A Dual-Disk File System: `ext4`

Mihai Budiu

April 16, 1997

Contents

1 Abstract

This paper presents the design and implementation of a Unix-compatible file system, `ext4`, which uses two disk partitions to store its data. One partition is used exclusively for directory-related information, and one partition for ordinary files. The file system is developed as a modification of the Linux-ubiquitous `ext2` file system. The aim is to benefit from:

- parallelism in the access to the two partitions, if they are stored on disks with separate controllers;
- simplified layout policies enhanced to favor access patterns specific for each type of file.

2 How to Read This Paper

This paper tries to be self-contained, and thus is a bit long. The many details are explained having in mind people who may try to replicate my experience. Some very common details (e.g. physical disk geometry) are considered well-known and not treated.

Some hints for the reviewers seem welcome: the essential sections describing my work are 1, 3.1, 4.1.5, 5, 6; one can safely read only these parts and have an image of the project.

3 Motivation; Related Work

In a world where processors and networks are faster and faster, the rate of increase of disk access speed is lagging behind. A decade has seen only a minor improvement in seek time: transfer time and rotational delay have witnessed a better improvement, due mainly to increased data density. (This is also the factor which enabled disk size to grow so fast.)

To compensate for this phenomenon considerable effort goes into squeezing every bit of performance from a file system by exploiting all kinds of idiosyncrasies of the mechanics, hardware and software. The efficiency factors targeted by file system writers are higher bandwidth and lower latency for file access.

3.1 Related Work

There seems to have been at least one attempt to implement a system similar to `ext4`, but not in that exact form. A survey of the literature and an inquiry on `comp.os.research` led me to the conclusion that the attempt might prove at least insightful.

The closest match to `ext4` is a three-disked file system, log-structured, which uses one disk for data, one for directory-type data and one for the log itself. It is described in [1].

3.2 Other Approaches for Increasing Efficiency

Here is a summary of the methods and design alternatives that seem to be used effectively for increasing file system performance, or just the user perception of it:

- Reduced physical access time by careful data layout: if the blocks of data that are "consumed" consecutively by the OS are set "close to each other" in disk geometry (which does not necessarily mean contiguously placed!) then substantial increase in speed can be perceived. Typical figures are 3ms for track-to-track seek but 14ms for average seek time. One immediate consequence is that all file systems treat the disk as a sequence of blocks: it is almost as cheap to read a block (of consecutive bytes) as it is to read a single byte.

  Data layout policies tend to treat specially the pieces of data which are probably used in sequence, like consecutive blocks of a same file, or inodes referred by a directory. See [6].

A special form of data layout is observed in log-structured file systems, described briefly below (and more in [7]).

- Aggressive (client) caching: caches remain the most effective means of speeding up file access, because of the huge difference between the access time of RAM and disks (the ratio has 5 orders of magnitude!). Caching can be done both at the server side and at the client side (in the case of network file-systems); client caching pays most ([10], [11]).

  Observe that usually caching on the client disk is slower than caching on the server RAM, because the local area networks can provide a much higher bandwidth than the disks.

- Asynchronous writes: the most common way of using caches is for delaying the writing of the data, allowing for:

  - Write batching (e.g. in the log-structured file systems) [7]; this gives fewer seeks for the same transfered quantity;
  - Write re-ordering (usually done by the disk driver), which tries to minimize the mechanical motion;
  - Possible avoidance of disk transfer altogether for shortly lived data (e.g. temporary files);
  - Writing only the final result of a sequence of modifications.

- Read-ahead: a more intelligent use of the caches tries to guess ahead of time which blocks are needed, and to initiate I/O for them before the applications trigger the read. When the read finally happens, the data is already present in the cache and can be serviced without user perceived latency. Needless to say, the risk is to make an unnecessary disk access. Fortunately, sequential accesses can be predicted quite reliably from past file access patterns: if a file reads a couple of blocks in sequence, very likely it will continue to do so [4].

- RAID (Redundant Arrays of Independent Disks) stripping: writing/reading successive blocks on separate disks can be a big win, as the data transfer to one disk can proceed in parallel to the transfer from the neighbours (the bus allows high enough rates).

  Sometimes RAID disk systems are designed to provide increased reliability, using one disk for storage of redundant information (i.e. parity). For instance, redundant RAID schemes with two disks (which are also called "mirrored" disks); the redundancy schemes tend to be less efficient, especially on small writes, because they have to read all the data blocks on which the redundant checksum depends.

- log writes: "logging" is a technique borrowed from the database community, in which instead of writing the data, one describes the modifications made to the data. Reading the last modifications enables you to determine the actual value. Each piece of data will have "versions", and each version occupies a new place on the disk [7].

