the cache; however, such a setup degraded interactive performance — after extracting tar file, the user likely wants his previously used programs and their files to remain in the cache rather than using the cache for an incomplete part of the extracted tree.

10.6 Benchmark summary

I benchmarked various Linux filesystems and my novel Spad filesystem and Spad kernel.

It was shown that SpadFS performs comparable to other filesystems and outperforms them in some occasions.

It was shown that for certain workloads the design of the VFS layer matters more than the physical layout of data on the disk.

It can be seen that the novel feature ”cross-file readahead” can guarantee decent performance when accessing more directory trees simultaneously.

10.7 Limited possibility to make conclusions from experiments

It should be noted that any strong conclusion about the used algorithms cannot be drawn from the benchmarks. The reason is that performance of the filesystem is determined by many design decisions and many other factors, some of them unknown and unmeasurable (for example the skill of the programmer, his effort and motivation, time pressure put on him...). In natural sciences, effects of such random variables can

[1] Note that SpadFS also reorders directory entries but doesn’t suffer from this problem because the file information is stored directly in the directory entry.

be mitigated by performing the experiment on a large group of experimental objects and control objects — for example, if a doctor wants to prove that a drug is effective to treat a disease, he must perform the test on several tens or hundreds of patients; part of them receives the drug and the other part doesn't — so that difference in individual patients doesn't affect the result. If the experiment were performed only on one patient getting the drug and one not getting the drug, the experiment is flawed — it cannot be determined if the results are caused by the drug or by natural variation in the strength of the human immune system.

In computer science it is impossible to make a properly controlled experiment — it is impossible to pay several tens of programmers to implement a filesystem with the same algorithm or design decision (so that variance in the skill of individual programmers or the effects of minor decisions unrelated to the measured algorithm would be canceled in the result), therefore it is impossible to make any certain conclusion about the efficiency of algorithms or design decisions.

An example of these effects can be seen in table 10.1. Ext4, JFS and XFS all have exactly the same algorithms for directory organization (per-directory B-trees) and maintaining consistency (journaling), but the result of these filesystems differs significantly — to open a file in the directory with 10 000 000 entries, XFS does 16 times more accesses than Ext4. The difference is not caused by the algorithms, but by the implementation. The algorithm only guarantees $O(\log n)$ complexity — the absolute number is a matter of the designers' and programmers' skill, not a matter of the algorithm.

Other smaller differences between filesystems with the same algorithms can be seen in other charts: In section 10.2.4 (figures 10.11, 10.12, 10.13) Ext4, JFS and XFS have the same file-mapping algorithm when rewriting or reading files, but performance differs: XFS consumes 1.5 times more CPU time than JFS when rewriting. In section 10.2.5, in figure 10.16 we can see that Ext4 takes three times more CPU time than JFS, in figure 10.17 XFS takes four times more CPU time than JFS or Ext4, although the same algorithm is used. In section 10.2.13, in figure 10.28, the same algorithms are used in Ext4, JFS and XFS when reading or searching the directory tree, but the results differ by approximately 1.5.

It is possible to make conclusions that are bound to specific conditions of the experiment. For example, the claim "Mikuláš Patočka's Linux implementation of SpadFS performs 37% better than Ext2 in the postmark on Pentium 4" is true and can be experimentally verified. Some statements of this kind are presented in previous sections.

But such a claim doesn't make any contribution to human society. Because such a claim isn't general. Other researchers or software engineers who develop other filesystems are going to get different results ... even if they use the same algorithms and decisions as I did.

It should also be noted that the experimental results (even if they show high performance) do not imply that the filesystem is suitable for use by users. Unfortunately, the scientific experimental method cannot be used to satisfy users' requirements.

