VERITAS Software Corporation

File System
White Paper

VERITAS Software Corporation • 1600 Plymouth Street • Mountain View, CA 94043 • 1.800.258.8649 in the USA • 415.335.8000 • FAX 415.335.8050 • E-mail: vx-sales@veritas.com • World Wide Web: http://www.veritas.com/

Table of Contents .....

VERITAS Software Corporation

File System - White Paper

The VERITAS storage management product line has been designed in response to the needs of commercial computing environments. These are the systems that are now supporting mission critical applications and playing a major role in corporate computing services. Typical environments include online transaction processing systems, networked database servers, and high performance file services.

VERITAS specializes in systems storage management and high availability technology which encompasses a product line that offers high performance, availability, data integrity, and integrated, online administration. VERITAS provides three complementary products: VERITAS FirstWatch®, VERITAS Volume Manager™, and the VERITAS File System™.

The VERITAS File System is a high availability, high performance, commercial grade file system providing features such as transaction based journaling, fast recovery, extent-based allocation, and online administrative operations such as backup, resizing, and defragmentation of the file system.

This is the first part in a series of VERITAS White Papers that discuss computer file systems. This first paper provides a discussion of the basic components and mechanics involved in a computer file system. It is intended to be a general discussion on these components, and their interaction.

In this paper we will use a general UNIX file system model for discussion and introduce the basic mechanics behind UNIX file system read and write processes. Where applicable we will discuss certain features of the VERITAS File System as they apply to file system mechanics.

Note: The audience for this paper includes systems engineers, administrators and integrators who desire a breakdown and explanation of the general internals and mechanics of a computer file system. For further reading on this subject, consult other VERITAS White Papers, including the *File System Performance* and *VERITAS File System Performance* White Papers. These papers discuss general file system performance mechanics as well as the performance related internals, mechanics, benchmark test results and analysis, of the VERITAS File System.

A computer file system is the true heart of any information management system. From small personal computers to multiple processor super computers, their respective file systems serve as the point where all the information they process must eventually be stored.

Not only does all of a computer's data reside in its file system, a large part of the computer system's overall value can be defined as the information that resides in the file system. Some information system managers would contend that the **majority** of their system value is stored in their file system.

With this literal wealth residing on some type of electronic media there are several identified needs for information system managers as it applies to their computer's file system. First and foremost is integrity. An underwriting principle for the operation of any computer file system is that any information sent to the file system will be exactly the same when it is retrieved from the file system. A second identified need is availability. Since very few computers can operate without a file system, making sure that a computer file system is available, or at least not creating unavailability as a result of its own standard system operation, is a big concern.

Once you have established your file system's integrity and availability, the last major concern is performance. However, making sure that the computer system can be accessed in a reasonable time is a constantly moving target. To further complicate matters, as computing technology advances, the amount of information stored has increased dramatically. All of this tends to complicate the issues of performance, availability and integrity.

Note: It is evident that the cost of managing data and storage continues to rise. According to Strategic Research Corporation, disk storage requirements are growing at rates exceeding 60%, and the average yearly cost for managing storage is $350,000 for a PC LAN network and $750,000 for a UNIX network. In the March 1995 Strategic Research Corporation report entitled "Managing Network Storage", studies show that for every dollar spent on storage, a company will spend nearly $7 per year managing that same storage.

The key to providing computer file system performance, availability, and integrity is understanding the role and operation of each component in a file system. Like many complex systems, a computer is made up of many interconnected systems, or layers. Each layer performs a specific function, usually in conjunction with other layers. Things that affect one layer will typically affect other layers and impact overall computer operation.

A computer file system is no different. The computer file system consists of multiple system layers which work together to provide file system services to the computer. This white paper will examine each of these layers, their internals and mechanics, using a general UNIX model. Subsequent VERITAS White Papers will use this framework for discussing and understanding key file system

issues such as performance, availability, and integrity.

The VERITAS Product Line

The VERITAS storage management product line has been designed in response to the needs of commercial computing environments. These are the systems that are now supporting mission critical applications, and playing a major role in corporate computing services. Typical environments include online transaction processing systems, both inter as well as intra-networked database servers, and high performance file services.

VERITAS specializes in systems storage management technology which encompasses a product line that offers high performance, availability, data integrity, and integrated, on-line administration. VERITAS provides three complementary products: VERITAS FirstWatch, VERITAS Volume Manager, and the VERITAS File System.

VERITAS FirstWatch is a system and application failure monitoring and management system that provides high-availability for mission-critical applications. FirstWatch dramatically increases server and application availability through the use of duplicate cluster monitoring processes on each node in a FirstWatch system pair. These monitoring processes communicate with each other over dedicated, duplicated heartbeat links.

The VERITAS Volume Manager is a virtual disk management system providing features such as mirroring, striping, disk spanning, hot relocation and I/O analysis. The VERITAS Visual Administrator™ exists as a graphical interface to VERITAS Volume Manager offering visual representation and management of the virtual disk subsystem including drag and drop features and simple or complex device creation.