  This technique is effective only in the presence of very large caches, because it assumes most traffic is for writes; it also assumes that large writes can be make with a single disk access, allocating the new versions of the blocks contiguously.

This is why the scheme needs a garbage collector to get rid of the old versions and to compact the free space; contiguity of free space is a requisite for high write bandwidth. Garbage collection is an important source of overhead.

- Separate disks for logging modifications: in order not to hamper the fast contiguous writes in log file systems, you can try to keep the log on a disk, and to move the data cleaned by the garbage collector to another one. This will quickly segregate long-lived data [14].
- Replication for load balancing (distributed file systems): the same idea as in redundant RAID devices, but this time used at the entire file system level: if you duplicate the data, you can distribute requests from many clients evenly across servers, which will then have shorted queues of outstanding requests, implying a decreased latency. Special techniques have to be used to ensure consistency among servers (e.g. quorum[8], immutable files[19] or master/slave relationships).
- Request batching: used in some NFS installations and a basic technique in log file systems: gather a lot of data that you write in one big shot [11]; this implies fewer seeks.
- Quick indexing of blocks, directories, inodes (e.g. hashing/B-trees): in the Unix file system, as will be clear from the discussion below, many levels of indirection have to be traversed to locate a block on disk (the whole directory path and up to three indirection levels once the "inode" is located). Also, the in-core data structures (e.g. inodes of open files) have to be retrieved quickly; using effective data structures may help [16].
- Use idle time for cleaning/parity computation: if the file server experiences burst activity, it is a good idea to use the quiescent periods for mundane tasks, like cleaning (logs), parity computation (RAID), compression (of files), compaction (of free space) [15], de-fragmentation (of files).
- Delayed block allocation/pre-allocation/re-allocation: generally blocks are allocated to files at the time of the overflow of the current allocated space. Usually the `write` system call will proceed like this:

  ```
  while (size) {
          bh = allocate_buffer();
          copy_data(source, bh, MIN(size, buffer_size));
          size -= buffer_size;
          if (size < 0) size = 0;
  }
  ```

  So even for a bulk write, the buffers are allocated one by one. This may not produce contiguous buffers, as the allocation routine has no hints about the previous allocations. (This code may also block in the `allocate_buffer()` function, so many other allocations may happen in other threads before this loop terminates.)

  Thus, although the kernel knows from the beginning how much space it will need for file growth during the current system call, usual schemes fail to allocate this space contiguously, even if it is available!

Four (partial) workarounds exist:

- Use "extent" allocation: an extent is a contiguous range of blocks [20]. The allocator will allocate the whole space necessary to accommodate the growth of the file in one piece (using the information available at the time of the `write` system call). (This complicates the cache management, as the region might not fit entirely in the cache);
- Re-allocate the blocks at cache flush time so that blocks logically contiguous in a file become contiguous on the disk. This scheme was implemented in the context of some FFS versions [17];
- Whenever you have to allocate a block, preallocate a couple of sequent blocks. Drop (free) those blocks later if the file growth is not sequential. This scheme is used in the `ext2` file system [4];
- Do not allocate the blocks on disk until the cache flush time comes. This scheme is used in the IRIX file system [16].

- Kernel/user space file systems: this is part a multifaceted debate about where to place the file system in the operating system; kernel space file systems avoid multiple context switches, while special-purpose user-level file systems may be tailored for special needs.
- Multithreaded file systems (e.g. a separate thread for cleaning, or for each disk): even if the classical file systems act in the context of multiple caller processes, the "multiple threads" are non-preemptively scheduled, and are demand-driven (i.e. do something only when you need it). Introducing (kernel-level) preemptive scheduled threads in the file system may enable you to carry multiple activities in parallel easily [16], [15].
- Transparent data compression (e.g. stackers): while data compression takes time, it also reduces space, which improves disk transfer time. Achieving higher rates with compressed data may be feasible, but it has the danger of transforming the file system activity from one which is I/O bound to one which is CPU bound, wasting other important resources.

Although this enumeration may seem overly general and unrelated to the main topic of this paper, most of the ideas are effectively used in the `ext4` file system (mostly because they are inherited from its `ext2` ancestor). The next section of this paper will try to prove this assertion.

4 Architecture

Basically `ext4` tries to make use of a modified type of RAID-0 technique. RAID-0 techniques transform two partitions (disks) into a single "virtual" partition, either by concatenating their blocks, or by interlacing them (i.e. even-numbered blocks taken from one physical partition and odd-numbered from the other). The RAID-0 has no knowledge about the file system structure, and achieves "striping" at the device driver level (note in passing that Linux has a RAID-0 facility, the `md` driver, which however has no connection to our project).

