Introduction

The author is one of many OS researchers who are attempting to unify the name spaces in the operating system in varying ways [e.g. Pike ‘‘The Use of Name Spaces in Plan9’’]. None of us are well funded compared with the size of the task, and I am far from an exception to this rule. The natural consequence is that we each have attacked one small aspect of the task. My contribution is in incorporating small objects into the file system name space *effectively*.

This implementation offers value to the average Linux user, in that it offers generally good performance compared to the current Linux file system known as ext2fs.It also saves space to an extent that is important for some applications, and convenient for most. It does extremely well for large directories, and has a variety of minor advantages.

Since ext2fs is very similar to FFS and UFS in performance, the implementation also offers potential value to commercial OS vendors who desire greater than ext2fs performance without directory size issues, and who appreciate the value of a better foundation for integrating name spaces throughout the OS.

Content table | Next

Previous | Content table | Next

Why Is There A Move Among Some OS Designers Towards Unifying Name Spaces?

An operating system is composed of components that access other components through interfaces. Operating systems are complex enough that, like national economies, the architect cannot centrally plan the interactions of the components that it is composed of. The architect can provide a structural framework that has a marked impact on the efficiency and utility of those interactions. Economists have developed principles that govern large economic systems. Are there system principles that we might use to start a discussion of the ways increasing component interactivity via naming system design impacts the total utility of an operating system? I propose these:

- If one increases the number of other components that a particular component can interact with, one increases its expressive power and thereby its utility.
- One can increase the number of other components that a particular component can interact with either by increasing the number of interfaces it has, or by increasing the number of components that are accessible by its current interfaces.
- The cost of component interfaces dominates software design cost., like the cost of wires dominates circuit design cost.
- Total system utility tends to be proportional not to the number of components, but to the number of possible component interactions.

It is not simply the number of components that one has that determines an OS's expressive power, it is the number of opportunities to use them that determines it. The number of these opportunities are proportional to the number of possible combinations of them, and the number of possible combinations of them are determined by their connectedness. Component connectedness in OS design is determined by name space design, to much the same extent that buses determine it in circuit design.

Allow me to illustrate the impact of these principles with the use of an imaginary example. Suppose two imaginary OS vendors with equally talented programmers hire two different OS architects. Suppose one of the architects centers the OS design around a single name space design that allows all of the components to access all other components via a single interface (assume this is possible, it is a theoretical example). Suppose the other allows the ten different design groups in the company that are developing components to create their own ten name spaces. Suppose that the unified name space OS architect has half of the resources of the fragmented name space OS architect and creates half as many components. While the number of components is half as large, the number of connections is $1/2^2/((1/10^2)*10)$ times larger. If you accept my hypothesis that utility is more proportional to connections than components, then the unified operating system with half the development cost will still offer more expressive utility. That is a powerful motivation.

To return briefly to the long ago researched principles governing another member of the class of large systems, the economies of nations, it is perhaps interesting to note that Adam Smith in ``The Wealth of Nations'' engaged in substantial study of the link between the extent of interconnectedness and the development of civilization, where the extent of interconnectedness was determined by waterways, etc. The link he found for economic systems was no less crucial than what is being suggested here for the effect of component interconnectedness on the total utility of software systems. I suggest that I am merely generalizing a long established principle from another field of science, namely that total utility in large systems with components that interact to generate utility is determined by the extent of their interconnection.

There are many exceptions to these principles: not all chips on a motherboard sit on the bus, and analogous considerations apply to both OS design and the economies of nations. I hope the reader will accept that space considerations make it appropriate to gloss over these, and will consider the central point that under some circumstances unifying name spaces in a design can dramatically improve the utility of an OS. That can be an enormous motivation, and it has moved a number of OS researchers in their work [e.g. Pike ``The Use of Name Spaces in Plan9'' and ``The Hideous Name'' http://magnum.cooper.edu:9000/ ~rp/html/rob.html.

Unfortunately, it is not a small technical effort to combine name spaces. To combine 10 name spaces requires, if not the effort to create 10 name spaces, perhaps an effort equivalent to creating 5 of the name spaces. Usually each of the name spaces has particular performance and semantic power requirements that require enhancing the unified name space, and it usually requires technical innovation to combine the advantages of each of the separated name spaces into a unified name space. I would characterize none of the research groups currently approaching this unification problem as having funding equivalent to what went into creating 5 of the name spaces they would like to unify, and we are certainly no exception. For this reason I have picked one particular aspect of this larger problem for our focus: allowing small objects to effectively share the same file system interface that large objects use currently.

As operating systems increase the number of their components, the higher development cost of a file system able to handle small files becomes more worth the multiplicative effect it has on OS utility, as well as its reduction of OS component interface cost.

Previous | Content table | Next

Previous | Content table | Next

Should File Boundaries Be Block Aligned?

Making file boundaries block aligned has a number of effects: it minimizes the number of blocks a file is spread across (which is especially beneficial for multiple block files when locality of reference across files is poor), it wastes disk and buffer space in storing every less than fully packed block, it wastes I/O bandwidth with every access to a less than fully packed block when locality of reference is present, it increases the average number of block fetches required to access every file in a directory, and it results in simpler code.

The simpler code of block aligning file systems follows from not needing to create a layering to distinguish the units of the disk controller and buffering algorithms from the units of space allocation, and from not needing to optimize the packing of nodes as is done in balanced tree algorithms. For readers who have not been involved in balanced tree implementations, algorithms of this class are notorious for being much more work to implement than one would expect from their description. Sadly, they also appear to offer the highest performance solution for small files, once I remove certain simplifications from their implementation and add certain optimizations common to file system designs. I regret that code complexity (30k lines) is a major disadvantage of the approach compared to the 6k lines of the ext2fs approach.

I started our analysis of the problem with an assumption that I needed to aggregate small files in some way, and that the question was, which solution was optimal? The simplest solution was to aggregate all small files in a directory together into either a file or the directory. But any aggregation into a file or directory wastes part of the last block in the aggregation. What does one do if there are only a few small files in a directory, aggregate them into the parent of the directory? What if there are only a few small files in a directory at first, and then there are many small files, how do I decide what level to aggregate them at, and when to take them back from a parent of a directory and store them directly in the directory. As we did our analysis of these questions we realized that this problem was closely related to the balancing of nodes in a balanced tree. The balanced tree approach, by using an ordering of files which are then dynamically aggregated into nodes at a lower level, rather than a static aggregation or grouping, avoids this set of questions.

In my approach I store both files and filenames in a balanced tree, with small files, directory entries, inodes, and the tail ends of large files all being more efficiently packed as a result of relaxing the requirements of block alignment, and eliminating the use of a fixed space allocation for inodes. I have a sophisticated and flexible means for arranging for the aggregation of files for maximal locality of reference, through defining the ordering of items in the tree. The body of large files is stored in unformatted nodes that are attached to the tree but isolated from the effects of possible shifting by the balancing algorithms.

Approaches such as [Apple] and [Holton and Das] have stored filenames but not files in balanced trees. None of the file systems C-FFS, NTFS, or XFS aggregate files, all of them block align files, though all of those also do some variation on storing small files in the statically allocated block address fields of inodes if they are small enough to fit there.[C-FFS] has published an excellent discussion of both their approach and why small files rob a conventional file system of performance more in proportion to the number of small files than the number of bytes consumed by small files. However, I must note that their notion of what constitutes small is different from ours by one or two orders of magnitude. Their use of an exo-kernel is simply an excellent approach for operating systems that have that as an available option.

