An Object Store for The Walnut Kernel

by

Stewart Smith, BCompSc

sesmith@csse.monash.edu.au

Literature Review

Bachelor of Computer Science with Honours (1608)

Supervisor: Dr. Ronald Pose

rdp@csse.monash.edu.au

Associate Supervisor: Dr. Carlo Kopp

carlo@csse.monash.edu.au

School of Computer Science and Software Engineering

Monash University

November, 2003

Contents

Chapter 1

Literature Review

The Walnut Kernel(Castro, 1996) is a persistent, capability based operating system developed at and extending the previous work in the School of Computer Science and Software Engineering at Monash University. Building on work in the Password-Capability System(Anderson, Pose and Wallace, 1986), Walnut has some unique features such as its capability architecture, charging scheme and persistence. Walnut requires a slightly specialised set of features from its equivalent of a file system, the Walnut object store. Various deficiencies with the original design were found, and it was decided that alternative systems should be explored.

Most modern Operating Systems base their file semantics on those of 4.4BSD UNIX with a largely POSIX compliant programmers interface. This system is well tested and its faults and shortcomings are well known. However, many advances in data storage methods have been made on such systems and they deserve close attention.

1.1 Persistent Systems

Persistence is the property of a system where created objects continue to exist and retain their values between runs of the system. An example of a persistent system is a persistent programming language, in such a language the contents of variables are preserved across runs of the program.

A persistent operating system is different from most popular operating systems as (ideally) no state is lost after an (expected or unexpected) system restart. In Walnut, processes are first class objects and are backed to disk. Although persistence has existed in various forms for a long time, occasionally making its way into a production system(Liedtke, 1993), they have not received widespread usage. More recently, there has been an interest in persistence in distributed systems(Elnozahy, Johnson and Zwaenepoel, 1992). There have been many interesting, and rather complete, persistent research operating systems developed including the focus of our research, The Walnut Kernel.

1.1.1 The Walnut Kernel

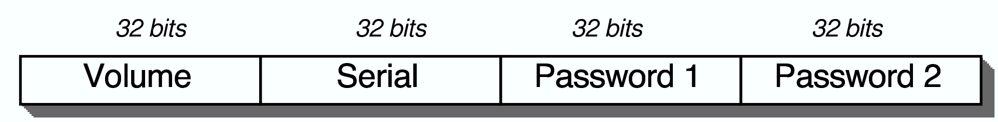
The Walnut Kernel(Castro, 1996) builds on previous work in the Password-Capability System(Anderson et al., 1986) to create a persistent and secure operating system for general use. Walnut is designed to be portable[1] across different architectures but currently only runs on the Intel i486 compatible CPUs with limited support for other PC peripheral hardware. It has been shown that it is possible to implement ANSI stdio compatible inter-process communication to Walnut(Kopp, 1996) and gain most of the functionality of a modern UNIX like Inter-Process Communications (IPC) system.

The basic unit of storage abstraction in Walnut is an object and all data made a available to user programs is made available through objects. An object is a series of pages of which any number of these are memory mapped into a process' address space. In the current implementation, the page size is that of the host CPU architecture. This design makes it very difficult to exchange data between architectures with dissimilar page sizes. This is a problem and one which future work on Walnut will one day have to address.

Volumes represent physical media on which Walnut Objects are stored. In the current system, Objects are permanently associated with a volume (i.e. you cannot transparently move an object to another volume) and objects cannot span more than one volume. Each volume has a unique identifier, known as the **volume number**. The current system only allows 32bits for the volume number, so if Walnut were to become widely used, collisions are bound to occur. There is currently no facility to deal with this.

A **Serial** number is the unique identifier for an object on a volume - rather similar to the UNIX i-node number. A serial is only unique for a specific volume and should be as random as possible as part of Walnut's security is the difficulty of guessing valid object identifiers.

A **Password Capability** is associated with a set of attributes: a set of rights and a view. The capability is used to identify what access permissions the holder has to an object.



32 bits	*32 bits*	*32 bits*	*32 bits*
Volume	Serial	Password 1	Password 2

Figure 1.1: Structure of a Walnut Kernel Capability

All Walnut objects and processes (except the `init` process) are persistent. It is thought(Wallace, Pose, Castro, Kopp, Pringle, Gunawan and Jan..., 2003) that any clean up of user device

[1] Although the current implementation lacks certain infrastructure and code design to accomplish this

drivers could either be handled by the `init` process on startup or by receiving a specific type of message on system reboot.

Walnut objects have the following properties(Castro, 1996):

- They are permanently associated with a volume. They cannot be moved to another volume or span more than one volume.
- Pages are allocated on the first reference to them.
- If the number of guaranteed pages has been exceeded, and there are unreserved pages on the volume, then additional pages may be allocated to the object.
- If there are not any unreserved pages available, an exception will occur.
- Attempts to access beyond the limit of an object will cause an exception.
- The main memory acts as a cache of objects.

The backup and restoration of individual objects becomes problematic when an object is permanently associated with one volume (as it is in Walnut). Unsolved problems include: how to backup a volume without compromising the security of the system, how to restore a volume without compromising system security, how to backup a single object and how to restore a single object (especially if the original has been removed) without compromising security. Currently, the accepted attitude(Wallace et al., 2003) is that it is each users duty to back up their own objects. They could do this by periodically running a backup process and giving it capabilities to objects they want backed up. This does not, however, facilitate restoration of objects and capabilities.

Walnut builds on the Password-Capability System's(Anderson et al., 1986) concept of rental, which is used for the garbage collection of objects(Wallace and Pose, n.d.). To continue to exist, an object must be able to afford to pay for the resources it uses. Each object has an amount of money (stored in its **money word**) which may be drawn by those holding capabilities with withdraw rights or by the Walnut Kernel for the use of storage space. Processes also have a **cash word** which stores money used to pay for kernel services. The **rent collector** process periodically deducts rent for disk space occupied by objects. It is commonly accepted(Wallace et al., 2003) that the rent collector should be flexible, possibly charging different amounts at times of high load or high disk usage.

Charging for services also increases the security of the system(Wallace and Pose, n.d.). If each attempt to load a capability is charged for, then attempting to guess a valid capability becomes an extremely expensive operation. It is thought(Wallace et al., 2003) that charging more for failed attempts could also be useful.

Walnut Disk Structures

A Walnut Volume has three components: the **Disk-ID-block**, the **bitmap** and all other blocks which are used for storing objects. It is important to note that Walnut does not currently provide any form of journaling, consistency checking utility or guaranteed synchronous updates. This means that robustness of the current Walnut data store does not compare favourably to that of modern file systems in the event of a crash, reflecting its origins as a research platform.

On disk, Walnut objects are split into two parts: the **body** and the **dope**. The dope is all the system information related to an object including the object Header, the capability table and a set of page tables. The body is the contents of the object. The dope may grow or shrink in size separately from that of the body. A process object is the only type which enforces structure on the body but apart from that there is no distinction between processes and other objects.

Walnut does not currently keep a separate index data structure for translating object serials into disk locations of the header blocks. Instead, Walnut uses the least significant bits of the object's serial number as the block number of the **first header block** of the object. To prevent a possible attack on the security of the system by forging a serial number and a **first header block** to gain unauthorised access to other objects, the volume bitmap tracks which blocks are legitimate first header blocks. The bitmap contains a two bit value describing each block on the volume. A block can be **free**, a **first header block**, **in use** or **bad**. A full diagram of the walnut disk structure can be seen in Figure 1.2.