Our file system tries to take advantage of the knowledge of block contents, and splits data on two disks at a "higher" level.

4.1 Evolutionary Path

The `ext4` file system is the result of a long evolution of file system technology, which starts with the initial Unix file system, designed in the seventies by Ken Thomson. Many ideas are still basic (like inodes); a chronological description of the evolution of the Unix file systems, besides the interest in itself, will shed some light on the choices which gave rise to the design of `ext4`. The informed reader may skip with no loss the following paragraphs, down to the one discussing `ext4`.

4.1.1 The Classical Unix File System (UFS)

The "prototype" file system remains the original Unix file system, designed by Ken Thomson. Its modularity, cleanliness, and simplicity are offset only by the, sigh, low efficiency. In a way or another, the sequent systems are all "patches" to the original design, and they try to squeeze a few more drops of performance by sacrificing the elegance of the design: "if it works fast it doesn't matter if it's ugly." All the sequent systems provide a superset of the original interface.

The file model in UNIX is very simple: a simple byte array with a very large maximal size.

The figure 1 represents the placement of the kernel file system procedures amongst other kernel services.

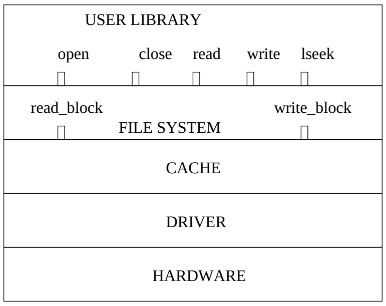

Figure 1: File System Placement in the Operating System

The driver (and the cache) offer access to the disk viewed like a huge *block* array. The file system reads and writes such blocks in one operation. Each disk partition has to hold an independent file system.

The file system uses the partition blocks in the following manner (figure 2):

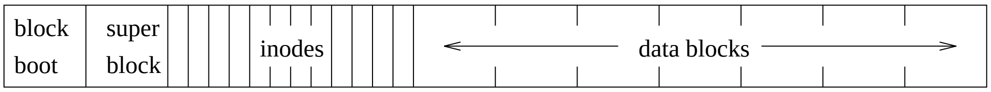

Figure 2: Partition Utilization by File System

The *superblock* contains a description of the *global* parameters of the file system. Some examples are: partition size, partition mount time, number of inodes, free blocks; free inodes, file system type ("magic number").

Next a number of blocks is allocated for storing individual file descriptions. Each file is described by an *inode* or information node. The number of inodes is allocated statically, at file system creation time. All relevant file attributes are kept in inodes, including a representation of the list of the blocks used by the file, (figure 3). The indirect blocks need to be present only if the size of the file is big enough (i.e. the simple indirect pointer may be NULL). This scheme has many merits; let us only observe that files with "holes" inside can exist.