The reality is that users have multiple requirements for software (such as a filesystem). The main user requirements are:

- The software has good performance
- The software is reliable
- The software is delivered on time
- The software has features and functionality

From a user's point of view, the user wants all these requirements. But from a developer's point of view, the requirements are in conflict with each other. In other words, if the developer improves any of the user's requirements, he inevitably damages at least one other requirement:

- If the developer wants to improve performance, he can use less safe programming methods (decrease reliability) or he can allocate more time for optimizations and performance evaluation (increase delivery time) or he can allocate more time for optimizations and performance evaluation and reduce the time spent on implementing functionality (decrease functionality).
- If the developer wants to increase reliability, it can be done either by using safer programming methods or languages (decrease performance) or by allocating more time for testing (increase delivery time) or by reducing code size (decreasing functionality).
- Delivery time can be improved by reducing time spent on optimizations (decrease performance), time spent on testing (decrease reliability) or time spent on implementing functionality (decrease functionality).
- Functionality improvements will take some time; this time will either cause delay in delivery (decrease delivery time) or it can be done at a cost of reducing time spent on optimizing (decrease performance) or testing (decrease reliability).

If the developer wants to satisfy the user, he must find a balance between the user's requirements. Any of these four requirements outlined above can be scientifically measured and compared. But their balance can't be evaluated and compared. There is no known function that would assign weight to each requirement and calculate a comparable value so that the developer could evaluate the suitability of his software. For example, if I am going to ask "does the user rather want 5% performance increase in a specific benchmark or does he rather want bootloader support allowing him to install the operating system on the filesystem?", there is no known scientific model, no known experimental method, that could answer this question.

In practical software development this balance between users' requirements is being found intuitively by software designers. But scientists do not allow using intuition in design and they do reasoning based solely on verifiable measurable results — so they can hardly find the right balance and produce usable software.

11. Conclusion

In this thesis I described the process of designing the Spad filesystem. I showed the algorithms used in existing filesystems and I showed how they affected my decision making.

The Spad filesystem performs comparably to other filesystems, outperforms them at a few benchmarks (for example at postmark or some operations with the directory tree), and the Linux implementation has 3.7 times less lines of code than the smallest Linux filesystem providing data consistency, Ext3.

The Spad filesystem has the lowest amount of time spent in the filesystem driver of all measured filesystems, providing the best CPU-time scalability for fast devices. It has the lowest amount of accesses for large directories of all measured filesystems, only 3 accesses for a directory with 10 000 000 entries.

11.1 Summary of my design

I selected methods that provide good performance and that are relatively easy to implement.

The Spad filesystem uses a novel method for maintaining data consistency, crash counts. The Spad filesystem embeds file information directly in the directory, saving one seek when accessing files. For indexing directories, SpadFS uses extendible hashing providing a low number of accesses in large directories. For file allocation SpadFS uses direct/indirect blocks containing extents — they are simpler to implement than B-trees and they provide comparable performance. For free space management SpadFS uses double-linked lists of extents — they are simple to manipulate and provide average $O(\sqrt{n})$ complexity.

During my work it turned out that the design of the general filesystem layer in the operating system (VFS) affects performance even more than the physical layout of data on disk. VFS is shared by all filesystems on the particular operating system; thus it's not easy to change it on systems with many filesystem drivers, such as Linux. I developed a new VFS design on my new operating system, Spad. The Spad kernel contains a novel feature, cross-file readahead, that provided 4-times better performance on Pentium 4 with Maxtor IDE disk when alternatively accessing two directory trees. The Spad kernel provides better results for cached operations than the Linux kernel; for postmark it is twice faster on Pentium 4 than the Linux implementation.