Semantics (files), packing (blocks/nodes), caching(read ahead sizes, etc.), and the hardware interfaces of disk (sectors) and paging (pages) all have different granularity issues associated with them: a central point of our approach is that the optimal granularity of these often differs, and abstracting these into separate layers in which the granularity of one layer does not unintentionally impact other layers can improve space/time performance. Reiserfs innovates in that its semantic layer often conveys to the other layers an ungranulated ordering rather than one granulated by file boundaries. The reader is encouraged to note the areas in which reiserfs needs to go farther in its doing so while reading the algorithms.

Previous | Content table | Next

Previous | Content table | Next

Balanced Trees and Large File I/O

There has long been an odd informal consensus that balanced trees are too slow for use in storing large files, perhaps originating in the performance of databases that have attempted to emulate file systems using balanced tree algorithms that were not originally architected for file system access patterns or their looser serialization requirements. It is hopefully easy for the reader to understand that storing many small files and tail ends of files in a single node where they can all be fetched in one I/O leads directly to higher performance. Unfortunately, it is quite complex to understand the interplay between I/O efficiency and block size for larger files, and space does not allow a systematic review of traditional approaches. The reader is referred to [FFS], [Peacock], [McVoy], [Holton and Das], [Bach], [OLE], and [NTFS] for treatments of the topic, and discussions of various means of 1) reducing the effect of block size on CPU efficiency, 2) eliminating the need for inserting rotational delay between successive blocks, 3) placing small files into either inodes or directories, and 4) performing read-ahead. More commentary on these is in the annotated bibliography.

Reiserfs has the following architectural weaknesses that stem directly from the overhead of repacking to save space and increase block size: 1) when the tail (files < 4k are all tail) of a file grows large enough to occupy an entire node by itself it is removed from the formatted node(s) it resides in, and it is converted into an unformatted node ([FFS] pays a similar conversion cost for fragments), 2) a tail that is smaller than one node may be spread across two nodes which requires more I/O to read if locality of reference is poor, 3) aggregating multiple tails into one node introduces separation of file body from tail, which reduces read performance ([FFS] has a similar problem, and for reiserfs files near the node in size the effect can be significant), 4) when you add one byte to a file or tail that is not the last item in a formatted node, then on average half of the whole node is shifted in memory. If any of your applications perform I/O in such a way that they generate many small unbuffered writes, reiserfs will make you pay a higher price for not being able to buffer the I/O. Most applications that create substantial file system load employ effective I/O buffering, often simply as a result of using the I/O functions in the standard C libraries.

By avoiding accesses in small blocks/extents reiserfs improves I/O efficiency. Extent based file systems such as VxFS, and write-clustering systems such as ext2fs, are not so effective in applying these techniques that they choose to use 512-byte blocks rather than 1k blocks as their defaults. Ext2fs reports a 20% speedup when 4k rather than 1k blocks are used, but the authors of ext2fs advise the use of 1k blocks to avoid wasting space.

There are a number of worthwhile large file optimizations that have not been added to either ext2fs or reiserfs, and both file systems are somewhat primitive in this regard, reiserfs being the more primitive of the two. Large files simply were not my research focus, and it being a small research project I did not implement the many well known techniques for enhancing large file I/O. The buffering algorithms are probably more crucial than any other component in large file I/O, and partly out of a desire for a fair comparison of the approaches I have not modified these. I have added no significant optimizations for large files, beyond increasing the block size, that are not found in ext2fs. Except for the size of the blocks, there is not a large inherent difference between: 1) the cost of adding a pointer to an unformatted node to my tree plus writing the node, and 2) adding an address field to an inode plus writing the block. It is likely that except for block size the primary determinants of high performance large file access are orthogonal to the decision of whether to use balanced tree algorithms for small and medium sized files.

For large files we get some advantage from not having our tree being more balanced than the tree formed by an inode which points to a triple indirect block. We haven't an easy method for measuring the performance gain from that though.

There is performance overhead due to the memory bandwidth cost of balancing nodes for small files. We think it is worth it though.

Previous | Content table | Next

Previous | Content table | Next

Serialization and Consistency

The issues of ensuring recoverability with minimal serialization and data displacement necessarily dominate high performance design. Let's define the two extremes in serialization so that the reason for this can be clear. Consider the relative speed of a set of I/O's in which every block request in the set is fed to the elevator algorithms of the kernel and the disk drive firmware fully serially, each request awaiting the completion of the previous request.; Now consider the other extreme, in which all block requests are fed to the elevator algorithms all together, so that they may all be sorted and performed in close to their sorted order (disk drive firmwares don't use a pure elevator algorithm). The unserialized extreme may be more than an order of magnitude faster, due to the cost of rotations and seeks. Unnecessarily serializing I/O prevents the elevator algorithm from doing its job of placing all of the I/O's in their layout sequence rather than chronological sequence. Most of high performance design centers around making I/O's in the order they are laid out on disk, and laying out blocks on disk in the order that the I/O's will want to be issued.

Snyder discusses a file system that obtains high performance from a complete lack of disk synchronization, but is only suitable for temporary files that don't need to survive reboot. I think its known value to Solaris users indicates that the optimal buffering policy varies from file to file.

Ganger discusses methods for using ordering of writes rather than serialization for ensuring conventional file system meta-data integrity, [McVoy] previously suggested but did not implement ordering of buffer writes.

Ext2fs is fast in substantial part due to avoiding synchronous writes of metadata, and I have much personal experience with it that leads me to prefer compiles that are fast. [I would like to see it adopt a policy that all dirty buffers for files not flagged as temporary are queued for writing, and that the existence of a dirty buffer means that the disk is busy. This will require replacing buffer I/O locking with copy-on-write, but an idle disk is such a terrible thing to waste.:-)]

[NTFS] by default adds unnecessary serialization to an extent that even older file systems such as [FFS] do not, and its performance characteristics reflect that. In fairness, it should be said that it is the superior approach for most removable media without software control of ejection (e.g. IBM PC floppies).

Reiserfs employs a new scheme called preserve lists for ensuring recoverability, which avoids overwriting old meta-data by writing the meta-data nearby rather than over old meta-data.

Previous | Content table | Next

Why Aggregate Small Objects at the File System Level?

There has long been a tradition of file system developers deciding that effective handling of small files is not significant to performance, and the application programmers caring enough about performance for small files to not store them as separate entities in the file system. To store small objects one may either make the file system efficient for the task, or sidestep the problem by aggregating small objects in a layer above the file system. Sidestepping the problem has three disadvantages: utility, code complexity, and performance.

Utility and Code Complexity: Allowing OS designers to effectively use a single namespace with a single interface for both large and small objects decreases coding cost and increase expressive power of components throughout the OS. I feel reiserfs shows the effects of a larger development investment focused on a simpler interface when compared with many solutions for this currently available in the object oriented toolkit community, such as the Structured Storage available in Microsoft's [OLE]. By simpler I mean I added nothing to the file system API to distinguish large and small objects, and I leave it to the directory semantics and archiving programs to aggregate objects. Multiple layers cost more to implement, cost more to code the interfaces for utilizing, and provide less flexibility.

Performance: It is most commonly the case that when one layers one file system on top of another the performance is substantially reduced, and Structured Storage is not an exception to this general rule. Reiserfs, which does not attempt to delegate the small object problem to a layer above, avoids this performance loss. I have heard it suggested by some that this layering avoids the performance loss from syncing on file close as many file systems do. I suggest that this is adding an error to an error rather than fixing it.