This method of locating objects has several drawbacks; the **rent collector** needs to operate on all objects on a volume and the only way to currently do this is to search the volume bitmap for blocks marked as **header blocks**, seek to that disk block, read the header blocks for the object, update the money word and write the header blocks back to disk. This is going to cause a lot of seeking and become problematic during periods of high disk IO. The current model also reduces what serial numbers you can store on a specific volume without having collisions with other object headers or data blocks.

The most critical drawback of this indexless scheme is the problem of automatically resuming of processes. As there is no quick way to locate process objects on a volume, the header blocks for **all** objects on a volume must be read and examined to see if they represent a process. On small volumes, this IO is fairly insignificant, but with the size of today's disks and the number of objects that todays users store on disk, the amount of IO needed to query each object on a volume to determine if it's a process becomes staggering.

As the implementation of a UNIX like environment on Walnut is possible(Kopp, 1996) and is generally viewed as the quickest and most effective way to port software to the system, a comparison with the workloads of other UNIX based systems would seem fair. It should also be noted that a 4.2BSD environment was implemented on Walnut's predecessor, the Multi although nothing was ever published about this achievement. A quick survey on the number of files present on a variety of UNIX based systems (Table 1.1.1) shows that the

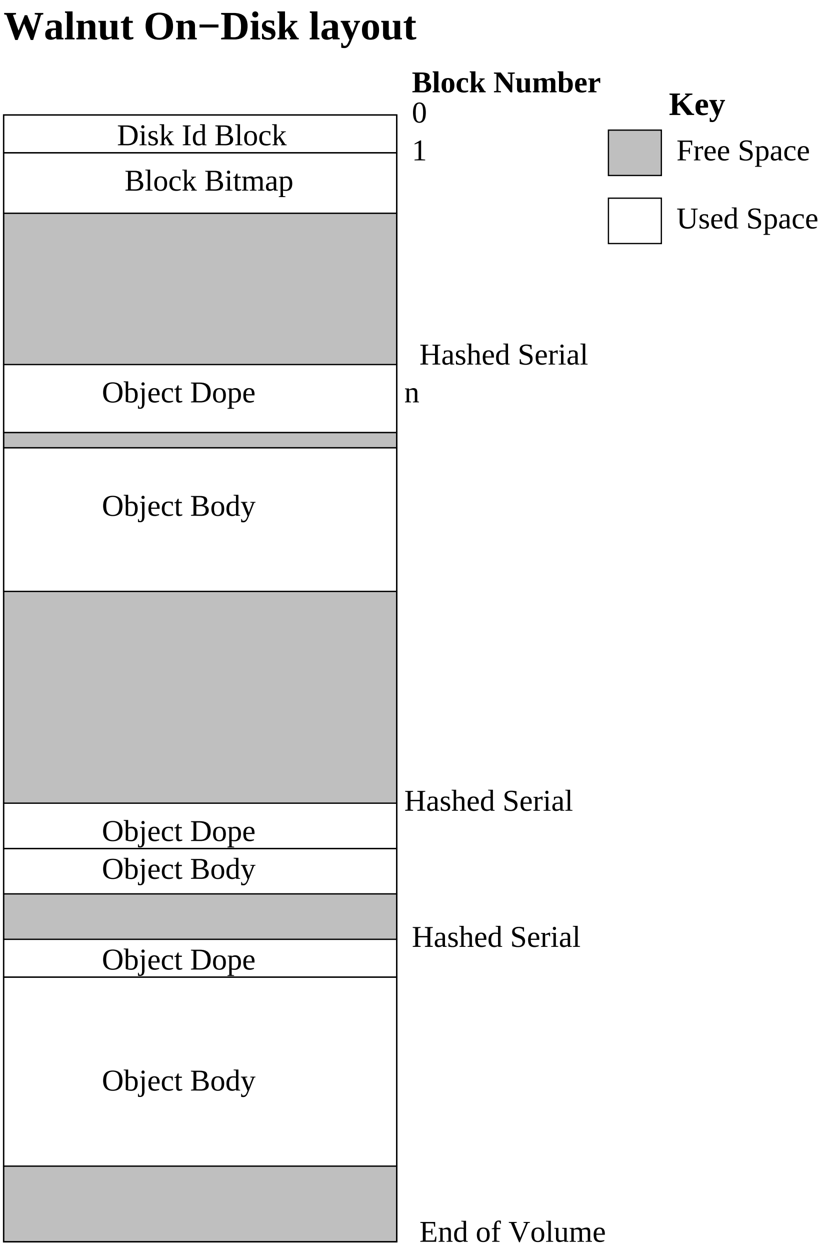


Figure 1.2: Walnut Disk Layout

Machine	Used Disk Space	Number of Objects (files)
pheonix	1.7GB	109,365
saturn	23GB	299,528
crashtestdummy	810MB	44,344
willster	34GB	422,095
cancer	71.3GB	3,065,590
yoyo	35.8GB	1257217

The machines pheonix, saturn, crashtestdummy, cancer and willster all run Debian GNU/Linux and are my personal machines while yoyo is a shared system with 50 to 60 regularly active users and several hundred rather inactive users.

Figure 1.3: Survey of disk usage on various UNIX based systems

number of objects even on small disks is quite large. With even the smallest disk examined, scanning header blocks for over 100,000 objects will consume significant disk bandwidth and even in the best conditions for disk I/O, this would take more than a few seconds of real time.

There is one area in which the current system completely fails and that is providing any form of inter-object consistency after an unclean shutdown. Since persistence is meant to be transparent to processes, the data they access should not change between the process being suspended and the process being resumed. This means that the image on disk must be a snapshot of the process and all of its data at a specific moment in time as any variation would have the same effect of modifying a running processes data which will in all likelihood lead to unpredictable results(Smith, 2003).

The Current State of the Walnut Kernel

The current Walnut implementation is unfortunately not up to the standard of other operating system projects and strongly shows its immaturity. There has been little work on the kernel since its original implementation and a lot of knowledge about how parts of it work has been lost over time. Although recent work by Stanley Gunawan has updated the code and build system so that development can be done on recent releases of FreeBSD (namely 4.4 and 4.5), problems have arisen in trying to get user mode processes to execute. There is very little hardware support and any change in hardware configuration requires source code modifications.

The Walnut source code is sparsely documented and is believed(Wallace et al., 2003)) to use “reserved” and “unused” fields in some data structures for internal scratch space. There is an overuse of constructs such as goto that have the effect of greatly obsfucating large parts of the kernel. While such constructs are can be useful in speed optimisation and occasionally even code clarity, this is not the case in much of the Walnut code. In many of the subsystems it is a complex operation to understand what is being done and how it is being done. If a port to a 64bit architecture were to be attempted, almost the entire code base would need to be rewritten or closely audited to remove assumptions on word size.

Walnut has proven great theory, spawning several papers and other projects but the current implementation would need a large amount of work, or indeed a total rewrite before extensions to the original design could be pursued within a reasonable time frame on the existing code-base.

1.1.2 Mungi

One of the larger research projects regarding persistent systems is Mungi. Mungi(Heiser, Elphinstone, Vochteloo, Russell and Liedtke, 1998) is a persistent, Single-Address-Space Operating System (SASOS) developed on top of the L4(Ceelen, 2002) Micro kernel at the University of New South Wales. The system was designed to be a pure SASOS without sacrificing features such as protection, encapsulation and orthogonal persistence. The base abstractions that Mungi provides are: capability, object, task, thread and protection domain. Mungi also has the concept of a bank account which is (similar to Walnut's concept of money) is used to limit and control resource use.

Like Walnut, an object in Mungi is the basic storage abstraction. All objects exist within a 64bit address space and since Mungi was designed for 64bit platforms (such as MIPS and Alpha) the single address space was not deemed an unreasonable limitation on the storage capacity of a single system. The single address space approach does mean that an implementation on a 32 bit architecture would Beverly limit the amount of storage accessible compared to modern standards.