References

[1] Brent Welch — The File System Belongs in the Kernel
`Proceedings of the 2nd Usenix Mach Symposium, 1991, pp. 233-250`

[2] Dennis M. Ritchie — The Evolution of the Unix Time-sharing System
`Lecture Notes in Computer Science #79: Language Design and Programming Methodology, Springer-Verlag, 1980`

[3] L. W. McVoy, S. R. Kleiman — Extent-like Performance from a UNIX File System
`Proceedings of Usenix Winter Conference, 1991, pp. 33-43`

[4] A. Forin, G. R. Malan. — An MS-DOS File System for UNIX
`Proceedings of Usenix Winter Conference, 1994, pp. 337--354`

[5] R. Duncan — Design goals and implementation of the new High Performance File System
`Microsoft Systems Journal 4/5, pp. 1-13`

[6] D.Bridges — Inside the High Performance File System
`Significant Bits magazine, 1996`

[7] M. K. McKusick, W. N. Joy, S. J. Leffler, R. S. Fabry — A Fast File System for UNIX
`ACM Transactions on Computer Systems 2, 1984, vol. 3, pp. 181-197`

[8] M. K. McKusick, G. R. Ganger — Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem
`Proceedings of the 1999 Usenix Annual Technical Conference, Freenix track, pp. 1-18`

[9] G. Ganger, M. McKusick, C. Soules, and Y. Patt. — Soft updates: a solution to the metadata update problem in file systems
`ACM Transactions on Computer Systems, 2000, pp. 127-153`

[10] M. K. McKusick — Running ”fsck” in the Background
`Proceedings of the Usenix BSDCon 2002 Conference, pp. 55-64`

[11] R. Card, T. Ts'o, S. C. Tweedie — Design and implementation of the second extended filesystem
`Proceedings of the First Dutch International Symposium on Linux, 1994`

[12] D. Phillips — A Directory Index for Ext2
`Proceedings of the 2001 Annual Linux Showcase and Conference`

[13] T. Y. Ts'o — Planned Extensions to the Linux Ext2/Ext3 Filesystem
`Proceedings of the Usenix 2002 Annual Technical Conference`

[14] S. C. Tweedie — Journaling the Linux Ext2fs Filesystem
`LinuxExpo '98, pp. 25-29, 1998`

[15] Steve Best — JFS log: How the journaled file system performs logging
`Proceedings Of The 4th Annual Linux Showcase & Conference, 2000, pp. 163-168`

[16] J. Mostek, B. Earl, S. Levine, S. Lord, R. Cattelan, K. McDonell, T. Kline, B. Gaffey, R. Ananthanarayanan — Porting the SGI XFS File System to Linux
`Proceedings of the Usenix 2000 Annual Conference`

[17] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau — End-to-end Data Integrity for File Systems: A ZFS Case Study
`Proceedings of the 8th Usenix Conference on File and Storage Technologies, 2010, pp. 29-42`

[18] L. N. Bairavasundaram., G. R. Goodson, S. Pasupathy, J. Schindler — An analysis of latent sector errors in disk drives
`ACM SIGMETRICS Performance Evaluation Review, Vol. 35, Issue 1, 2007`

[19] O. Rodeh — B-trees, shadowing, and clones
`ACM Transactions on Storage (TOS), Vol. 3, Issue 4, 2008`

[20] R. Fagin — Extendible Hashing: A Fast Access Mechanism for Dynamic Files
`ACM Transactions on Database Systems, 1979, pp. 315-344`

[21] S. R. Soltis, T. M. Ruwart, M. T. O'Keefe — The Global File System
`Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems, 1996, pp. 319-342`

[22] S. R. Soltis, G. M. Erickson, K. W. Preslan, M. T. O'Keefe, T. M. Ruwart — The Global File System: A Filesystem For Shared Disk Storage
`IEEE Transactions on Parallel and Distributed Systems, 1997`

[23] Steven Whitehouse — The GFS2 Filesystem
`Proceedings of the Linux Symposium 2007, pp. 253-259`

[24] M. Patočka — Maintaining consistency in disk file systems
`Proceedings of ITAT 2004`

[25] M. Patočka — Using crash counts to maintain filesystem consistency
`Proceedings of the CITSA 2006, vol. 3, pp. 118-122`