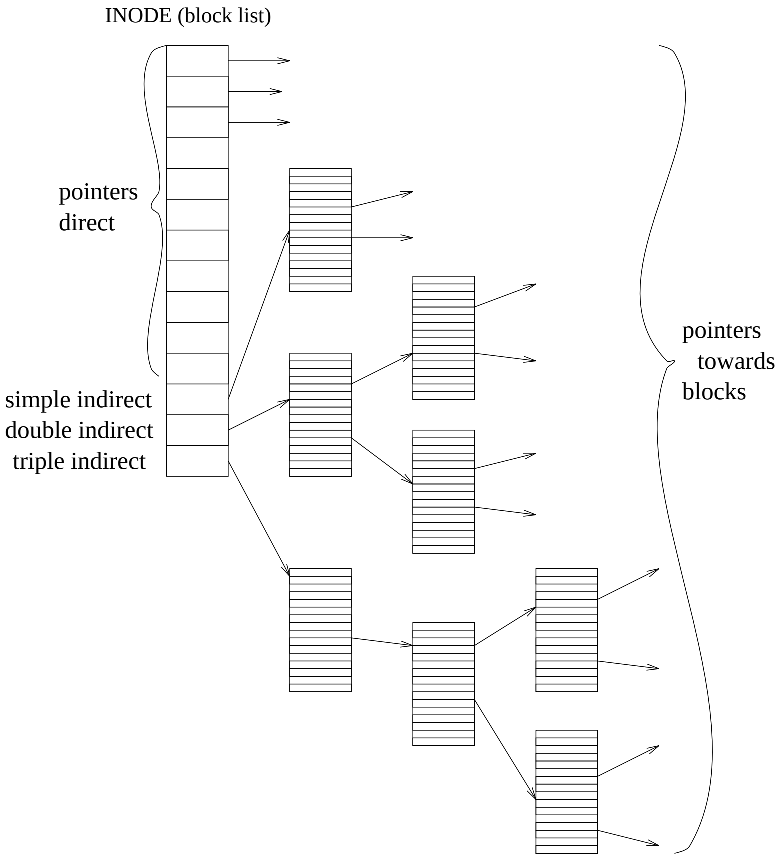

Figure 3: Block List Representation in Inodes

The directories are actually in all respects like ordinary files (i.e. they have an inode, data blocks, they grow in the same manner), but the operating system cares about the *meaning* of the directory contents. The directory structure is very simple: each is an *array of links*.

A *link* is a structure which associates a name (string) to an inode number. Figure 4 shows the directory structure.

Each file has to have at least one link in one directory. This is true for directories too, except for the *root* directory. All files can be identified by their *path*, which is the list of links which have to be traversed to reach the file (either starting at the root directory, or at the *current* directory). A file can have links in many directories; a directory has to have a single link towards itself (except "." and ".."), from its *parent* directory.

The Unix program **mkfs** transforms a "raw" partition into a file system by creating the superblock, initializing inodes and chaining all the data blocks into a huge list of blocks available for future file growth.

The free blocks are practically grouped into a big fictitious file, from which they are retrieved

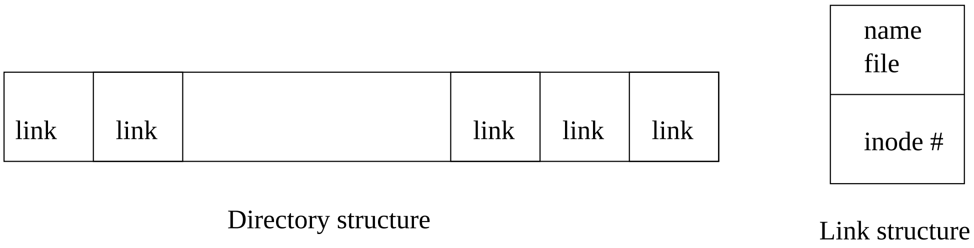

Figure 4: Directory and Link Structure

on demand (when other files or directories grow), and to which they return on file removal or truncation.

Let us observe that all operations carried on files have to bring the relevant data (e.g. inodes) in-core (in the RAM). The in-core representation of the data structures is significantly more complex than the structure on disk, because a lot of information implicit on the disk is missing in-core (for instance the inode number, the device, the file system type).

4.1.2 The Berkeley Fast File System (FFS)

The main deficiency of the Version VII file system (the one presented above) lies in its total ignorance of disk geometry. After awhile, the free list has a random ordering, so blocks are allocated haphazardly, leading to very long seek times inside files.

The FFS [6] tried to solve this problem, by forcing clustering of the data. The main idea was to divide the disk into "cylinder groups", which are sets of neighbouring cylinders (i.e. with small seek time between the component tracks). Each cylinder is treated like an independent partition in UFS, having its own superblock, inodes, free list and data blocks, but allowing overflow into neighbouring cylinder groups.

FFS tried to allocate "logically close" blocks close. For instance, it allocates (if possible) regular file inodes in the same group as the directory holding them, and data blocks for a file in the same group as the file's inode. On the contrary, it allocates directory inodes on the less loaded group, and it forces the change of cylinder group at each megabyte of file size (to avoid fragmentation inside a cylinder group). The allocation uses geometry parameters to compute a "closest block".

The allocation schemes of FFS work well if the disk is loaded under 90% of full capacity, else they trash.

Besides improved layout, FFS brought some other ideas to the area, like:

- Very large blocks (8Kb) for fewer seeks and fewer indirections (because more files are described by direct blocks);
- "Fragments" (splitting a block in pieces) for avoiding internal fragmentation losses in incompletely used blocks;
- Symbolic links;
- Variable sized file names (and long file names), opposed to the 14 character limit in the old file system;

- Locking of files;
- The atomic `rename()` operation;
- Different update policies for critical file system information (e.g. inodes, superblocks): synchronous writes, to minimize losses in case of crash;
- Replicated superblock in each group;
- The `/lost+found` directory for repaired files;
- Disk quotas for multiuser environments;
- The linked list (tree) of free blocks is replaced with a bitmap. This has the tremendous advantage that contiguity of free space can be immediately checked.