Let me make clear that I believe those who write such layers above the file system do not do so out of stupidity. I know of at least one company at which a solution that layers small object storage above the file system exists because the file system developers refused to listen to the non-file system group's description of its needs, and the file system group had to be sidestepped in generating the solution.

Current file systems are fairly well designed for the purposes that their users currently use them for: my goal is to change file size usage patterns. The author remembers arguments that once showed clearly that there was no substantial market need for disk drives larger than 10MB based on current usage statistics. While [C-FFS] points out that 80% of file accesses are to files below 10k, I do not believe it reasonable to attempt to provide statistics based on usage measurements of file systems for which small files are inappropriate to use that will show that small files are frequently used. Application programmers are smarter than that. Currently 80% of file accesses are to the first order of magnitude in file size for which it is currently sensible to store the object in the file system. I regret that one can only speculate as to whether once file systems become effective for small files and database tasks, usage patterns will change to 80% of file accesses being to files less than 100 bytes. What I can do is show via the 80/20 Banded File Set Benchmark presented later that in such circumstances small file performance potentially dominates total system performance.

In summary, the on-going reinvention of incompatible object aggregation techniques above the file system layer is expensive, less expressive, less integrated, slower, and less efficient in its storage than incorporating balanced tree algorithms into the file system.

Previous | Content table | Next

Previous | Content table | Next

Tree Definitions

Balanced trees are used in databases, and more generally, wherever a programmer needs to search and store to non-random memory by a key, and has the time to code it this way.

The usual evolution for programmers is to first think that hashing will be simpler and more efficient, and then realize only after getting into the sordid details of it that the combination of space efficiency, minimizing disk accesses, and the feasibility of caching tho top part of the tree, makes the tree approach more effective. It is the usual thing to first try to do hashing, and then by the time the details are worked out, to have a balanced tree. The cost of effectively handling bucket overflow just isn't less than the cost of balancing, unless the buchets are always all in RAM. Hashing is often a good solution when there is no non-random memory involved, such as when hashing a cache. The Linux dcache code uses hashing for accessing a cache of in-memory directory entries. Sometimes one uses partial or full hashing of keys within that balanced tree. If you do full hashing within a tree, and you cache the top part of that tree, you have something rather similar to extensible hashing, except it is more flexible and efficient.

Sometimes programmers code using unbalanced trees. Most filesystems do essentially that. Balanced trees generally do a better job of minimizing the average number of disk accesses. There is literature establishing that balanced trees are optimal for the worst case when there is no caching of the tree. This is rather pointless literature, as the average case when cached is what is important, and I am afraid that the existing literature proves that which is feasible to prove rather than that which is relevant. That said, practitioners know from experience that making the tree less balanced leads to more I/Os. Discussions of the exceptions to this are rather interesting but not for here....

I regret that I must assume that the reader is familiar with basic balanced tree algorithms [Wood], [Lewis and Denenberg], [Knuth], [McCreight]. No attempt will be made to survey tree design here since balanced trees are one of the most researched and complex topics in algorithm theory, and require treatment at length. I must compound this discourtesy with a concise set of definitions that sorely lack accompanying diagrams, my apologies. Finally, I'll truly annoy the reader by saying that the header files contain nice ascii art, and if you want full definition of the structures, the source is the place.

Classically, balanced trees are designed with the set of keys assumed to be defined by the application, and the purpose of the tree design is to optimize searching through those keys. In my approach the purpose of the tree is to optimize the reference locality and space efficient packing of objects, and the keys are defined as best optimizes the algorithm for that. Keys are used in place of inode numbers in the file system, thereby choosing to substitute a mapping of keys to node location (the internal nodes) for a mapping of inode number to file location. Keys are longer than inode numbers, but one needs to cache fewer of them than one would need to cache inode numbers when more than one file is stored in a node.

In my tree, I still require that a filename be resolved one component at a time. It is an interesting topic for future research whether this is necessary or optimal. This is more complex of an issue than a casual reader might realize: directory at a time lookup accomplishes a form of compression, makes mounting other name spaces and file system extensions simpler, makes security simpler, and makes future enhanced semantics simpler. Since small files typically lead to large directories, it is fortuitous that as a natural consequence of our use of tree algorithms, our directory mechanisms are much more effective for very large directories than most other file systems are (notable exceptions include [Holton and Das]).

The tree has three node types: internal nodes, formatted nodes, and unformatted nodes.

The contents of internal and formatted nodes are sorted in the order of their keys. (Unformatted nodes contain no keys.)

Internal nodes consist of pointers to sub-trees separated by their delimiting keys. The key that precedes a pointer to a sub-tree is a duplicate of the first key in the first formatted node of that sub-tree. Internal nodes exist solely to allow determining which formatted node contains the item corresponding to a key. Reiserfs starts at the root node, examines its contents, and based on it can determine which subtree contains the item corresponding to the desired key. From the root node reiserfs descends into the tree, branching at each node, until it reaches the formatted node containing the desired item.

The first (bottom) level of the tree consists of unformatted nodes, the second level consists of formatted nodes, and all levels above consist of internal nodes. The highest level contains the root node. The number of levels is increased as needed by adding a new root node at the top of the tree.

All paths from the root of the tree to all formatted leaves are equal in length, and all paths to all unformatted leaves are also equal in length and 1 node longer than the paths to the formatted leaves. This equality in path length, and the high fanout it provides is vital to high performance, and in the Drops section I will describe how the lengthening of the path length that occurred as a result of introducing the [BLOB] approach (the use of indirect items and unformatted nodes) proved a measurable mistake.

Formatted nodes consist of items. Items have four types: direct items, indirect items, directory items, and stat data items. All items contain a key which is unique to the item. This key is used to sort, and find, the item.

Direct items contain the tails of files, and tails are the last part of the file (the last file_size modulo FS block size of a file).

Indirect items consist of pointers to unformatted nodes. All but the tail of the file is contained in its unformatted nodes.

Directory items contain the key of the first directory entry in the item followed by a number of directory entries.

Depending on the configuration of reiserfs, stat data may be stored as a separate item, or it may be embedded in a directory entry. We are still benchmarking to determine which way is best.

A file consists of a set of indirect items followed by a set of up to two direct items, with the existence of two direct items representing the case when a tail is split across two nodes. If a tail is larger than the maximum size of a file that can fit into a formatted node but is smaller than the unformatted node size (4k), then it is stored in an unformatted node, and a pointer to it plus a count of the space used is stored in an indirect item.

Directories consist of a set of directory items. Directory items consist of a set of directory entries. Directory entries contain the filename and the key of the file which is named. There is never more than one item of the same item type from the same object stored in a single node (there is no reason one would want to use two separate items rather than combining).

The first item of a file or directory contains its stat data.

When performing balancing, and analyzing the packing of the node and its two neighbors, we ensure that the three nodes cannot be compressed into two nodes. I feel greater compression than this is best left to an FS cleaner to perform rather than attempting it dynamically.

Reiserfs structures

Reiserfs Tree has Max_Height = N (current default value for N = 5):
The tree lais in the disk blocks.
Each disk blocks that belongs the reiserfs tree has Block Head

The disk Block   (Internal Node of the tree is the place for keys and pointers to disk blocks)

Block_Head	Key 0	Key 1	Key 2	---	Key N	Pointer 0	Pointer 1	Pointer 2	---	Pointer N	Pointer N+1	........Free Space......