[26] V. Prabhakaran, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau — Analysis and Evolution of Journaling File Systems
`Proceedings of the Usenix 2005 Annual Technical Conference, pp. 105-120`

[27] M. Rosenblum , J. Ousterhout — The LFS Storage Manager
`Proceedings of the 1990 Summer Usenix, pp. 315-324`

[28] M. Rosenblum and J. Ousterhout — The design and implementation of a log-structured file system
`ACM Transactions on Computer Systems, 1992, pp. 26-52`

[29] M. I. Seltzer, K. Bostic — An Implementation of a Log-Structured File System for UNIX
`Proceedings of the 1993 Winter Usenix Conference, pp. 307-326`

[30] W. Wang, Y. Zhao, and R. Bunt — HyLog: A High Performance Approach to Managing Disk Layout
`Proceedings of the 3rd Usenix Conference on File and Storage Technologies, 2004, pp. 145-158`

[31] J. E. Johnson, W. A. Laing — Overview of the Spiralog File System
`Digital Technical Journal, volume 8, number 2`

[32] D. Roselli, J. R. Lorch, T. E. Anderson — A Comparison of File System Workloads
`Proceedings of the Usenix 2000 Annual Conference, pp. 41-54`

[33] G. R. Ganger, Y. N. Patt — Metadata update performance in file systems
`Proceedings of the 1st Usenix conference on Operating Systems Design and Implementation, 1994`

[34] M. I. Seltzer, G. R. Granger, M. K. McKusick, K. A. Smith, C. A. N. Soules, C. A. Stein — Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems
`Proceedings of the Usenix 2000 Annual Conference`

[35] David A. Patterson, Garth Gibson, Randy H. Katz — A case for redundant arrays of inexpensive disks
`Proceedings of the ACM SIGMOD international conference on Management of data, 1988, pp. 109-116`

[36] H. Garcia-Molina, K. Salem — The impact of disk striping on reliability
`IEEE Database Engineering Bulletin, 1988, pp. 26-39`

[37] A. Sweeney — Scalability in the XFS File System
`Proceedings of the Usenix 1996 Technical Conference, pp. 1-14`

[38] G. R. Ganger and M. F. Kaashoek — Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files
`Proceedings of the Usenix 1997 Annual Technical Conference`

[39] G. R. Ganger, M. F. Kaashoek — Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files
`In Proceedings of the 1997 Usenix Technical Conference, 1997, pp. 1-17`

[40] K. A. Smith and M. Seltzer — A Comparison of FFS Disk Allocation Policies
`Proceedings of the Usenix 1996 Annual Technical Conference`

[41] A. B. Downey — The structural cause of file size distributions
`Proceedings of the 2001 Joint International Conference on Measurement and Modeling of Computer Systems`

[42] A. Riska, E. Riedel — Disk Drive Level Workload Characterization
`Proceedings of the Usenix 2006 Annual Technical Conference, pp. 97-102`

[43] Werner Vogels — File system usage in Windows NT 4.0
`Proceedings of the ACM SOSP 1999, pp. 93-109`

[44] J. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, J. G. Thompson — A Trace-Driven Analysis of the UNIX 4.2 BSD File System
`ACM SIGOPS Operating Systems Review, Vol. 19, Issue 5, 1985, pp. 15-24`

[45] K. M. Evans, G. H. Kuenning — A study of irregularities in file-size distributions
`Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems`

[46] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf — Characterizing flash memory
`Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009, pp. 24-33`

[47] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, R. Panigrahy — Design tradeoffs for SSD performance
`Proceedings of the Usenix 2008 Annual Technical Conference, pp. 57-70`

[48] S. Boboila, P. Desnoyers — Write Endurance in Flash Drives: Measurements and Analysis
`Proceedings of the 8th Usenix Conference on File and Storage Technologies, 2010, pp. 115-128`

[49] H. Kim and S. Ahn — A Buffer Management Scheme for Improving Random Writes in Flash Storage
`Proceedings of the 6th Usenix Conference on File and Storage Technologies, 2008`