4.1.3 The Virtual File System Switch

SunOS introduced yet another extension to the file system technology, which allowed a kernel to simultaneously support multiple file systems which differ wildly in the internal architecture. This was built to allow a kernel to support simultaneously a "local" file system and an NFS-mounted file system.

The technique, named "Virtual File System Switch, (VFS)" is reminiscent of object-oriented programming: you have a base class, named `filesystem` which has a bunch of virtual methods which are overridden by every other custom file system present in the kernel.

VFS works on its own data structures, called "virtual inodes", or "vnodes", on which it can carry many operations which may be file system independent. Each vnode describes a file belonging to some file system. Whenever VFS needs custom operations, it dispatches the call to the file system owning the inode (figure 5).

All information about physical file placement is file system dependent (thus unknown to VFS); this allows us to add yet another file system which uses special layout (in our case two disks) without changing anything in the VFS layer.

4.1.4 The `ext2` Linux File system

The `ext2` file system draws heavily on the legacy of the FFS and VFS systems. It has its own characteristics, though:

- Gives up block fragments; space is less an issue with current disk sizes; on the other hand fragment relocation on file growth is no more a source of overhead;
- Uses (cylinder) groups, with bitmaps for free inode and free block tracking (figure 6);
- Uses pre-allocation techniques to achieve contiguity for file blocks; each growing file tries to reserve a number of consecutive blocks, which are released if the growth is not sequential;
- All bitmaps are reduced to one block in size, for search efficiency reasons;

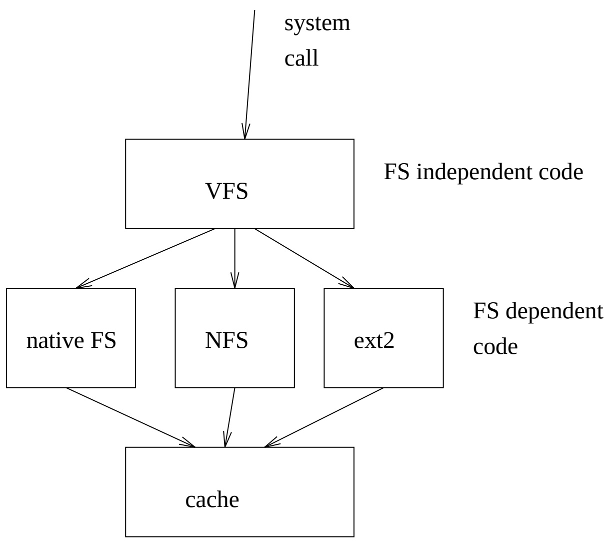

Figure 5: Virtual File System Switch

- Immutable files, append-only files are enforced by the kernel; `ioctl()`'s handle their attributes;
- Directory entries have variable size; directory manipulation is allowed only via special system calls (e.g. `readdir()`, and not `read()`);
- "Clean unmount" bits in the superblock allow to bypass costly consistency checks at boot time;
- "Fast" symbolic links (store information in the inode part reserved for block pointers if it fits, and do not allocate a block);
- Some extra features are planned but not yet implemented apparently: transparent compression, file undelete.

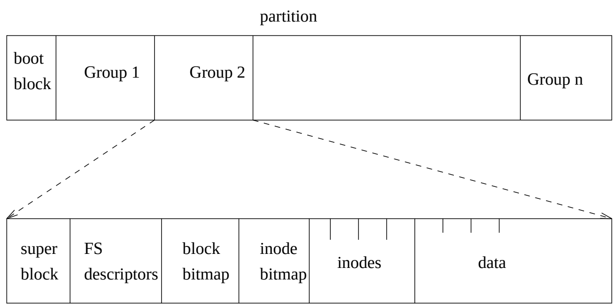

Figure 6: Group Structure in `ext2`

The power of the VFS is apparent in Linux in its entirety, as Linux supports with the same easiness as many as 10 different file systems, ranging from NFS to DOS FAT.

There are three principal data structures in the Linux VFS layer, which point to file system dependent and independent parts. These are:

- The superblock of each mounted file system;
- the inodes of each "loaded" object (file, directory, pipe, etc.);
- the `file` structures, which describe open files.

Each in-core version of such an object has a structure with pointers to the handlers manipulating that specific object. Figure 7 illustrates this fact for the inodes. (In official SunOS terminology the structure is a vnode, and the file system-dependent part is the inode, but Linux names both inodes.)