The disk Block   (Leaf Node of the tree is the place for the Items and Items headers)

Block_Head	IHead 0	IHead 1	IHead 2	---	IHead N	.....................Free Space..................	Item N	---	Item 2	Item 1	Item 0

The disk Block (Unformatted Node   of the tree is the place for the data of the big file)

....

Reiserfs objects: Files, Directories
Max number of objects = 2^32-4 = 4 294 967 292

Each object is a number of items :
Files items :
1. StatData item + [Direct item]   (for small files : size from 0 bytes to

MAX_DIRECT_ITEM_LEN=blocksize-112 bytes)
2. StatData item + InDirect item + [Direct item]   (for big files   : size   >
MAX_DIRECT_ITEM_LEN bytes)

Directory items :
1. StatData item + Directory item

Every reiserfs object has Object ID and Key .

Internal Node structures

The disk Block   (Internal Node of the tree is the place for keys and pointers to disk blocks)

Block_Head	Key 0	Key 1	Key 2	---	Key N	Pointer 0	Pointer 1	Pointer 2	---	Pointer N	Pointer N+1	........Free Space......

struct block_head

Field Name	**Type**	Size (in bytes)	Description
blk_level	unsigned short	2	Level of block in the tree (1-leaf; 2,3,4,... - internal;
blk_nr_item	unsigned short	2	Number of Keys in an Internal block. Or Number of Items in a Leaf block.
blk_free_space	unsigned short	2	Block Free Space in bytes
blk_right_delim_key	struct key	16	Right delimiting key for this block (for Leaf nodes only)
	total	6 or 22	(6) 8 bytes for internal nodes ; (22) 24 bytes for leaf nodes

struct key

Field Name	**Type**	**Size (in bytes)**	**Description**
k_dir_id	__u32	4	ID of the parent directory
k_object_id	__u32	4	ID of the object (also it is the number of inode)
k_offset	__u32	4	Offset from beginning of the object to the current byte of the object
k_uniqueness	__u32	4	Type of the item (StatData = 0, Direct = -1, InDirect = -2, Directory = 500)
	total	16	16 bytes

struct disk_child (Pointer to disk block)

Field Name	Type	Size (in bytes)	Description
dc_block_number	unsigned long	4	Disk child's block number.
dc_size	unsigned short	2	Disk child's used space.
	total	6	(6) 8 bytes

Leaf Node structures

The disk Block (Leaf Node of the tree is the place for the Items and Items headers)

Block_Head	IHead 0	IHead 1	IHead 2	---	IHead N	.....................Free Space..................	Item N	---	Item 2	Item 1	Item 0

struct block_head

Field Name	Type	Size (in bytes)	Description
blk_level	unsigned short	2	Level of block in the tree (1-leaf; 2,3,4,... - internal;
blk_nr_item	unsigned short	2	Number of Keys in an Internal block. Or Number of Items in a Leaf block.
blk_free_space	unsigned short	2	Block Free Space in bytes
blk_right_delim_key	struct key	16	Right delimiting key for this block (for Leaf nodes only)
	total	22	(22) 24 bytes for leaf nodes

Everything in the filesystem is stored as a set of items. Each item has its item_head. The item_head contains the key of the item, its free space (for indirect items) and specifies the location of the item itself within the block.

struct item_head (IHead)

Field Name	Type	Size (in bytes)	Description
ih_key	struct key	16	Key to search the item. All item headers is sorted by this key
u.ih_free_space u.ih_entry_count	__u16	2	Free space in the last unformatted node for an InDirect item; 0xFFFF for a Direct item ; 0xFFFF for a Stat Data item. The number of directory entries for a Directory item.
ih_item_len	__u16	2	total size of the item body
ih_item_location	__u16	2	an offset to the item body within the block
ih_reserved	__u16	2	used by reiserfsck
	total	24	24 bytes

There are 4 types of items: **stat_data item**, **directory item**, **indirect item**, **direct item.**

struct stat_data (reiserfs version of UFS disk inode minus the address blocks)

Field Name	Type	Size (in bytes)	Description
sd_mode	__u16	2	file type, permissions
sd_nlink	__u16	2	number of hard links
sd_uid	__u16	2	owner id
sd_gid	__u16	2	group id
sd_size	__u32	4	file size
sd_atime	__u32	4	time of last access
sd_mtime	__u32	4	time file was last modified
sd_ctime	__u32	4	time inode (stat data) was last changed (except changes to sd_atime and sd_mtime)
sd_rdev	__u32	4	device
sd_first_direct_byte	__u32	4	Offset from the beginning of the file to the first byte of direct item of the file. (-1) for directory (1) for small files (file has direct items only) (>1) for big files (file has indirect and direct items) (-1) for big files (file has indirect, but has not direct item)
	total	32	32 bytes

Directory item :

deHead 0	deHead 1	deHead 2	---	deHead N	**fileName N**	**---**	**fileName 2**	**fileName 1**	**fileName 0**

Direct item :

........................Small File Body............................

InDirect item :

unfPointer 0	unfPointer 1	unfPointer 2	---	unfPointer N

unfPointer - pointer to unformatted block (unfPointer size = 4 bytes).
Unfomatted blocks contain the body of a big file.

struct reiserfs_de_head (deHead)

Field Name	Type	Size (in bytes)	Description
deh_offset	__u32	4	third component of the directory entry key (all reiserfs_de_head sorted by this value)
deh_dir_id	__u32	4	objectid of the parent directory of the object, that is referenced by directory entry
deh_objectid	__u32	4	objectid of the object, that is referenced by directory entry
deh_location	__u16	2	offset of name in the whole item
deh_state	__u16	2	1) entry contains stat data (for future) 2) entry is hidden (unlinked)
	total	16	16 bytes

fileName - the name of the file (array of bytes of variable length).
Max length of file name = blocksize - 64 (for 4kb blocksize Max name length = 4032 bytes).

Previous | Content table | Next

Previous | Content table | Next

Using the Tree to Optimize Layout of Files

There are four levels at which layout optimization is performed: 1) the mapping of logical block numbers to physical locations on disk 2) the assigning of nodes to logical block numbers, 3) the ordering of objects within the tree, and 4) the balancing of the objects across the nodes they are packed into.

Previous | Content table | Next

Previous | Content table | Next

Physical Layout

This is performed by the disk drive manufacturer for SCSI, for IDE drives this logical block numbers to physical location mapping is done by the device driver, and for all drives it is also potentially done by volume management software. The logical block number to physical location mapping by the drive manufacturer is usually done using cylinders. I agree with the authors of [ext2fs] and most others that the significant file placement feature for FFS was not the actual cylinder boundaries, but placing files and their inodes on the basis of their parent directory location. FFS used explicit knowledge of actual cylinder boundaries in its design. I find that minimizing the distance in logical blocks of semantically adjacent nodes without tracking cylinder boundaries accomplishes an excellent approximation of optimizing according to actual cylinder boundaries, and I find its simplicity an aid to implementation elegance.