[50] I. Dowse, D. Malone — Recent Filesystem Optimizations in FreeBSD
`Proceedings of the Usenix 2002 Annual Technical Conference, pp. 245-258`

[51] M. Seltzer, K. A. Smith — File System Logging Versus Clustering: A Performance Comparison
`Proceedings of the 1995 Usenix Annual Technical Conference`

[52] Juan Piernas — DualFS: A New Journaling File System without Meta-data Duplication
`16th Annual ACM International Conference on Supercomputing, 2002, pp. 137-146`

[53] Marshall Kirk McKusick — Fsck-The Unix File System Check Program
`Computer Systems Research Group, UC Berkeley, 1985`

[54] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau — SQCK: A Declarative File System Checker
`Proceedings of the 8th Usenix Symposium on Operating Systems Design and Implementation, 2008`

[55] L. N. Bairavasundaram, G. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau — An Analysis of Data Corruption in the Storage Stack
`Proceedings of the 6th Usenix Conference on File and Storage Technologies, 2008, pp. 223-238`

[56] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau — IRON File Systems
`Proceedings of the ACM SOSP 2005, pp. 206-220`

[57] M. Patočka — VSPACE: An object oriented framework for communication between system components
`Proceedings of the CCCT 2006, vol. 2, pp. 318-324`

[58] M. Patočka — An Architectures for High Performance File System I/O
`Proceedings of the WASET, 2007, vol. 23, pp. 250-255`

[59] V. S. Pai, P. Druschel, W. Zwaenepoel — IO-Lite: A Unified I/O Buffering and Caching System
`ACM Transactions on Computer Systems, 1997, pp. 15-28`

[60] C. D. Cranor, G. M. Parulkar. — The UVM Virtual Memory System
`Proceedings of the 1999 Usenix Annual Technical Conference, pp. 117-130`

[61] Z. Zhang and K. Ghose — yFS: A Journaling File System Design for Handling Large Data Sets with Reduced Seeking
`Proceedings of the 2nd Usenix Conference on File and Storage Technologies, 2003, pp. 59-72`

[62] A. E. Papathanasiou, M. L. Scott — Aggressive Prefetching: An Idea Whose Time Has Come
`HotOS X, Tenth Workshop on Hot Topics in Operating Systems, 2005`

[63] B. S. Gill and L. A. D. Bathen — AMP: Adaptive Multi-stream Prefetching in a Shared Cache
`Proceedings of the 5th Usenix Conference on File and Storage Technologies, 2007, pp. 185-198`

[64] T. M. Kroeger, D. E. Long — Design and Implementation of a Predictive File Prefetching Algorithm
`Proceedings of the 1999 Usenix Annual Technical Conference`

[65] C. S. Ellis, D. Kotz — Prefetching in file systems for MIMD multiprocessors
`IEEE Transactions on Parallel and Distributed Systems, Vol. 1, 1990, pp. 306-314`

[66] P. Cao, E. W. Felten, A. R. Karlin, K. Li — A Study of Integrated Prefetching and Caching Strategies
`In Proceedings of the ACM SIGMETRICS, 1995`

[67] Jeffrey Katcher — PostMark: A New File System Benchmark
`Technical Report TR3022, Network Appliance Inc., 1997`

[68] Record Management Services
`http://en.wikipedia.org/wiki/Record_Management_Services`

[69] A General-Purpose File System For Secondary Storage
`http://www.multicians.org/fjcc4.html`

[70] FAT filesystem
`http://en.wikipedia.org/wiki/File_Allocation_Table`

[71] M. Patočka — HPFS Driver for Linux
`http://artax.karlin.mff.cuni.cz/~mikulas/hpfs/`

[72] R. Russon, Y. Fledel — NTFS documentation
`http://www.linux-ntfs.org/`

[73] J. Kratochvíl — Captive: The first free NTFS read/write filesystem for GNU/Linux
`http://www.jankratochvil.net/project/captive/`