Mungi attempts to improve efficiency by having three classes of data objects:

- Transient and unshared
- Transient and shared
- Persistent

Persistent system purists would argue that Mungi is not a pure persistent system due to its support for non-persistent objects. Mungi supporters would rebut this with claims of performance improvements. Since there is not a variety of completed persistent systems to perform real-world performance tests, the claims of added performance cannot be proven over other techniques. It has been indicated(Ceelen, 2002) that transient objects in Mungi would only be used for device drivers and other objects which cannot easily be restored after a system reboot.

A Mungi object exists until it is explicitly destroyed. For each process, Mungi keeps a **kill list** of objects which that process has created. When the process finishes, Mungi will remove all processes on the **kill list**. There exists a system call to remove an object from the **kill list** so that it may outlive its creator. This approach contrasts sharply with the

Walnut view of money and paying for services (including storage). The Mungi approach means that it is possible for a buggy process to create objects, not reference them and have them exist for long periods of time (or forever if the process never quits). In Walnut, these objects would be garbage collected if they are unable to pay rent.

Unfortunately, the currently available implementations of Mungi do not have persistence implemented. Additionally, the requirement of 64bit hardware supported by L4/MIPS or L4/Alpha greatly reduces the range of machines able to run Mungi and totally eliminate commodity PC hardware.

1.1.3 Consistency in Persistent Systems

One of the main problems with persistent systems is how the on disk representation of the system should be kept consistent in the event of a crash. Since processes are persistent and will usually reference objects other than themselves, it is important that these objects were all written to disk in one atomic operation, or data a process is using could mysteriously change or disappear (due to it not have being flushed to disk)(Smith, 2003).

One method to ensure on disk consistency is Checkpointing(Skoglund, Ceelen and Lidtke, 2000). Checkpointing works by taking a snapshot of the contents of memory at a specific moment and writing that snapshot atomically to disk. The last (successful) snapshot written to disk will be the state in which the system is restored.

It has been shown(Elnozahy et al., 1992) with distributed applications that the overhead for checkpointing can be quite minimal (less than 1% for six out of eight applications, with the highest overhead at 5.8%). It is natural that **incremental checkpointing** is used to reduce the number of disk writes for each snapshot. Using **copy-on-write** memory protection, it is possible for a snapshot to be written to disk while processes continue to execute, hence having asynchronous checkpointing(III and Singhal, 1993). Such techniques were also raised during Wallace et al. (2003) were thought to be worth investigating for Walnut.

In the worst case scenario, each time a snapshot is taken (Elnozahy et al. (1992) used two minutes as an interval) the entire contents of memory is dirty and must be written to disk. It is possible that such methods as partially writing snapshots in between the time they are taken or a variation in time when snapshots are taken could help alleviate this bottleneck. It has also been shown that relaxing consistency(Janssens and Fuchs, 1993) can help in reducing the overhead of checkpointing.

1.2 Existing File Systems

Since most popular Operating Systems use an explicit storage system and a variation on UNIX file semantics, an examination of how file systems work, their performance and reliability could help in understanding what features a Walnut data store requires.

Although there has been many papers and books published on data structures, using these structures on disk is rather different than using them in memory. The overwhelming majority of texts focus on in-memory structures and not on how to optimise these for use on disk. Folk and Zoellick (1987) discusses several of the differences between data structures in memory and on disk (within files). It is clear from Folk and Zoellick (1987) that many trade offs are made in the design of file formats, it also becomes clear that it is the same way with file systems.

1.2.1 BSD FFS

The Berkeley Software Distribution's Fast File System (FFS)(*A Fast File System for UNIX*, 1984) dramatically improved file system performance over existing systems. Improvements over previous UNIX File Systems introduced by FFS include:

- Larger block sizes (at least 4096 bytes)
- the use of *cylinder groups* to exploit the physical properties of a disk to reduce seek times.
- improved reliability though careful ordering of file system meta-data writes

A lot of the performance gain of FFS was due to the use of larger disk blocks (4096 byte or larger instead of the more common 512 byte), allowing more files to fit into a single disk read. However, with 4096 byte blocks *A Fast File System for UNIX* (1984) reported that with a set of files about 775MB in size, about 45.6% of the disk space was used by the file system. The solution FFS chose was to split each block into fragments of 512 bytes (or more commonly 1024 bytes) and allocate fragments instead of disk blocks. This allowed the speed increases of having a larger block size while not wasting space with small files.

FFS keeps with the basic UNIX file system abstractions. Each i-node represents a file on disk, it contains information such as the length of the file (in bytes) and what disk blocks are being used by it. FFS i-nodes have direct, indirect and doubly indirect block pointers (See Figure 1.4 (Card, Ts'o and Tweedie, n.d.)).

This means that for large files it is faster to locate disk blocks for the beginning of files than it is towards the end. In this way, FFS is biased towards small files, where only the direct block pointers are used. This is a valid assumption for many UNIX systems as there tends to be a large amount of small files.

Directories are files containing a list of names and i-node numbers. The special entry '.' means the current directory (and should be the currently i-node number) and '..' means the directory above the current directory. The directory structure is a tree. I-nodes may appear multiple times within the tree as FFS allows for "hard links"; i-nodes can appear under more than one name.

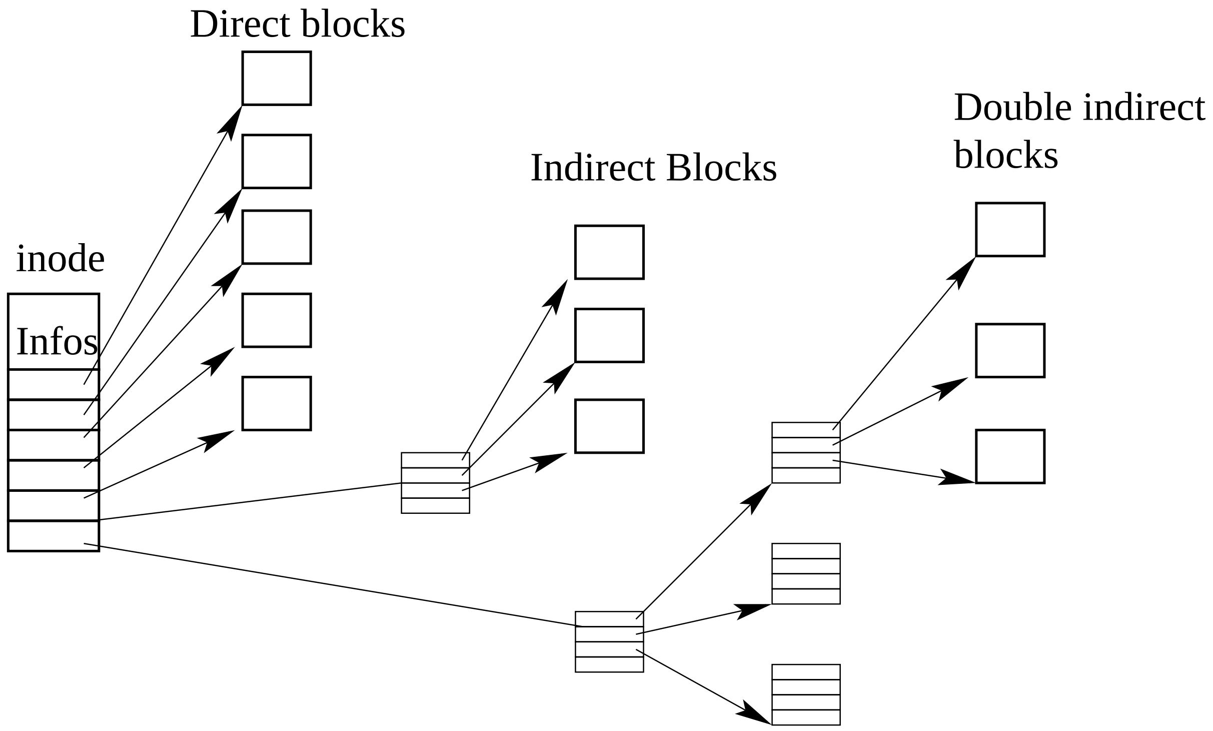


Figure 1.4: Direct, Indirect, Doubly Indirect and Triply indirect blocks in file systems such as FFS and ext2