Previous | Content table | Next

Previous | Content table | Next

Node Layout

When I place nodes of the tree on the disk, I search for the first empty block in the bitmap (of used block numbers) which I will find if I start at the location of the left neighbor of the node in the tree ordering, and move in the direction I last moved in.. This was experimentally found to be better than the following alternatives for the benchmarks employed: 1) taking the first non-zero entry in the bitmap, 2) taking the entry after the last one that was assigned in the direction last moved in (this was 3% faster for writes and 10-20% slower for subsequent reads), 3) starting at the left neighbor and moving in the direction of the right neighbor. When changing block numbers for the purpose of avoiding overwriting sending nodes before shifted items reach disk in their new recipient node (see description of preserve lists later in paper), the benchmarks employed were ~10% faster when starting the search from the left neighbor rather than the node's current block number, even though it adds significant overhead to determine the left neighbor (the current implementation risks I/O to read the parent of the left neighbor). It used to be that we would reverse direction when we reached the end of the disk drive. Fortunately we checked to see if it makes a difference which direction one moves in when allocating blocks to a file, and indeed we found it made a significant difference to always allocate in the increasing block number direction. We hypothesize that this is due to matching disk spin direction by allocating using increasing block numbers.

Previous | Content table | Next

Previous | Content table | Next

Ordering within the Tree

While I give here an example of how I have defined keys to optimize locality of reference and packing efficiency, I would like to stress that key definition is a powerful and flexible tool that I am far from finished experimenting with. Some key definition decisions depend very much on usage patterns, and this means that someday one will select from several key definitions when creating the file system. For example, consider the decision of whether to pack all directory entries together at the front of the file system, or to pack the entries near the files they name. For large file usage patterns one should pack all directory items together, since systems with such usage patterns are effective in caching the entries for all directories. For small files the name should be near the file. Similarly, for large files the stat data should be stored separately from the body, either with the other stat data from the same directory, or with the directory entry. (It was likely a mistake for me to not assign stat data its own key in the current implementation, as packing it in with direct and indirect items complicates our code for handling those items, and prevents me from easily experimenting with the effects of changing its key assignment.)

It is not necessary for a file's packing to reflect its name, that is merely my default. With each file my next release will offer the option of overriding the default by use of a system call. It is feasible to pack an object completely independently of its semantics using these algorithms, and I predict that there will be many applications, perhaps even most, for which a packing different than that determined by object names is more appropriate.

Currently the mandatory tying of packing locality and semantics results in the distortion of both semantics and packing from what might otherwise be their independent optimums, much as tying block boundaries to file boundaries distorts I/O and space allocation algorithms from their separate optimums. For example, placing most files accessed while booting in their access order at the start of the disk is a very tempting future optimization that the use of packing localities makes feasible to consider.

The Structure of a Key: Each file item has a key with structure <locality_id, object_id, offset, uniqueness>. The locality_id is by default the object_id of the parent directory. The object_id is the unique id of the file, and this is set to the first unused objectid when the object is created. The tendency of this to result in successive object creations in a directory being adjacently packed is often fortuitous for many usage patterns. For files the offset is the offset within the logical object of the first byte of the item. In version 0.2 all directory entries had their own individual keys stored with them and were each distinct items, in the current version I store one key in the item which is the key of the first entry, and compute each entry's key as needed from the one key stored in the item. For directories the offset key component is the first four bytes of the filename, which you may think of as the lexicographic rather than numeric offset. For directory items the uniqueness field differentiates filename entries identical in the first 4 bytes. For all item types it indicates the item type and for the leftmost item in a buffer it indicates whether the preceding item in the tree is of the same type and object as this item. Placing this information in the key is useful when analyzing balancing conditions, but increases key length for non-directory items, and is a questionable architectural feature.

Every file has a unique objectid, but this cannot be used for finding the object, only keys are used for that. Objectids merely ensure that keys are unique. If you never use the reiserfs features that change an object's key then it is immutable, otherwise it is mutable. (This feature aids support for NFS daemons, etc.) We spent quite some time debating internally whether the use of mutable keys for identifying an object had deleterious long term architectural consequences: in the end I decided it was acceptable iff we require any object recording a key to possess a method for updating its copy of it. This is the architectural price of avoiding caching a map of objectid to location that might have very poor locality of reference due to objectids not changing with object semantics. I pack an object with the packing locality of the directory it was first created in unless the key is explicitly changed. It remains packed there even if it is unlinked from the directory. I do not move it from the locality it was created in without an explicit request, unlike the [C-FFS] approach which stores all multiple link files together and pays the cost of moving them from their original locations when the second link occurs. I think a file linked with multiple directories might as well get at least the locality reference benefits of one of those directories.

In summary, this approach 1) places files from the same directory together, 2) places directory entries from the same directory together with each other and with the stat data for the directory. Note that there is no interleaving of objects from different directories in the ordering at all, and that all directory entries from the same directory are contiguous. You'll note that this does not accomplish packing the files of small directories with common parents together, and does not employ the full partial ordering in determining the linear ordering, it merely uses parent directory information. I feel the proper place for employing full tree structure knowledge is in the implementation of an FS cleaner, not in the dynamic algorithms.

Previous | Content table | Next

Previous | Content table | Next

Node Balancing Optimizations

When balancing nodes I do so according to the following ordered priorities:

1. minimize number of nodes used
2. minimize number of nodes affected by the balancing operation
3. minimize the number of uncached nodes affected by the balancing operation
4. if shifting to another formatted node is necessary, maximize the bytes shifted

Priority 4) is based on the assumption that the location of an insertion of bytes into the tree is an indication of the likely future location of an insertion, and that policy 4 will on average reduce the number of formatted nodes affected by future balance operations. There are more subtle effects as well, in that if one randomly places nodes next to each other, and one has a choice between those nodes being mostly moderately efficiently packed or packed to an extreme of either well or poorly packed, one is more likely to be able to combine more of the nodes if one chooses the policy of extremism. Extremism is a virtue in space efficient node packing. The maximal shift policy is not applied to internal nodes, as extremism is not a virtue in time efficient internal node balancing.

Previous | Content table | Next

Previous | Content table | Next

Drops (the difficult design issues

in the current version that our next version can do better)

Consider dividing a file or directory into `drops`, with each drop having a separate key, and no two drops from one file or directory occupying the same node without being compressed into one drop. The key for each drop is set to the key for the object (file or directory) plus the offset of the drop within the object. For directories the offset is lexicographic and by filename, for files it is numeric and in bytes. In the course of several file system versions we have experimented with and implemented solid, liquid, and air drops. Solid drops were never shifted, and drops would only solidify when they occupied the entirety of a formatted node. Liquid drops are shifted in such a way that any liquid drop which spans a node fully occupies the space in its node. Like a physical liquid it is shiftable but not compressible. Air drops merely meet the balancing condition of the tree.

Reiserfs 0.2 implemented solid drops for all but the tail of files. If a file was at least one node in size it would align the start of the file with the start of a node, block aligning the file. This block alignment of the start of multi-drop files was a design error that wasted space: even if the locality of reference is so poor as to make one not want to read parts of semantically adjacent files, if the nodes are near to each other then the cost of reading an extra block is thoroughly dwarfed by the cost of the seek and rotation to reach the first node of the file. As a result the block alignment saves little in time, though it costs significant space for 4-20k files.

Reiserfs with block alignment of multi-drop files and no indirect items experienced the following rather interesting behavior that was partially responsible for making it only 88% space efficient for files that averaged 13k (the linux kernel) in size. When the tail of a larger than 4k file was followed in the tree ordering by another file larger than 4k, since the drop before was solid and aligned, and the drop afterwards was solid and aligned, no matter what size the tail was, it occupied an entire node.