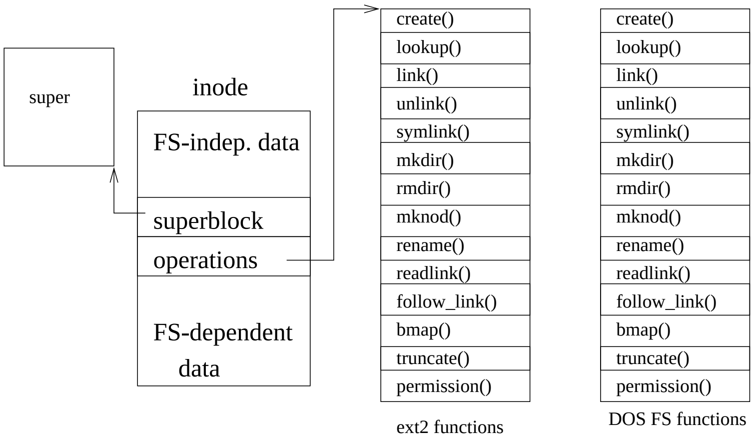

Figure 7: Inode structure

Here is a typical piece of code from the kernel, in the file system-*dependent* routine of `ext2` which loads an inode:

```
if (REGULAR(inode->mode))
        inode->operations = &ext2_file_inode_operations;
else if (DIRECTORY(inode->mode))
        inode->operations = &ext2_dir_inode_operations;
else if (SYMLINK(inode->mode))
        inode->operations = &ext2_symlink_inode_operations;
else ...
```

The VFS will call the inode operations indirectly (e.g. `inode->operations->link()`), without having to know anything about the implementation.

4.1.5 The `ext4` File System

`ext4` is basically a refinement of `ext2`, which uses two partitions simultaneously, ideally placed on separate disks.

Both partitions contain the information ordinarily found inside a file system: inodes, direct and indirect blocks, superblocks and bitmap information. The only difference between the two partitions is that all directories will sit on one of them (both the blocks and inodes), and ordinary files on the other one.

Right now all other types of files (symbolic links, named pipes, special files) are kept on the same partition as the regular files, although their right place obviously is the directories' partition, because their usage pattern is supposedly similar to the directories themselves. This change should not prove difficult.

This layout allows the operations on files and directories to proceed in parallel. Requests for reading/writing a directory block and a file block can be processed simultaneously, reducing user-perceived latency.

Let us observe that simultaneous outstanding requests for directory and file manipulation arise even in the context of I/O of a single user process, because of the write-behind and read-ahead actions of the cache; this means they are not an exotic circumstance, and we are indeed trying to solve a real problem.

Basically two things have to be changed in `ext2` to obtain `ext4`:

- Mounting and unmounting operations have to operate on two partitions at once, and cross-check the validity of the data structures;
- All operations which deal with loading/saving of blocks have to be customized, to choose one or the other partition, according to the final destination of the manipulated block.

5 Development

The development platform of the project is Linux OS, with a kernel version 2.1.18.

The development comprises not only a modification of the kernel sources of the `ext2` file system, but also of the associated utilities, including the `mkfs` and `fsck` programs. One conspicuous absent from this suite is the `mount/umount` executable programs, which don't even need recompilation. The kernel does most of the argument parsing!

Only one real design difficulty had to be faced, and this is concerned with the availability of the information. Recall that a directory holds in the links only pairs $\langle$ name, inode number $\rangle$. To open a file one needs first to load the corresponding inode into memory.

But having two partitions, I end up having twice the inode 5. Which inode 5 is to be loaded when a directory entry contains the number 5? The one on the partition with directories, or the one on the partition with files? To disambiguate I played a dirty trick, recording the partition number (and implicitly the file type...) inside the inode number. The last bit was chosen for this purpose.

In this way all directories have even-numbered inodes, and all regular files have odd-numbered inodes. This change is transparent to the applications and to the VFS, who do give any meaning to inode numbers.

Because disk blocks can be referenced only from one place at a time (i.e. two files cannot share a block) we did not use the same technique for block numbering. But then every time we need to read/write say block 7, we must have at hand information indicating which partition is considered. We distinguish some cases (refer again to figure 6):

- If the block belongs already to a file, the file inode number hints to the partition, as described above.
- The blocks which belong to the superblocks and cylinder group descriptors are loaded only at mount/umount time, when information about the partition in question is available.
- The free blocks and bitmaps which describe free blocks and free inodes are accessed only to (de)allocate blocks/inodes for files. But:
  - When I deallocate a block, I know from which file it comes, and thus I know on which partition it resides;
  - When I allocate objects:
    * When I want to allocate an inode, I am told which file type I want to create, so I know on which partition to allocate the inode;
    * When blocks are allocated, they are allocated for the growth of an existing file, and its inode is known to the block allocation routine; but then we also know on which partition to allocate the blocks.