The VERITAS File System is a high availability, high performance, commercial grade file system providing features such as transaction based journaling, fast recovery, extent-based allocation, and on-line administrative operations such as backup, resizing and defragmentation of the file system.

The similarity in naming between a computer file system and a manual filing system is not accidental. Both systems share similar characteristics. Files can be stored as well as retrieved in either system. In most implementations, in order to speed up access to the actual files, both systems utilize indices for locating the files.

Instead of paper and manila file folders, computer files are usually collections of electronic bits stored on a magnetic medium. Instead of a card index to the location of files, computers use a variety of index tables for the same purpose. These index tables allow the system to retrieve the file needed, by locating its exact position on the disk. This index information in computer file system parlance is often referred to as *metadata*. This metadata contains all the important identifying information about the file, such as location, size, etc.

Like many complex systems, computers are made up of separate layers that interact with one another. The purpose of this layering model is to allow computer system components to operate

interchangeably. This approach allows one model of SCSI hard drive to work with different computer processors and operating systems.

As one layer in a computer system, the file system can be further subdivided into separate layers. Each layer typically receives requests from the layers directly above it, and delivers requests to the layers directly below it. Once the bottom of the file system is reached and the information is retrieved, this request is handed back up the chain to complete the request.

File System Layers

This next section will use an example of a computer file system to illustrate the basic mechanics of the different layers involved. These file system layers are defined here only for the purposes of this tutorial discussion. Other publications may define the file system layers differently, depending upon the specific discussion.

While this discussion is general in nature, it is based on the Sun Microsystems implementation of the UNIX operating system, called Solaris™. As a result, we will present here a general Solaris model for file system mechanics.

We will define the file system layers into the following categories:

- OS Input / Output (I/O) Library
- Type Independent File System
- Type Dependent File System
- Storage Hardware Device Driver
- Storage Hardware

OS Input / Output (I/O) Library

These are the actual operating system calls that either an application or the operating system itself would use to manipulate the file system. Typical I/O calls include file create, file write, file read, file seek, file delete, and file attribute change.

Type Independent File System

In Solaris, the type independent file system is called the Virtual File System, or VFS. In terms of interoperability, implementing a type independent file system layer as part of a file system, allows different type dependent file systems to exist. These are sometimes referred to as installable file systems.

VFS is closely tied to the virtual memory architecture of Solaris. The Solaris system memory is divided into sections, called pages, of equal size. The memory pages used to store file blocks, is sometimes referred to as buffer cache, or disk I/O cache.

When a call is made to the Solaris operating system (OS) for the address of a page the first time, since the address is not in memory, it generates a *page fault*. The Solaris kernel then runs special code called a *fault handler* that takes the address, and converts it into an in-memory index structure called a *vnode*. The vnode basically identifies which file block needs to be retrieved for the calling application. The last thing that the fault handler does is hand the vnode information off to the type dependent file system with the request for that block.

Type Dependent File System

The type dependent file system is the installable portion of the Solaris file system. This is the part of the file system that handles the I/O requests from VFS.

Solaris's legacy file system is the UNIX File System or UFS. This file system was originally developed at the University of California at Berkeley, as part of their work on the Berkeley Software Distribution (BSD) UNIX. Later this file system was enhanced by the same group in an effort to create a file system layout that was more resilient to failure, and had higher performance than prior UNIX file systems. This layout was initially called the Berkeley Fast File System, or FFS, and is now often supplied with UNIX systems as UFS.

The very first thing that UFS does, once handed a VFS request, is to scan the system memory pages, or the buffer cache, and verify whether or not the requested block is already in memory. If it is in memory the page is returned to the calling application. If it is not in memory, then UFS must retrieve it from the secondary storage system.

The type dependent file system, like the type independent system, uses index tables for identifying and locating each file. Some of the tables used by UFS are the *superblock*, and *inode*, and the actual block addressing involves logical and physical blocks. The superblock is a table that contains information about the entire file system, including free block lists, and other static parameters like total file system size, and the file system individual block size. The inode is an index table used by UFS to track things like the file size, the location of the first few blocks of the file, and the date created. Since the information contained in the vnode contains some of the same information contained in an inode, each inode is directly associated with a vnode.

UFS breaks up each file into blocks that are numbered sequentially starting at 0. These are referred to as logical blocks and they represent the smallest unit of a file in the system. Each logical file block is associated with a unique set of physical disk blocks and represent the smallest unit of the physical disk. The disk blocks are numbered sequentially beginning with 0, for each physical disk. This means that every file in a system will have a single logical block numbered 0, while every physical disk will have a single physical disk block numbered 0.

This design accomplishes two things. The first is that it decouples the file blocks from the actual physical disk blocks used for the file. This allows a variety of allocation methods to be utilized. The second thing that is accomplished is that this allows the logical file block size to be different from the physical disk block size.

The VERITAS File System is an example of an installable, type dependent file system. The VERITAS File System currently supports many commercial implementations of UNIX including Solaris.