For a file (e.g. /home/test/testfile) on an FFS volume to be accessed, the following sequence of events must take place (assuming the volume is mounted):

- Read root directory i-node
- Read root directory blocks, searching for path entry ('home')
- Read sub-directory's i-node
- read sub-directory's blocks, searching for path entry ('test')
- read directory's i-node
- read directory's blocks, searching for path entry ('testfile')
- Read file's i-node
- read file's blocks.

If all directories in this path occupied one disk block, there will need to be 7 blocks read from disk *before* any of the file's content is read.

However, the advantage FFS has over many of its predecessors is *cylinder groups* (Figure 1.5). A cylinder group is a collection of one or more cylinders on a disk. Each cylinder group has its own i-node table, block bitmap and backup super block. The purpose of a cylinder group is to exploit the benefits of locality. Files and directories used together should be in the same cylinder group, reducing seek time. The users also has the advantage of being able to do simultaneous updates to the file system meta data if the implementation supports fine grained locking as different threads could operate on different cylinder groups.

However, most modern disks hide their physical geometry from the operating system, instead preferring a *Logical Block Addressing* (LBA) scheme where disk blocks are not referenced by Cylinder, Head and Sector but by a block number. In these systems, block N+1 will be the next block to N (possibly read by a different head, but this is transparent), following the direction that the disk rotates and the transition between cylinders is transparent.

Due to the array of i-nodes being separate from the file data, seeks are inevitable between reading an i-node and reading the data. Cylinder groups reduced the distance considerably but FFS performance suffers from this, as do all systems which keep i-nodes separate from data. The speed of being able to look up an i-node in O(1) time (i-node array start block number + i-node number) is at the expense of a seek to i-node data. Caching of i-nodes by the Operating System helps overcome this seek time for frequently or recently accessed i-nodes, but initial lookup does not benefit.

Since FFS prefers to store all the files in a directory within the same cylinder group and if we make the assumption that files within a directory will be accessed at roughly the same

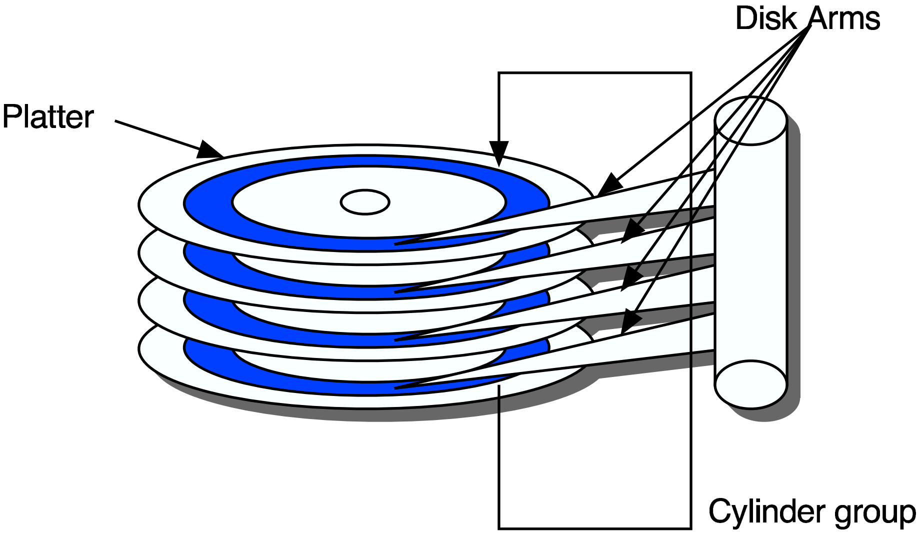


Figure 1.5: FFS Cylinder Group

time (for example, opening a Maildir style mailbox), a read ahead by the operating system could greatly reduce disk seeks from the i-node table to file data.

The reliability of FFS is largely based on carefully ordered synchronous meta data writes. Meta data writes are carefully ordered so that in the event of a crash, the `fsck` (File System ChecK) utility can relatively easily recover the file system to a sane state. However, improvements have been made which allows the file system check operation to happen in the background. These improvements are known as Soft Updates.

Soft Updates

Soft Updates(McKusick and Ganger, 1999) is a new and rather interesting way to help improve the reliability and recovery time of BSD's FFS and remove the need to wait for a file system check after an unclean mount. The principle behind Soft Updates is to track meta data dependencies and do carefully ordered writes to disk to ensure that on disk meta data is consistent. The only operation required after a crash with Soft Updates is a `fsck` process that can reclaim any allocated but unused disk blocks. This process can be run in the background as freeing blocks does not require exclusive access to the file system.

Because of the added complexity of tracking meta data dependencies, it is admitted by the authors of the Soft Updates code in FreeBSD(*The FreeBSD Handbook*, 1995-2003) there is a higher risk of bugs in the code. There is also a higher memory penalty for using soft updates but the main advantages are the little (or no) change to the on disk format and unlike a journaling filesystem, meta data is not written to disk twice. There has been much debate over which approach (soft updates or journaling) provides better performance or is a cleaner approach although little data backing up any of the arguments has been produced and there are not any (easily locatable) convincing benchmarks.

1.2.2 Linux ext2

The ext2 file system(Card et al., n.d.) has been, for many years, the defacto Linux file system. Recently, most distributions have replaced it with the ext31.2.3 due to its added reliability. The design of the ext2 file system was heavily influenced by FFS. The data structures are rather similar in design, the most notable exception being the absence of fragments.

The main difference with ext2 is the relaxed data and meta-data integrity. All writes to the ext2 file system only make it to disk when the Operating System chooses to flush dirty blocks to disk. There is the option to do synchronous meta-data updates, but it is seldom enabled due to the loss in performance. This relaxed attitude to on-disk consistency accounts for many of ext2's speed advantages over other file systems (Giampaolo, 1999). The down side is the higher risk of file system corruption in the event of a crash and the added complexity in recovery for `fsck` utilities. With the work in Tweedie (1998), journaling has been added to the ext2 file system without a significant performance penalty.

There has also been some work to solve the problem of users accidently deleting files with the ext2 undelete project(Crane, 1999). The ext2 undelete, like undelete programs for other file systems, attempt to re-link a directory entry with the (now deallocated) i-node. This solution is not ideal, as it depends on the file system being in much the same state as when the file was removed for the i-node not to have been re-used. It also requires exclusive write access to the file system, something that is unacceptable in a multi-user environment. Debate on if this is a job that a file system should attempt to solve carries on and is unlikely to be resolved.

The ext2 file system, like FFS before it, does not cope very well with large directories. When FFS and ext2 were designed, this wasn't too much of a problem as disks were not large enough to hold enough files for this to become a real problem. However, with today's large disks and systems such as the Maildir mailbox format (where each email message is a file in a directory) it is conceivable to have directories with 60,000 files in them[2].