This proves that at anytime there is no ambiguity about the partition to be accessed to manipulate an indicated block; this is why we don't need to use discriminatory schemes in block numbering to indicate the partition, like we did for inodes.

Here is a brief chronological account of the phases of the development process:

1. Careful study the source code; isolation of the parts that need changes;
2. The development proceeded by modifying the existing source code of the `ext2` file system, and the source files of the file system utilities `mkfs`, `fsck`, etc. A suite of shell/awk scripts traversed the trees of the source files, applying uniform modifications to the identifier and file names. Basically the string "ext2" got replaced to "ext4" and the string "e2fs" to "e4fs";
3. This change produced a fully functional file system. The modified code was added to the kernel, compiled and the system rebooted. At this point I could mount ordinary `ext2` partitions under the name `ext4`;
4. Changes to the header files which define the superblock structure were made; no other data structures needed redefinition;
5. The files which deal with superblock manipulation were changed to load information from both partitions;
6. I tracked down any input/output operations at the cache level (where buffer blocks are read/written by the file system) and directed them to use the correct partition;
7. I modified the block and inode allocation procedures to use the appropriate partition;
8. I had to modify `mkfs` to create the correct data structures on each of the partitions;
9. Debugged the resulting code.

6 Future Work

Some work remains to be done for the system to prove useful. The following areas will receive special attention, roughly in the specified order:

- Enhance the `mkfs` program to have a better interface (i.e. command line arguments);
- Design and test the system under some representative benchmarks; compare it with the regular `ext2` file system;
- Modify the `fsck` program to cope with `ext4` partitions;
- Write special allocation algorithms to benefit additional freedom offered by two partitions;
- Move the special files to the directories partition (probably they have a similar access pattern).

7 Performance

Even before making any measurements, we can anticipate that we will not get tremendous performance increase, and that ordinary benchmarks should not show remarkable advantages of the new design. This is due to the following factors:

- The implementation does yet not fully take advantage of the two-disked architecture; only when asynchronous access is in question (for the delayed writes and read-ahead) the device drivers themselves might provide some parallelism;
- The implementation does not yet attempt to reorganize the data layout on the disks to benefit the differing usage patterns of regular files and directories; the design is still very much indebted to the original `ext2` and FFS sources;
- The main benefits should be visible in the context of multithreaded applications (or in multi-user activity), which have simultaneous outstanding I/O requests. Typical applications (non-multithreaded) generally generate requests sequentially, first scanning directories, and next accessing file data. The parallelism will be thus exploitable only if differing request types are outstanding. The typical benchmarks, on the other hand, are mainly sequential.

I have tried to evaluate the design using the famous Andrew benchmark, but the results are not relevant, due to the low precision of the timing available to the shell. I have thus run the following simple benchmark:

```
cycles=5
date
mount
while [ $cycles -ge 0 ]; do
  time cp -R /usr/src /fs1
  time cp -R /usr/src /fs2
  time cp -R /usr/src /fs3
```

```
  time cp -R /fs2/src /fs2/src1
  time cp -R /fs3/src /fs3/src1

  time rm -rf /fs1/src
  time rm -rf /fs2/src
  time rm -rf /fs3/src
  time rm -rf /fs2/src1
  time rm -rf /fs3/src1
  cycles=`expr $cycles - 1`
done
```

There are about 150Mb of data moved on each copy. The machine has 64Mb of RAM (so cache is essential in timing), has a Pentium processor at 166MHz, and two drives, denoted by a and b. [Need some disk information.]

The following file systems are involved:

Mount directory	file system type	disk
`/usr`	`ext2`	a
`/fs1`	`ext2`	a
`/fs2`	`ext2`	b
`/fs3`	`ext4`	dir on a, files on b

The results of the benchmark are encouraging; these are some of the timings, in seconds. This is true elapsed time, on an otherwise idle system.

Disks	Average time	Standard deviation
Copy tests		
a $\rightarrow$ a	254.2	0.7
a $\rightarrow$ b	206.6	0.6
a $\rightarrow$ a+b	200.9	0.5
b $\rightarrow$ b	199.9	0.7
a+b $\rightarrow$ a+b	183.4	0.5
Remove tests		
a	67.8	0.2
b	67.4	0.2
a+b	55.2	0.5
b	70.1	0.9
a+b	61.5	1.1

7.1 Comments

- I attribute the high values of the times which work on the partition `/usr` to the fragmentation of that heavily used partition which causes a lot of remote seeks.