In the current version we place all but the tail of large files into a level of the tree reserved for full unformatted nodes, and create indirect items in the formatted nodes which point to the unformatted nodes. This is known in the database literature as the [BLOB] approach. This extra level added to the tree comes at the cost of making the tree less balanced (I consider the unformatted nodes pointed to as part of the tree) and increasing the maximal depth of the tree by 1. For medium sized files, the use of indirect items increases the cost of caching pointers by mixing data with them. The reduction in fanout often causes the read algorithms to fetch only one node at a time of the file being read more frequently, as one waits to read the uncached indirect item before reading the node with the file data. There are more parents per file read with the use of indirect items than with internal nodes, as a direct result of reduced fanout due to mixing tails and indirect items in the node. The most serious flaw is that these reads of various nodes necessary to the reading of the file have additional rotations and seeks compared to the case with drops. With my initial drop approach they are usually sequential in their disk layout, even the tail, and the internal node parent points to all of them in such a way that all of them that are contained by that parent or another internal node in cache can be requested at once in one sequential read. Non-sequential reads of nodes are more than an order of magnitude more costly than sequential reads, and this single consideration dominates effective read optimization.

Unformatted nodes make file system recovery faster and less robust, in that one reads their indirect item rather than them to insert them into the recovered tree, and one cannot read them to confirm that their contents are from the file that an indirect item says they are from. In this they make reiserfs similar to an inode based system without logging.

A moderately better solution would have been to have simply eliminated the requirement for placement of the start of multi-node files at the start of nodes, rather than introducing BLOBs, and to have depended on the use of a file system cleaner to optimally pack the 80% of files that don't move frequently using algorithms that move even solid drops. Yet that still leaves the problem of formatted nodes not being efficient for mmap() purposes (one must copy them before writing rather than merely modifying their page table entries, and memory bandwidth is expensive even if CPU is cheap.)

For this reason I have the following plan for the next version. I will have three trees: one tree maps keys to unformatted nodes, one tree maps keys to formatted nodes, and one tree maps keys to directory entries and stat_data. Now it is only natural if you are thinking that that would mean that to read a file and access first the directory entry and stat_data, then the unformatted node, then the tail, one must hop long distances across the disk, going to first one tree and then the other This is indeed why it took me two years to realize it could be made to work. My plan is to interleave the nodes of the three trees according to the following algorithm:

Block numbers are assigned to nodes when the nodes are created, or preserved, and someday will be assigned when the cleaner runs. The choice of block number is based on first determining what other node it should be placed near, and then finding the nearest free block that can be found in the elevator's current direction. Currently we use the left neighbor of the node in the tree as the node it should be placed near. This is nice and simple. Oh well. Time to create a virtual neighbor layer.

The new scheme will continue to first determine the node it should be placed near, and then start the search for an empty block from that spot, but it will use a more complicated determination of what node to place it near. This method will cause all nodes from the same packing locality to be near each other, will cause all directory entries and stat_data to be grouped together within that packing locality, and will interleaved formatted and unformatted nodes from the same packing locality. Pseudo-code is best for describing this:

```
/* for use by reiserfs_get_new_blocknrs when determining where in the bitmap to
start the search for a free block, and for use by read-ahead algorithm when
there are not enough nodes to the right and in the same packing locality for
packing locality reading ahead purposes */
get_logical_layout_left_neighbors_blocknr(key of current node)
{
/* Based on examination of current node key and type, find the virtual neighbor of that node. */
   If body node
      if first body node of file
         if (node in tail tree whose key is less but is in same packing locality exists)
            return blocknr of such node with largest key
         else
            find node with largest key less than key of current node in stat_data tree
               return its blocknr
      else
         return blocknr of node in body tree with largest key less than key
of current node
```

```
   else
      if tail node
         if (node in body tree belonging to same file as first tail of current node exists)
            return its blocknr
         else if (node in tail tree with lesser delimiting key but same packing locality exists)
               return  blocknr of such node with largest delimiting key
         else
            return blocknr of node with largest key less than key of current node in stat_data tree
   else /* is stat_data tree node */
      if stat_data node with lesser key from same packing locality exists
         return blocknr of such node with largest key
      else /* no node from same packing locality with lesser key exists */
}

/* for use by packing locality read-ahead */
get_logical_layout_right_neighbors_blocknr(key of current node)
{
   right-handed version of get_logical_layout_left_neighbors_blocknr logic
}
```

It is my hope that this will improve caching of pointers to unformatted nodes, plus improving caching of directory entries and stat_data, by separating them from file bodies to a greater extent. I also hope that it will improve read performance for 1-10k files, and that it will allow us to do this without decreasing space efficiency.

Previous | Content table | Next

Previous | Content table | Next

Code Complexity

I thought it appropriate to mention some of the notable effects of simple design decisions on our implementation's code length. When we changed our balancing algorithms to shift parts of items rather than only whole items, so as to pack nodes tighter, this had an impact on code complexity. Another multiplicative determinant of balancing code complexity was the number of item types, and introducing indirect items doubled this, and changing directory items from being liquid drops to being air drops also increased it. Storing stat data in the first direct or indirect item of the file complicated the code for processing those items more than if I had made stat data its own item type.

When one finds oneself with an NxN coding complexity issue, it usually indicates the need for adding a layer of abstraction. The NxN effect of the number of items on balancing code complexity is an instance of that design principle, and we will address it in the next major rewrite. The balancing code will employ a set of item operations which all item types must support. The balancing code will then invoke those operations without caring to understand any more of the meaning of an item's type than that it determines which item specific item operation handler is called. Adding a new item type, say a compressed item, will then merely require writing a set of item operations for that item rather than requiring modifying most parts of the balancing code as it does now.

We now feel that the function to determine what resources are needed to perform a balancing operation, fix_nodes(), might as well be written to decide what operations will be performed during balancing since it pretty much has to do so anyway. That way, the function that performs the balancing with the nodes locked, do_balance(), can be gutted of most of its complexity.

Previous | Content table | Next

Previous | Content table | Next

Buffering & the Preserve List

We implemented for version 0.2 of our file system a system of write ordering that tracked all shifting of items in the tree, and ensured that no node that had had an item shifted from it was written before the node that had received the item was written. This is necessary to prevent a system crash from causing the loss of an item that might not be recently created. This tracking approach worked, and the overhead it imposed was not measurable in our benchmarks. When in the next version we changed to partially shifting items and increased the number of item types, this code grew out of control in its complexity. I decided to replace it with a far simpler to code scheme that was also more effective in typical usage patterns. This scheme was as follows:

If an item is shifted from a node, change the block that its buffer will be written to. Change it to the nearest free block to the old blocks left neighbor, and rather than freeing it, place the old block number on a ‘‘preserve list’’. (Saying nearest is slightly simplistic, in that the blocknr assignment function moves from the left neighbor in the direction of increasing block numbers.) When a ‘‘moment of consistency’’ is achieved, free all of the blocks on the preserve list. A moment of consistency occurs when there are no nodes in memory into which objects have been shifted (this could be made more precise but then it would be more complex). If disk space runs out, force a moment of consistency to occur. This is sufficient to ensure that the file system is recoverable. Note that during the large file benchmarks the preserve list was freed several times in the middle of the benchmark. The percentage of buffers preserved is small in practice except during deletes, and one can arrange for moments of consistency to occur as frequently as one wants to.

Note that I make no claim that this approach is better than the Soft Updates approach employed by [Granger] or by us in version 0.2, I merely note that tracking order of writes is more complex than this approach for balanced trees which partially shift items. We may go back to the old approach some day, though not to the code that I threw out.

Preserve lists substantially hamper performance for files in the 1-10k size range. We are re-evaluating them.

Ext2fs avoids the metadata shifting problem by never shrinking directories, and using fixed inode space allocations.