There has been work to overcome the inefficiencies in directory lookup in the ext2 filesystem. The Directory Index(Phillips, 2001) project has managed to devised method of directory indexing that is both backwards compatible with existing ext2 file systems and forwards compatible so that older ext2 code can fully access directory indexed volumes.

It is clear from ext2 that extensibility of the file system is very important for its continued use. There have been several important features added over time without breaking compatibility backwards compatibility.

[2] Real-world example of the linux-kernel mailing list messages from January-October 2003

1.2.3 Linux ext3

The ext3 filesystem is the same as the ext2 filesystem except for the addition of a transactional meta data journal(Tweedie, 1998). The implementation is fairly standard, employing the expected optimisations such as batched transactions. The on disk format is compatible with ext2, the journal simply being another file on the disk (albeit with a special i-node number). The goal of the ext3 project was to not destabilise the ext2 codebase but to add one new feature.

1.2.4 Network Appliance's WAFL

Network Appliance's (NetApp) WAFL(Hitz, Lau and Malcolm, n.d.) system grew out of the desire to have a solid and reliable system for networked file servers. WAFL (Write Anywhere File Layout) uses Snapshots (read only clones of the active file system) to provide access to historical data (for example, how the file system looked yesterday) and to ensure on disk consistency.

The requirements for their NFS server were(Hitz et al., n.d.):

- provide a fast NFS service
- support large file systems (tens of GB) that grow dynamically as disks are added
- high performance while supporting RAID
- restart quickly, even after an unclean shutdown due to power failure or system crash

Snapshots are made available to users through the `.snapshots` directory. Figure 1.6 (the example from (Hitz et al., n.d.)) shows how a user may recover what was in their file at the time of any of the snapshots being taken. The main advantage to system administrators is being able to take a live file system and reliably back up its contents.

The WAFL layout is best thought of as a tree where for each snapshot, the root node is duplicated and any modified nodes are copied to new locations and referenced to by the snapshot's root node. Figure 1.7 (adapted from (Hitz et al., n.d.)) illustrates the layout of the file system before and after snapshot creation, and modification.

Because WAFL only duplicates modified blocks, it is able to create a snapshot every few seconds to allow quick recovery after unclean shutdowns. When the new snapshot is created, the old one is marked as consistent. When the system starts up, it uses the most recent snapshot that was marked as consistent. Combined with a small (NVRAM based) log, this provides rapid crash recovery.

Batched disk writes and the ability to write any data to any part of the disk lead WAFL to perform rather well in NFS benchmarks. Hitz et al. (n.d.) debates the validity of these bench marks, it is fair to assume that the WAFL approach achieves good performance.

```
spike% ls -lut .snapshot/*/todo
-rw-r--r-- 1 hitz 52880 Oct 15 00:00
.snapshot/nightly.0/todo
-rw-r--r-- 1 hitz 52880 Oct 14 19:00
.snapshot/hourly.0/todo
-rw-r--r-- 1 hitz 52829 Oct 14 15:00
.snapshot/hourly.1/todo
...
-rw-r--r-- 1 hitz 55059 Oct 10 00:00
.snapshot/nightly.4/todo
-rw-r--r-- 1 hitz 55059 Oct 9 00:00
.snapshot/nightly.5/todo
```

Figure 1.6: User Access to Snapshots in WAFL

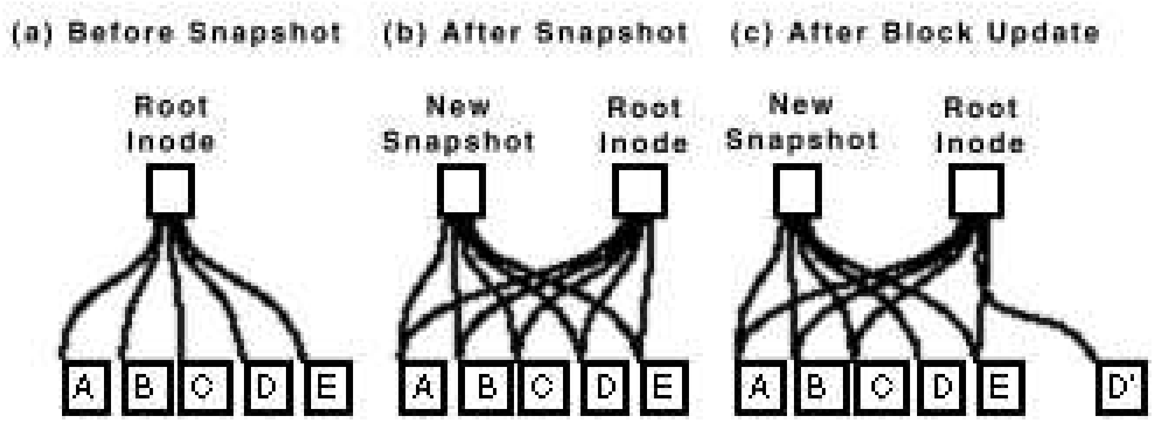


Figure 1.7: WAFL Before and After Snapshot

1.2.5 Reiser FS

ReiserFS has grown around Hans Reiser's desire to integrate the name space in the operating system. The desire was to create a fast and reliable filesystem for Linux to replace the ext2 file system. ReiserFS is a relatively new filesystem and some of it's tools (such as the file system check utility) have not yet reached maturity. However, many users like it due to its speed and efficiency, especially with small files.

Reiser (2001) claims that through the use of balanced trees, it is possible to get large performance improvements in file system throughput and fewer disk seeks during tree traversal. Reiser (2001) also believes that existing literature focuses too much on the worst case scenario, where none of the tree is cached. Reiser (2001) believes that this is not a useful thing to study as in the real world, parts of the tree will be cached for most of the time and performance in this case is more important.

The Reiser tree is designed to allow efficient handling of directories with thousands or hundreds of thousands of files, something that was previously inefficient. ReiserFS will also pack many small files into one block, so 100 byte files do not occupy one disk block each. BeFS(Giampaolo, 1999) can store a small amount of file data within the i-node, but each file will still use one full disk block. The reiserfs approach packs multiple i-nodes and data into a block, thus wasting less space. Nodes on disk which contain these small files are known as formatted nodes.

The use of formatted notes does mean that when a file outgrows the space it has in a formatted node, the system must spend extra time copying to elsewhere on disk. Eventually, as formatted nodes become more fragmented, a repacker will have to be run to help clean things up. This is the same fragmentation problem that has plagued file systems, the exception being that the reiserfs repacker should work on mounted file systems and work transparently.

The other problem with formatted nodes (which is mainly because of the design of the Linux kernel) is that memory mapping files stored in formatted nodes requires extra work. The Linux kernel can only memory map on page boundaries, which are usually the same size as a disk block (on i386, this is 4k). The computational overhead is an accepted tradeoff for the added disk space savings.

ReiserFS uses a block bitmap and some intelligent searching algorithms for allocating disk space. Reiser (2001) has empirically shown that their method, which could require extra reads from disk to traverse the tree gains good allocation and locality. With Reiser4(MacDonald, Reiser and Zarochentcev, 2002), in memory transactions use a duplicate of the block bitmap with changes only being committed once transactions enter the COMMIT stage. This allows for greater parallelism in writes to the file system as each transaction does not need to lock the single copy of the block bitmap.

Reiser (2001) argues that the speed advantages FFS gained through cylinder groups was more to do with placing semantically adjacent nodes close together on disk. Reiser (2001) found that this was an excellent approximation of the optimisations of placing data on disk according to actual cylinder boundaries (which are now often hidden from the programmer). Reiser differs from FFS in its implementation of locality is separated from the semantic layer, and it is (theoretically) possible to introduce smarter locality logic to produce a better locality_id for each object_id. A possible improvement could be to monitor usage patterns and lay out objects on disk accordingly. It was found that allocating blocks in the direction the disk spins (in increasing block numbers) significantly boosted performance over other policies of just allocating blocks 'near' each other..