[74] NTFS-3G: Stable Read/Write NTFS Driver
`http://www.ntfs-3g.org/`

[75] NTFS-3G Quality and Test Methods
`http://www.ntfs-3g.org/quality.html`

[76] M. K. McKusick — Soft Updates, Snapshots and Background Fsck
`http://www.mckusick.com/softdep/`

[77] Ext2-OS/2
`http://freshmeat.net/projects/ext2-os2/`

[78] Ext2 File System Driver for Windows NT 4.0
http://winext2fsd.sourceforge.net/

[79] Mac OS X Ext2 Filesystem
http://sourceforge.net/projects/ext2fsx/

[80] Bug in Ext3
http://www.uwsg.iu.edu/hypermail/linux/kernel/0310.3/0887.html

[81] Namesys — ReiserFS
http://www.namesys.com/

[82] IBM — Comparison of JFS and JFS2
http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.baseadmn/doc/baseadmndita/fs_jfs_jfs2.htm

[83] SGI — XFS: A high-performance journaling filesystem
http://oss.sgi.com/projects/xfs/

[84] Sun Microsystems — ZFS On-Disk Specification
http://opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf

[85] The Jargon File
http://www.catb.org/~esr/jargon/html/C/critical-mass.html

[86] Description of B-trees
http://en.wikipedia.org/wiki/B-Tree

[87] J. Hundstad — Journaled filesystems, known instability
http://www.uwsg.indiana.edu/hypermail/linux/kernel/0501.2/0465.html

[88] Bugs in XFS
http://marc.theaimsgroup.com/?l=linux-xfs&r=1&b=200405&w=2

[89] M. Patočka — Srovnávací studie jádra Linuxu a FreeBSD, page 70
http://artax.karlin.mff.cuni.cz/~mikulas/doc/dipl.ps.gz

[90] D. Phillips — Explanation of Phase Tree
http://www.uwsg.iu.edu/hypermail/linux/kernel/0007.2/0305.html

[91] R. Avitzur — A Programmer's Apology: September 2006 Archives
http://avitzur.hax.com/2006/09/

[92] J. Andrews — Linux 2.6.6-mm5, Request Barriers
http://kerneltrap.org/node/3176

[93] S. Best, D. Kleikamp — JFS layout
http://mics.org.uk/usr/share/doc/packages/jfsutils/jfslayout.pdf

[94] F. Buchholz — The structure of the Reiser file system
http://www.cerias.purdue.edu/homes/florian/reiser/reiserfs.php

[95] Jeff Bonwick — Space Maps
http://blogs.sun.com/bonwick/date/200709

[96] Val Henson — Automatic performance tuning in the zettabyte file system
http://www.filibeto.org/~aduritz/truetrue/solaris10/henson-self-tune.pdf

[97] PC Perspective — Long-term performance analysis of Intel Mainstream SSDs
http://www.pcper.com/article.php?aid=669

[98] LWN — The Orlov block allocator
`http://lwn.net/Articles/14633/`

[99] AI-complete problem
`http://en.wikipedia.org/wiki/AI_complete`

[100] J. Andrews — Linux: Reiser4 and the Mainline Kernel
`http://kerneltrap.org/node/5679`

[101] Avadis Tevanian - FSX - file system exerciser
`http://www.codemonkey.org.uk/projects/fsx/`

Appendix A: Content of the CD

The enclosed CD contains the following files and directories:

`THESIS.PDF` — this thesis.
`BENCH/` — source code of benchmarking program and data.
`VYSLEDKY/` — results obtained from benchmarks and scripts to convert them to graphs.
`SPADFS/` — stable release of SpadFS for Linux.
`SPAD/` — source code of the Spad operating system. Implementation of SpadFS can be found in subdirectory `DRIVERS/FS/SPADFS/`.