Previous | Content table | Next

Lessons From Log Structured File Systems

Many techniques from other file systems haven't been applied primarily so as to satisfy my goal of giving reiserfs 1.0 only the minimum feature set necessary to be useful, and will appear in later releases. Log Structured File Systems [Rosenblum and Ousterhout] embody several such techniques, which I will describe after I mention two concerns with that approach:

- With small object file systems it is not feasible to cache in RAM a map of objectid to location for every object since there are too many objects. This is an inherent problem in using temporal packing rather than semantic packing for small object file systems. With my approach my internal nodes are the equivalent of this objectid to location map, but internal node total size is proportional to the number of nodes rather than the number of objects. You can think of internal nodes as a compression of object location information made effective by the existence of an ordering function, and this compression is both essential for small files, and a major feature of my approach.
- I like obtaining good though not ideal semantic locality without paying a cleaning cost for active data. This is a less critical concern.

I frequently find myself classifying packing and layout optimizations as either appropriate for implementing dynamically or appropriate only for a cleaner. Optimizations whose computational overhead is large compared to their benefit tend to be appropriate for implementation in a cleaner, and a cleaner's benefits mostly impact the static portion of the file system (which typically consumes ~80% of the space.) Such objectives as 100% packing efficiency, exactly ordering block layout by semantic order, using the full semantic tree rather than parent directory in determining semantic order, compression, these are all best implemented by cleaner approaches. In summary, there is much to be learned from the LFS approach, and as I move past my initial objective of supplying a minimal feature higher performance FS I will apply some of those lessons.

In the Preserve Lists section I speculate on the possibilities for a fastboot implementation that would merge the better features of preserve lists and logging.

Previous | Content table | Next

Directions For the Future

To go one more order of magnitude smaller in file size will require adding functionality to the file system API, though it will not require discarding upward compatibility. The use of an exokernel is a better approach to small files if it is an option available to the OS designer, it is not currently an option for Linux users. In the future reiserfs will add such features as lightweight files in which stat_data other than size is inherited from a parent if it is not created individually for the file, an API for reading and writing to files without requiring the overhead of file handles and open(), set-theoretic semantics, and many other features that you would expect from researchers who expect to be able to do all that they could do in a database, in the file system, and never really did understand why not.

Previous | Content table | Next

Previous | Content table | Next

Conclusion

Balanced tree file systems are inherently more space efficient than block allocation based file systems, with the differences reaching order of magnitude levels for small files. While other aspects of design will typically have a greater impact on performance for large files, in direct proportion to the smallness of the file the use of balanced trees offers performance advantages. A moderate advantage was found for large files. Coding cost is mostly in the interfaces, and it is a measure of the OS designer’s skill whether those costs are low in the OS. We make it possible for an OS designer to use the same interface for large and small objects, and thereby reduce interface coding cost. This approach is a new tool available to the OS designer for increasing the expressive power of all of the components in the OS through better name space integration. Researchers interested in collaborating or just using my work will find me friendly. I tailor the framework of my collaborations to the needs of those we work with. I GPL reiserfs so as to meet the needs of academic collaborators. While that makes it unusable without a special license for commercial OSes, commercial vendors will find me friendly in setting up a commercial framework for commercial collaboration with commercial needs provided for.

Previous | Content table | Next

Previous | Content table | Next

Acknowledgments

Hans Reiser was the project initiator, primary architect, supplier of funding, and one of the programmers. Some folks at times remark that naming the filesystem Reiserfs was egotistic. It was so named after a potential investor hired all of my employees away from me, then tried to negotiate better terms for his possible investment, and suggested that he could arrange for 100 researchers to swear in Russian Court that I had had nothing to do with this project. That business partnership did not work out.

Vladimir Saveljev, while he did not author this paper, worked long hours writing the largest fraction of the lines of code in the file system, and is remarkably gifted at just making things work. Thanks Vladimir. Anatoly Pinchuk wrote much of the core balancing code, and too much of the rest to list here. Thanks, Anatoly. It is the policy of the Naming System Venture that if someone quits before project completion, and then takes strong steps to try to prevent others from finishing the project, that they shall not be mentioned in the acknowledgements. This was all quite sad, and best forgotten. I would like to thank Alfred Ajlamazyan for his generosity in providing overhead at a time when his institute had little it could easily spare. Grigory Zaigralin is thanked for his work in making the machines run, administering the money, and being his usual determined to be useful self. Igor Chudov, thanks for such effective procurement and hardware maintenance work. Eirik Fuller is thanked for his help with NFS and porting to 2.1. I would like to thank Remi Card for the superb block allocation based file system (ext2fs) that I depended on for so many years, and that allowed me to benchmark against the best. Linus Torvalds, thank you for Linux.

Previous | Content table | Next

Previous | Content table | Next

Business Model and Licensing

I personally favor performing a balance of commercial and public works in my life. I have no axe to grind against software that is charged for, and no regrets at making reiserfs freely available to Linux users. This project is GPL'd, but I sell exceptions to the GPL to commercial OS vendors and file server vendors. It is not usable to them without such exceptions, and many of them are wise enough to understand that:

- the porting and integration service we are able to provide with the licensing is by itself worth what we charge,
- that these services impact their time to market,
- and that the relationship spreads the development costs across more OS vendors than just them alone

I expect that Linux will prove to be quite effective in market sampling my intended market, but if you suspect that I also like seeing more people use it even if it is free to them, oh well.

I believe it is not so much the cost that has made Linux so successful as it is the openness. Linux is a decentralized economy with honor and recognition as the currency of payment (and thus there is much honor in it). Commercial OS vendors are, at the moment, all closed economies, and doomed to fall in their competition with open economies just as communism eventually fell. At some point an OS vendor will realize that if it:

- opens up its source code to decentralized modification,
- systematically rewards those who perform the modifications that are proven useful,
- systematically merges/integrates those modifications into its branded primary release branch while adding value as the integrator,

that it will acquire both the critical mass of the internet development community, and the aggressive edge that no large communal group (such as a corporation) can have. Rather than saying to any such vendor that they should do this now, let me simply point out that whoever is first will have an enormous advantage.....

Since I have more recognition than money to pass around as reward, my policy is to tend to require that those who contribute substantial software to this project have their names attached to a user visible portion of the project. This official policy helps me deal with folks like Vladimir, who was much too modest to ever name the file system checker vsck without my insisting. Smaller contributions are to be noted in the source code, and the acknowledgements section of this paper.

If you choose to contribute to this file system, and your work is accepted into the primary release, you should let me know if you want me to look for opportunities to integrate you into contracts from commercial vendors. Through packaging ourselves as a group, we are more marketable to such OS vendors. Many of us have spent too much time working at day jobs unrelated to our Linux work. This is too hard, and I hope to make things easier for us all. If you like this business model of selling GPL'd component software with related support services, but you write software not related to this file system, I encourage you to form a component supplier company also. Opportunities may arise for us to cooperate in our marketing, and I will be happy to do so.

Previous | Content table | Next

Previous | Content table

References

G.M. Adel'son-Vel'skii and E.M. Landis, An algorithm for the organization of information, Soviet Math. Doklady 3, 1259-1262, 1972, This paper on AVL trees can be thought of as the founding paper of the field of storing data in trees. Those not conversant in Russian will want to read the [Lewis and Denenberg] treatment of AVL trees in its place. [Wood] contains a modern treatment of trees.