Reiser (2001) debates the merits of whether files should be block aligned, and argues that the ReiserFS model of not block aligning files gives higher performance for small files. Reiser (2001) also makes the conjecture that the current usage patterns of file systems are because of how the file systems have been designed. Current systems are not optimised for small objects, and at layers above the file system, much aggregation of objects is done. If it were more optimal to have many 100 byte files than one 4kb file, then maybe each option in a configuration file would become a separate object on disk. This is very interesting when considering that it is impossible to get any usage statistics for Walnut (as so few systems exist) and user code is limited. Current Walnut implementations limit objects to multiples of page size, but this is seen as a major limitation.

Reiser (2001) does state that the most interesting features are yet to come, and some of these are starting to appear in the next revision of ReiserFS, Reiser4.

Reiser FS 4

The main difference between Reiser FS Version 3 and Version 4 is that Version 4 is an atomic filesystem(Reiser, 2002-2003). Each operation (including writes) happens atomically, giving the impression of full data journaling. Reiser actually uses a similar system to WAFL(Hitz et al., n.d.), Wandering Logs.

Wandering logs means that any (free) area on disk can become part of the on disk journal. This is used so that each file write can be journaled physically close to the file(MacDonald et al., 2002). This can dramatically reduce the number of disk writes needed to achieve atomic writes as instead of copying blocks from the log into the file, we can simply rewrite the block pointers in the file system to point to the blocks in the log. The disadvantage is that it is possible for files to become more fragmented after an update on a disk where the free space is highly fragmented. Although, subsequent operations will not suffer this fragmentation and over time, this could actually decrease the fragmentation of files on disk.

The Reiser (2002-2003) tree, known as a 'dancing tree', which means it is only balanced in response to memory pressures triggering a flush to disk or as a result of a transaction closure (which forces nodes to be flushed to disk)(Reiser, 2002-2003). The speed improvements come from balancing the tree less often than with every update

This is similar to optimisations other systems (XFS in particular) make just before they commit to disk. It becomes clear that batching file system operations and optimising their flush to disk improves both read and write performance. This stems from how the designers look at the problem, reiserfs and xfs designers view the file system problem as one of many disk operations, not just a sequence of single disk operations and their design (and increased performance) indicates this.

1.2.6 BeOS BeFS

In early versions of the Be Operating System, extra information about files was kept in a database file on top of the filesystem. The separate database design was chosen due to the engineers desire to keep as much code in user space as possible. With wider use

```
typedef struct block_run
{
        int32 allocation_group;
        uint16 start;
        uint16 len;
} block_run;
```

Figure 1.8: BeFS Block Run

of the system, especially the POSIX support (which did not interface with the database file) problems were seen with keeping the database in sync with the contents of the file system.(Giampaolo, 1999)

The Be Filesystem (BeFS) was created out of a need for the BeOS to have a unified filesystem interface (VFS layer) and a fast, 64bit, journaled and database like filesystem(Giampaolo, 1999). Because of the nature of BeOS and their target audience, the ability to handle media files was a priority.

The BeFS is also interesting in that two engineers made the first beta release in only nine months with the final release shipping a month later. This is especially interesting given the high regard the BeOS file system is held in by many users of it. It's indexing capabilities are the envy of users of other systems even over six years after it's initial release and several years after the demise of the company.

The journaling implementation in BeFS is rather interesting as it does support the journaling of file data, but due to the limited size of the log file, only directory data is actually journaled. It would theoretically be possible to add data journaling to BeFS by allowing the journal to dynamically change size.

The BeFS **block_run** structure (Figure 1.8, from (Giampaolo, 1999) P47-48) is a unique way to address disk blocks. It takes advantage of the optimal (and common) case of several sequential blocks being allocated to a single file.

1.2.7 SGI's XFS

Silicon Graphics started the XFS project to replace their aging EFS file system under their version of UNIX, IRIX. The Project Description(Doucette, 1993c) listed the goals of XFS as: scalability (especially for large systems), compatibility among all supported SGI machines (especially small machines), support the functionality of EFS, to outperform EFS, high availability, quick recovery from failures and the system must be able to be extended in the future(Anderson, Doucette, Glover, Hu, Nishimoto, Peck and Sweeney, 1993).

The requirements to satisfy SGI's customers were rather unique at the time. SGI produced both high end (compared to the general PC industry) and very high end (1024 processors) machines and had many customers in the media business who needed to store large files, and lots of them. There was also the scientific community who often needed to operate on large sets of data, including large, sparsely populated arrays. There was also the need to have good performance on small files (less than 1kb) as most / and /usr file systems have many such files(Doucette, 1993c).

XFS was designed to be a 64 bit file system and SGI had to deal with the problems of integrating support for 64 bit file offsets into a system that largely relied on 32 bit offsets(Sweeney, 1993a). There was the unfortunate consequence that user code had to be changed to support the larger offsets and extra system calls were added. This has been the way that 64 bit file offsets have been implemented in several UNIX variants (including Linux) and the general consensus of the community is that this is the best way.

XFS gains great scalability(Sweeney, Doucette, Hu, Anderson, Nishimoto and Peck, n.d.) through its use of kernel threads(Doucette, 1993a) and message system(Doucette, 1993b) as well as fine grained locking throughout the code. The purpose behind this is to allow highly parallel access to the filesystem. Allowing multiple open transactions allows many processes to be updating the disk at once, a great benefit on large multi processors(Sweeney, 1993d)(Nishimoto, 1994).

For all its speed and parallelism advantages, the size of the XFS code is larger than any other file system discussed here. At about 120,000 lines of code, it is about fifteen times the size of ext3 and five times than of reiserfs. On small scale embedded devices, this size may still matter, but the ever decreasing cost of memory makes this point moot on most systems. Maintainability of such a large code base could become a problem, although the XFS project does not seem to have made any of these problems public.

XFS, like other systems, benefited greatly from a good simulation environment during development(Doucette, 1993d) and this method is echoed in Giampaolo (1999) as a good method to debug core file system code. This method is taken to the extreme in the User Mode Linux project, designed to enable testing of the entire kernel within a user process.

The XFS Namespace(Anderson, 1993b) is that of a traditional UNIX system. The main difference is using B*Trees for large directories. They have shown that optimising for small and large directories leads to increased performance, mainly due to a decreased number of disk reads. The in-node directories are a good example of this.

Like FFS, XFS splits the disk into cylinder (allocation) groups(Doucette, 1993e). XFS does this for increased parallel access to the file system as opposed to FFS's reasoning of increased locality. This works well for XFS as the hardware it was designed to run on is highly parallel and since the XFS transaction mechanism allows for multiple simultaneous transactions, this allows multiple simultaneous disk space allocations and deallocations(Sweeney, 1993d)(Nishimoto, 1994).

The super block of a file system is modified by most transactions and since there is data in the super block that must remain consistent, we have to journal changes to it. If we journal the entire super block, this limits us to commiting transactions serially which will no longer allow the optimisation of the order of transactions being committed to disk. This also becomes a rather obvious bottleneck for parallel file system updates. XFS uses a clever technique to bypass these problems and instead of journaling the super block itself, XFS will journal the changes to super block fields(Sweeney, 1993c).