- The local (i.e. source and destination on the same partition) copy test indicate a `ext4` having a performance increase of of $254-183 = 70s = 27\%$ over the disk a, and $199-183 = 16s = 8\%$ over the disk b.

- The delete test gives better timing for files created on empty partitions, which shows the influence of fragmentation of the free space. Our file system outperforms with $67 - 55 = 12s = 18\%$ the other partitions for deleting non-fragmented files, and $70 - 61 = 9s = 11\%$ for fragmented files.
- The performance could possibly be improved by enhancing the allocation techniques.
- The performance is likely to be even better for multithreaded (parallel) access to files; this test is still sequential in nature, and only the cache read-ahead and write-behind give rise to parallel disk access.
- A smaller RAM cache might favor our scheme; large caching probably tends to flatten the performance curves of the file systems (I have no way of disabling or limiting the cache size).

8 Conclusions

The observed results are not very good considering the resources thrown into play. Apparently the "md" driver for RAID-0 gives more benefits for the same resources (i.e. two disks) (I have been told; I don't know exactly what is the improvement in performance offered by it). More complex tests and better allocation techniques may reduce somehow the gap.

The performance is still better than two file systems which use the same resources, at least for the tests I carried out (compare copies a $\rightarrow$ b with a+b $\rightarrow$ a+b).

The amount of enhancement observed in my tests is also partially a measure of the amount of asynchronous operations of the cache, because it is caused only by these operations. Maybe a careful re-read of the whole file system code would reveal additional places where parallelization could give benefits (although I don't expect to find many such places; maybe for the synchronous writes).

The results are not bad enough to discourage further enhancements; the design has some viable features, which aren't maybe yet exploited to their full capacity.

9 References

1. *A High Performance Multi-Structured File System Design*; Keith Muller and Joseph Pasquale — Proceedings of the Thirteenth ACM Symposium on Operating System Principles
2. *Linux File Systems*; Rémy Card, Theodore Ts'o, Stephen Tweedie — Slides
3. *File Management in the Linux Kernel*; Rémy Card — Slides
4. *Optimizations in File Systems*; Stephen Tweedie — Slides
5. *Linux Kernel Source Code and Linux Disk Utilities*; Remi Card, Theodore Ts'o, Linus Torvalds — Linux kernel 2.0.18
6. *A Fast Filesystem for UNIX*; McKusick, William Joy, Laffler, Fabry; UC Berkeley — ACM

7. *The Design and Implementation of a Log-Structured File System*; Rosenblum and Ousterhout; UC Berkeley — ACM Transactions on Computer Systems, 10(1), Feb 1992

8. *Weighted Voting for Replicated Data*; David K Gifford; Stanford U — Proceedings of 7th ACM SOSP, 1979

9. *The HP AutoRAID Hierarchical Storage System*; J Wilkes, R Golding, C Staelin, T Sullivan; HP — Proceedings of 15th ACM SOSP, 1995

10. *Improving the Write Performance of an NFS Server*; Chet Juszczak; DEC — USENIX Winter 1994 Technical Conference

11. *Not Quite NFS, Soft Cache Consistency for NFS*; Rick Macklem; U Guelph — USENIX Winter 1994 Technical Conference

12. *Dynamic Vnodes — Design and Implementation*; Aju John; Digital Equipment Corporation — USENIX 1995 Technical Conference

13. *Filesystem Logging versus Clustering: A Performance Comparison*; Margo Seltzer, Keith A Smith; Harvard, H Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padamanabhan; UCBerkeley — USENIX 1995 Technical Conference

14. *Metadata Logging in an NFS Server*; Uresh Vahalia, Dennis Ting; EMC, Gary G Gray Abilene U — USENIX 1995 Technical Conference

15. *Heuristic Cleaning Algorithms in Log-Structured Filesystems*; Trevor Blackwell, Jeffrey Harris, Margo Seltzer; Harvard U — USENIX 1995 Technical Conference

16. *Scalability in the XFS File System*; Adam Sweeney, Don Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto and Geoff Peck; Silicon Graphics — USENIX 1996 Technical Conference

17. *A Comparison of FFS Disk Allocation Policies*; Keith A. Smith and Margo Seltzer; Harvard University — USENIX 1996 Technical Conference

18. *AFRAID — A Frequently Redundant of Independent Disks*; Stefan Savage, University of Washington and John Wilkes; HP Labs — USENIX 1996 Technical Conference

19. *Distributed Operating Systems — chap. 5 Distributed Filesystems*; Andrew S Tanenbaum — Prentice Hall, 1995

20. *comp.os.research FAQ*; Brian O'Sullivan — comp.os.research