[Apple] Inside Macintosh, Files, by Apple Computer Inc., Addison-Wesley, 1992. Employs balanced trees for filenames, it was an interesting file system architecture for its time in a number of ways, now its problems with internal fragmentation have become more severe as disk drives have grown larger, and the code has not received sufficient further development.

[Bach] Maurice J. Bach, ‘‘The Design of the Unix Operating System’’, 1986, Prentice-Hall Software Series, Englewood Cliffs, NJ, superbly written but sadly dated, contains detailed descriptions of the file system routines and interfaces in a manner especially useful for those trying to implement a Unix compatible file system. See [Vahalia].

[BLOB] R. Haskin, Raymond A. Lorie: On Extending the Functions of a Relational Database System. SIGMOD Conference (body of paper not on web) 1982: 207-212, See Drops section for a discussion of how this approach makes the tree less balanced, and the effect that has on performance.

[Chen] Chen, P.M. Patterson, David A., A New Approach to I/O Performance Evaluation---Self-Scaling I/O Benchmarks, Predicted I/O Performance, 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, also available on Chen’s web page.

[C-FFS] Ganger, Gregory R., Kaashoek, M. Frans, page with link to postscript paper A very well written paper focused on 1-10k file size issues, they use some similar notions (most especially their concept of grouping compared to my packing localities). Note that they focus on the 1-10k file size range, and not the sub-1k range. The 1-10k range is the weakpoint in reiserfs performance.

[ext2fs] by Remi Card extensive information, source code is available When you consider how small this file system is (~6000 lines), its effectiveness becomes all the more remarkable.

[FFS] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. ACM Transactions on Computer Systems, 2(3):181--197, August 1984 describes the implementation of a file system which employs parent directory location knowledge in determining file layout. It uses large blocks for all but the tail of files to improve I/O performance, and uses small blocks called fragments for the tails so as to reduce the cost due to internal fragmentation. Numerous other improvements are also made to what was once the state-of-the-art. FFS remains the architectural foundation for many current block allocation file systems, and was later bundled with the standard Unix releases. Note that unrequested serialization and the use of fragments places it at a performance disadvantage to ext2fs, though whether ext2fs is thereby made less reliable is a matter of dispute that I take no position on (reiserfs uses preserve lists, forgive my egotism in thinking that it is enough work for me to ensure that reiserfs solves the recovery problem, and to perhaps suggest that ext2fs would benefit from the use of preserve lists when shrinking directories)

[Ganger] Gregory R. Ganger, Yale N. Patt, ‘‘Metadata Update Performance in File Systems’’ abstract only

[Gifford], postscript only Describes a file system enriched to have more than hierarchical semantics, he shares many goals with this author, forgive me for thinking his work worthwhile. If I had to suggest one improvement in a sentence, I would say his semantic algebra needs closure.

[Hitz, Dave]http://www.netapp.com/technology/level3/3002.html A rather well designed file system optimized for NFS and RAID in combination. Note that RAID increases the merits of write-optimization in block layout algorithms.

[Holton and Das] , Holton, Mike., Das, Raj., ‘‘The XFS space manager and namespace manager use sophisticated B-Tree indexing technology to represent file location information contained inside directory files and to represent the structure of the files themselves (location of information in a file).’’ Note that it is still a block (extent) allocation based file system, no attempt is made to store the actual file contents in the tree. It is targeted at the needs of the other end of the file size usage spectrum from reiserfs, and is an excellent design for that purpose (and I would concede that reiserfs 1.0 is not suitable for their real-time large I/O market.) SGI has also traditionally been a leader in resisting the use of unrequested serialization of I/O. Unfortunately, the paper is a bit vague on details, and source code is not freely available.

[Howard] ‘‘Scale and Performance in a Distributed File System’’, Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satayanarayanan, N., Sidebotham, R.N., West, M.J., ACM Transactions on Computer Systems, 6(1), February 1988 A classic benchmark, it was too CPU bound for both ext2fs and reiserfs.

[Knuth] Knuth, D.E., The Art of Computer Programming, Vol. 3 (Sorting and Searching), Addison-Wesley, Reading, MA, 1973, the earliest reference discussing trees storing records of varying length.

[LADDIS] Wittle, Mark., and Bruce, Keith., ‘‘LADDIS: The Next Generation in NFS File Server Benchmarking’’, Proceedings of the Summer 1993 USENIX Conference.’’, July 1993, pp. 111-128

[Lewis and Denenberg] Lewis, Harry R., Denenberg, Larry ‘‘Data Structures & Their Algorithms’’, HarperCollins Publishers, NY, NY, 1991, an algorithms textbook suitable for readers wishing to learn about balanced trees and their AVL predecessors.

[McCreight] McCreight, E.M., Pagination of B*-trees with variable length records, Commun. ACM 20 (9), 670-674, 1977, describes algorithms for trees with variable length records.

[McVoy and Kleiman], the implementation of write-clustering for Sun’s UFS.

[OLE] ‘‘Inside OLE’’ by Kraig Brockshmidt, discusses Structured Storage, HREF="http://www.microsoft.com/mspress/books/abs/5-843-2b.htm" abstract only

[Ousterhout] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M.D. Kupfer, and J.G. Thompson. A trace-driven analysis of the UNIX 4.2BSD file system. In Proceedings of the 10^{th} Symposium on Operating Systems Principles, pages 15--24, Orcas Island, WA, December 1985.

[NTFS] ‘‘Inside the Windows NT File System’’ the book is written by Helen Custer, NTFS is architected by Tom Miller with contributions by Gary Kimura, Brian Andrew, and David Goebel, Microsoft Press, 1994, an easy to read little book, they fundamentally disagree with me on adding serialization of I/O not requested by the application programmer, and I note that the performance penalty they pay for their decision is high, especially compared with ext2fs. Their FS design is perhaps optimal for floppies and other hardware eject media beyond OS control. A less serialized higher performance log structured architecture is described in [Rosenblum and Ousterhout]. That said, Microsoft is to be commended for recognizing the importance of attempting to optimize for small files, and leading the OS designer effort to integrate small objects into the file name space. This book is notable for not referencing the work of persons not working for Microsoft, or providing any form of proper attribution to previous authors such as [Rosenblum and Ousterhout].

[Peacock] K. Peacock, ‘‘The CounterPoint Fast File System’’, Proceedings of the Usenix Conference Winter 1988

[Pike] Rob Pike and Peter Weinberger, The Hideous Name, USENIX Summer 1985 Conference Proceedings, pp. 563, Portland Oregon, 1985. Short, informal, and drives home why inconsistent naming schemes in an OS are detrimental. http://achille.cs.bell-labs.com/cm/cs/doc/85/1-05.ps.gz His discussion of naming in plan 9: http://plan9.bell-labs.com/plan9/doc/names.html

[Rosenblum and Ousterhout] ‘‘The Design and Implementation of a Log-Structured File System’’, Mendel Rosenblum and John K. Ousterhout, February 1992 ACM Transactions on Computer Systems, this paper was quite influential in a number of ways on many modern file systems, and the notion of using a cleaner may be applied to a future release of reiserfs. There is an interesting on-going debate over the relative merits of FFS vs. LFS architectures, and the interested reader may peruse http://www.sunlabs.com/~ouster/seltzer93.html and the arguments by Margo Seltzer it links to.

[Snyder] , ‘‘tmpfs: A Virtual Memory File System’’ discusses a file system built to use swap space and intended for temporary files, due to a complete lack of disk synchronization it offers extremely high performance.

[Vahalia] Uresh Vahalia, ‘‘Unix Kernal Internals’’