XFS also supports named meta-data to be associated with files, known as Extended Attributes
(Anderson, 1993a). Each attribute is a name and value pair, with a separate namespace also available for the administrator if they so wish to populate this separately from the user visible namespace. Unlike BeFS, XFS does not offer the ability to index these attributes, and unlike HFS Plus's ability to have arbitrarily sized meta-data streams, XFS limits the size of the meta-data. This severely limits what can be stored as meta-data on an XFS volume, but at the time XFS was being designed not many other systems supported any form of extended attributes.

1.2.8 MacOS and MacOS X's HFS and HFS Plus

The Macintosh's HFS and the updated HFS Plus(Inc., n.d.) volume formats are rather different from the UNIX based filesystems discussed here. It is clear from its data structures that it was designed to support a graphical environment, something which the MacOS was designed to be.

This is evident with the unique forks concept. Each file on a HFS (or HFS Plus) volume has two forks: a data fork and a resource fork. The format and access methods of the Resource Fork are specific to the MacOS and is documented in (**?**). Each fork is a stream of bytes and either fork can be of zero length. This is effectively the inverse of UNIX hard links. Hard links are many names for one stream of bytes, while forks are one name for many streams of bytes.

HFS Plus has an attribute file which is intended to support an arbitrary number of named forks sometime in the future. The goal behind this was to be able to attach extra data and meta-data to files and directories that is moved and removed with that object - an advantage over the more traditional method of adding hidden files to a directory. The MacOS currently only uses two such streams for its traditional Data Fork and Resource Fork.

A common way for MacOS based word processors to store documents was to have plain text in the data fork (so that the raw information could be easily read by other applications, even on other computing platforms) and all the formatting information in a resource in the Resource Fork. The MacOS provided the Resource Manager(**?**) set of APIs to manipulate a simple two-level namespace within the resource fork. It was also common for User Interface specific information to be stored within the resource fork. One such application was to store the name of the program used to create a document so that somebody trying to open it who did not possess the necessary software could be notified of what software they needed to open the document.

In contrast, the Be Operating System (BeOS) used indexed attributes to store and access meta-data for meta-data such as MIME types of files, Artist and Album titles of music files and the status of email messages. The query interface meant that users could perform complex searches of the contents of attributes. The BeOS engineers found that keeping the attributes associated with the file, on the file system as opposed to in a separate "attributes file" was of immense benefit in the general efficiency of the storage system(Giampaolo, 1999).

1.2.9 BSD LFS

The BSD Log Structured File System(**?**) is rather different from the traditional file system. The disk is treated as a log file, with each write being to new disk blocks, never overwriting previously used ones. This approach is believed to increase file system write performance and this is cited as the main reason to use a log based file system.

Although LFS does sequential writes, a set of FFS style index structures are maintained to support efficient random retrieval. The *i-node map* maps i-node numbers to disk addresses. An LFS disk is divided into fixed sized segments, typically of 512kb. When dirty (modified) blocks are to be written to disk, LFS will write a segment, or a *partial segment* (for when there are not enough dirty buffers to fill a segment) to disk. Each segment contains a summary block which contains the i-node and logical block number of each block in the segment. The *ifile* structure (Figure 1.9) is a read-only file on disk which contains the segment usage table and is used by the cleaner.

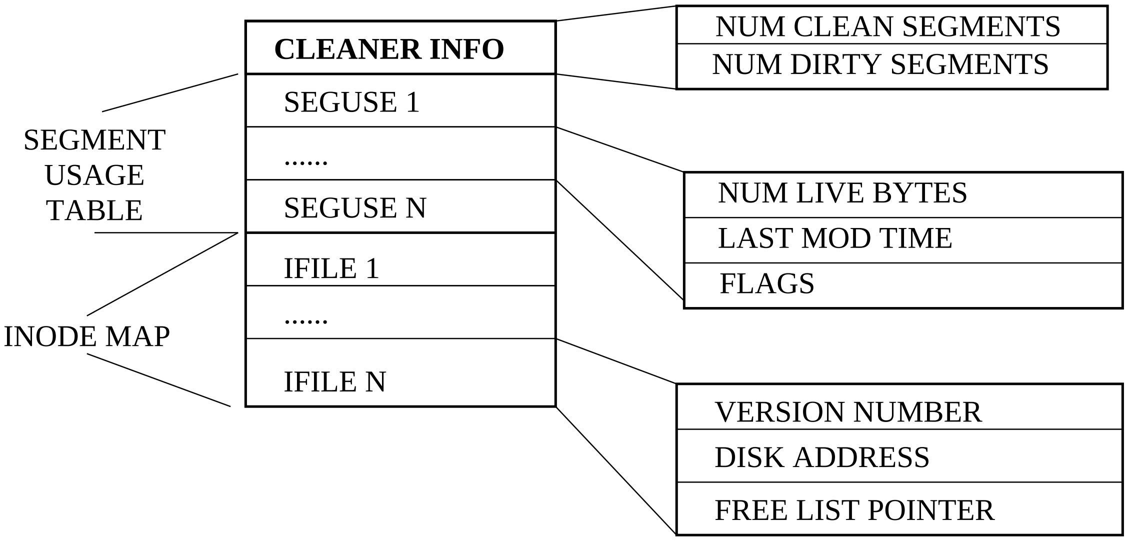


Figure 1.9: BSD LFS IFILE

There is a garbage collection process (the *cleaner*) which will reclaim disk blocks that belong to deleted files or have been superseded by more recent writes. This does lead to a performance hit when the disk is near full as the cleaner must run often. The kernel maintains a *segment usage table* which tracks the last modified time and the number of “live” bytes in each segment; the cleaner uses this table to find segments to clean.

Because of the log structure and the no-overwrite policy, it is possible to implement a facility to request previous revisions of a file. Although the BSD-LFS does not implement this, **?** does indicate that this, among other features could be easily added to LFS.

For an “unrm” (un-remove) utility, the primary problem to be solved would be locating the old inode on disk. It is theorised that this would be a rather trivial problem to solve(**?**).

The **?** paper shows favourable performance benchmarks, not showing any significant performance penalty over the use of FFS. Indeed, in some areas, such as simultaneous random updates, LFS has a large performance advantage over other systems. Real world use of LFS has been rather small, possibly due to peoples general reluctance to switch to something radically different than that which they are used to.

1.3 Revision Tracking

There have been various attempts to integrate revision tracking into the file system. The VMS operating system saved each revision of a file into a separate file with a version number appended to the name (’file;1’,’file;2’ etc). This can (partially) help the user reverse errors such as accidental deletion of files (or part of the content of a file) or wanting to change something back to the way it was. LFS shows that it could be possible to add revision tracking to an existing system without a large performance penalty, but no implementation exists to do this and the LFS user base is rather small despite having existed for nearly 10 years.

The best revision tracking most systems currently hope for is an undelete utility. Traditional undelete utilities for existing file systems are not guaranteed to work reliably or on active file systems(Crane, 1999). Some file systems lend themselves to effective undelete utilities, and others do not. They may have a preference to reuse recently deallocated disk blocks and i-nodes (such as XFS does) or dynamically allocate i-nodes anywhere on disk, making a search for the i-nodes of delete files a very expensive operation.

Many users have resorted to using version control systems such as RCS and CVS(Tichy, 1985) to store and track changes to their documents. Typically, these have required more of an expert knowledge than many end users have, and have not been used widely outside the developer community (who are used to such tools for Source Code Management). More mainstream applications such as Microsoft Word and AbiWord have started to integrate simple graphical interfaces to store and retrieve document revisions, but they have yet to gain common usage. The possibility of integrating such a feature into the file system is desirable as it would mean that revision tracking would be transparent to applications and instantly available to all the tools users are familiar with.

In both Tichy (1985) and MacDonald (n.d.), it is shown that the delta of two file revisions can be computed in reasonable time and stored efficiently. Systems such as WAFL(Hitz et al., n.d.) and LFS(**?**) have provided limited support for snapshots of the entire file system and in the case of WAFL, allowing users to access previous revisions of files. The WAFL approach uses significantly less disk than the VMS approach of revision tracking as WAFL only duplicates blocks that differ (and applicable meta-data blocks) between revisions.

The XDFS(MacDonald, n.d.) approach is rather different to that of WAFL, XDFS stores all deltas between a file's revisions. At any time, any revision of any file on an XDFS volume can be accessed, not just snapshots of the entire system.

1.4 Conclusions

Existing file systems have several things in common:

- the name resolution system is separate from how files are stored on disk
- Files are represented on disk by a structure generally referred to as an i-node containing a small amount of set meta-data and pointers to disk blocks containing file data
- Extensible systems have had features added when needed and remained popular.

There are also several trends in file system design:

- There is a trend to support arbitrary amounts of meta-data
- Speed for large files and large numbers of files is increasingly important
- Quick crash recovery is now a requirement
- A trend towards data journaling to ensure the contents of files are not corrupted after a crash.

References

A Fast File System for UNIX (1984). University of California, Berkeley.

Anderson, C. (1993a). xfs attribute manager design, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Anderson, C. (1993b). xfs namespace manager design, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Anderson, C., Doucette, D., Glover, J., Hu, W., Nishimoto, M., Peck, G. and Sweeney, A. (1993). xfs project architecture, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Anderson, M., Pose, R. D. and Wallace, C. S. (1986). A Password-Capability system, *The Computer Journal* **1-8**(1).

Card, R., Ts'o, T. and Tweedie, S. (n.d.). Design and implementation of the second extended filesystem, *Proceedings of the First Dutch International Symposium on Linux*, number ISBN 90 367 0385 9, Laboratoire MASI — Institut Blaise Pascal and Massachussets Institute of Technology and University of Edinburgh.
*http://web.mit.edu/tytso/www/linux/ext2intro.html

Castro, M. D. (1996). *The Walnut Kernel: A Password-Capability Based Operating System*, PhD thesis, Monash University.

Ceelen, C. (2002). Implementation of an orthogonally persistent l4 microkernel based system, *Technical report*, Universitat Karlsruhe. Supervisor: Cand. Scient and Espen Skoglund.
*http://i30www.ira.uka.de/teaching/thesisdocuments/ceelen_study_studi.pdf

Crane, A. (1999). Ext2 undeletion, *Technical report*.
*http://www.praeclarus.demon.co.uk/tech/e2-undel/

Doucette, D. (1993a). xfs kernel threads support, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Doucette, D. (1993b). xfs message system design, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Doucette, D. (1993c). xfs project description, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Doucette, D. (1993d). xfs simulation environment, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Doucette, D. (1993e). xfs space manager design, *Technical report*, Silicon Graphics.

Elnozahy, E. N., Johnson, D. B. and Zwaenepoel, W. (1992). The performance of consistent checkpointing, *Proceedings of the 11th Symposium on Reliable Distributed Systems*, IEEE Computer Society, Houston, Texas 77251-1892, pp. 39–47.
*http://www.cs.rice.edu/ dbj/ftp/srds92.ps

Folk, M. J. and Zoellick, B. (1987). *File Structures: A Conceptual Toolkit*, Addison-Wesley Publishing Company.

Giampaolo, D. (1999). *Practical FileSystem Design with the Be File System*, Morgan Kaufmann Publishers, Inc., chapter 1.

Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S. and Liedtke, J. (1998). The Mungi single-address-space operating system, *Software Practice and Experience* **28**(9): 901–928.
*citeseer.nj.nec.com/heiser98mungi.html

Hitz, D., Lau, J. and Malcolm, M. (n.d.). File system design for an nfs file server.
*http://www.netapp.com/tech_library/3002.html

III, G. G. R. and Singhal, M. (1993). Using logging and asynchronous checkpointing to implement recoverable distributed shared memory, *Technical report*, Department of Computer and Information Science, The Ohio State University.
*http://camars.kaist.ac.kr/ yjkim/ftsdsm/richard93using.pdf

Inc., A. C. (n.d.). Hfs plus volume format, *Technical report*, Apple Computer Inc. Technical Note TN1150.

Inside Macintosh: More Macintosh Toolbox (1993). Addison-Wesley.

Janssens, B. and Fuchs, W. K. (1993). Relaxing consistency in recoverable distributed shared memory, *Technical report*, Center for Reliable and High Performance Computing Coordinated Science Laboratory, University of Illinois.
*http://citeseer.nj.nec.com/rd/35597259%2C83458%2C1%2C0.25%2CDownload/http://citeseer.n

Kopp, C. (1996). *An i/o and stream inter-process communications library for password capability system*, Master's thesis, Department of Computer Science, Monash University.

Liedtke, J. (1993). A persistent system in real use: Experiences of the first 13 years, *German National Research Center for Computer Science* .
*http://citeseer.nj.nec.com/liedtke93persistent.html

MacDonald, J. P. (n.d.). File system support for delta compression, *Technical report*, University of California at Berkeley.

MacDonald, J., Reiser, H. and Zarochentcev, A. (2002). Reiser4 transaction design document, *Technical report*, Namesys.
*http://www.namesys.com/txn-doc.html

McKusick, M. K. and Ganger, G. R. (1999). Soft updates: A technique for eliminating most synchronous writes in the fast filesystem, *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, The USENIX Association.

Nishimoto, M. (1994). The log manager (xlm), *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Phillips, D. (2001). A directory index for ext2, *Technical report*.
*http://people.nl.linux.org/ phillips/htree/paper/htree.html

Reiser, H. (2001). Reiserfs v.3 whitepaper, *Technical report*, NameSys, 6979 Exeter Dr., Oakland, CA 94611-1625.
*http://www.namesys.com/content_table.html

Reiser, H. (2002-2003). Reiser4, *Technical report*, Namesys. Accessed: 23/4/2003.
*http://www.namesys.com/v4/v4.html

Seltzer, M., Bostic, K., McKusick, M. K. and Staelin, C. (1993). An implementation of a log-structured file system for unix, USENIX.

Skoglund, E., Ceelen, C. and Lidtke, J. (2000). Transparent orthogonal checkpointing through user-level pagers, *Technical report*, System Architecture Group, University of Karlsruhe.
*http://uhlig-langert.de/publications/files/l4-checkpointing.pdf

Smith, S. (2003). Interim honors presentation: The walnut kernel, *Technical report*, Monash University.

Sweeney, A. (1993a). 64 bit file access, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Sweeney, A. (1993b). xfs superblock management, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Sweeney, A. (1993c). xfs transaction mechanism, *Technical report*, Silicon Graphics.

Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M. and Peck, G. (1996). Scalability in the xfs file system, *USENIX Conference*.

The FreeBSD Handbook (1995-2003). The FreeBSD Documentation Project.
*http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/configtuning-disk.html

Tichy, W. F. (1985). RCS - a system for version control, *Software - Practice and Experience* **15**(7): 637–654.
*citeseer.nj.nec.com/tichy91rcs.html

Tweedie, S. C. (1998). Journaling the linux ext2fs filesystem, *Technical report*, LinuxExpo 98.

Wallace, C. and Pose, R. (1990). Charging in a secure environment.
*http://www.csse.monash.edu.au/courseware/cse4333/rdp-material/Bremen-paper.pdf

Wallace, C. S., Pose, R., Castro, M., Kopp, C., Pringle, G., Gunawan, S. and Jan... (2003). Informal discussions relating to the walnut kernel.