Storage Hardware Device Driver

The storage hardware device driver is generally the last piece of operating system software to handle a file system request. These device drivers are specific to a particular hardware device, and are usually optimized for that device.

Whenever an I/O call is received by the device driver it usually includes the following information:

- Whether the operation is for storage (input) or retrieval (output)
- The disk address (this is the original requested block number translated by the type dependent file system into drive, cylinder, surface, and sector coordinates)

- The memory address to copy to or from
- The amount of information to be transferred

In order to calculate the disk address, the type dependent file system takes information from the inode and runs a UNIX process that returns the physical block address. Remember that logical blocks are numbered unique only for each file, while physical blocks are uniquely numbered for each disk.

Once the physical block address is calculated, the actual I/O can begin.

Storage Hardware

The storage hardware is physically where everything gets stored. For the purposes of our discussion we will use the industry standard secondary storage example of hard drives as our example.

Accessing the disks is done through a disk controller. The controller along with the disks, make up the disk channel. There are many different kinds of disk channel technologies, including MFM or Modified Frequency Modulation, RLL or Run Length Limited, ESDI or Enhanced Small Device Interface, IDE or Integrated Device Electronics, ATA or Advanced Technology Attachment, and SCSI which stands for Small Computer Standard Interface.

Most open platform enterprise servers use SCSI as their standard secondary storage disk channel. The SCSI technology is further subdivided into successive specifications. These are commonly referred to as SCSI or SCSI-1, SCSI-2, and SCSI-3 specification. Each specification revision basically adds performance and configuration improvements.

Since most disk controller technologies allow for multiple physical disks to be supported via one disk channel, each physical disk in a single channel is given a unique number. Each disk also consists of multiple disk platters. Each platter has two sides, or surfaces, for storage. So each surface is identified with a unique number for each disk.

Finally information on each disk surface is stored in concentric circular patterns called tracks. Each track is further subdivided into units called sectors. The total tracks (on all surfaces) that can be accessed without movement of the drive head constitutes a cylinder. Depending on the disk drive size, sectors can vary from 32 bytes to 4096 bytes, usually being 512 bytes. As mentioned previously, I/O transfers between system memory and the disks are performed in units of one or more sectors, described here as physical disk blocks.

Physical disk blocks are usually referenced by a multi-part address, which includes the drive number, the surface, the track, and the sector. In order to access a block you need to provide an address in this multi-part format.

The final stage of a file system's I/O request is passed to the storage device in this manner. The request sent to the disk is a combination of whether it is a read or write operation, which physical disk address, which system memory address to copy to or from, and finally how much information is to be transferred.

UNIX File Reads

Putting all these layers together allows us to examine a file system read process from beginning to end. For this example we will use a database application, Oracle, running on a Sun UNIX workstation, with local storage.

1) The process begins when the user issues a request for Oracle to read in a record. Oracle will first check to see if the record required is in its own application memory space.

Note: Oracle maintains and indexes its own memory space separate from the operating system's memory pages. This is typical of applications, especially those applications who want to fine tune the memory cache for optimal performance. Oracle's memory space is called the System Global Area (SGA).

2) Once Oracle finds out that the record is not in memory it will invoke a Solaris I/O call to read the logical file block that contains the record into the Solaris system buffer.

2a) As discussed previously this generates a page fault which invokes the specific fault handler.

3) The fault handler takes the I/O request, and converts it into a vnode along with an offset address. This information identifies which file block from the file needs to be retrieved and copied into a memory.

4) The last thing that the fault handler does is hand the vnode off to the type dependent file system with a request to get that logical block.

5) Once handed a VFS request for a specific logical file block, UFS scans the system buffer cache, and verifies whether or not the vnode requested block is in memory.

5a) If it is in memory, the page is returned to Oracle and Oracle will copy the contents of the page into its SGA pages, using a memory to memory copy.

5b) If it's not in memory, then UFS must retrieve it from the secondary storage system.

Note: It is important to note here that while the type independent file system defines vnodes and the memory pages for caching the file blocks, it is the type dependent file system that actually manages the process of retrieving the page, or retrieving the page address, from the system cache.

6) UFS next converts the logical file block address to a physical block address using information from the file's inode, in order to correctly format the actual disk I/O request.

7) UFS then sends a read I/O request to the disk driver with the physical block address and the memory address to copy to.

8) The device driver schedules the request to be done.

9) The storage system finds the physical block and transfers their content to the system memory

page specified.

10) The fault handler routine finishes once the appropriate file block has been copied to the memory page.

11) VFS then returns the memory page address to the calling application, Oracle.

12) Oracle copies the contents of the page into its SGA pages, using a memory to memory copy.

UNIX File Writes

There are many similarities between the UNIX read and write processes. In some instances prior to issuing a system write, the UNIX OS will issue a system read on the same block in order to ensure updating the latest version of the file block. The processes used to calculate logical and physical block addresses are the same between reads and writes. There are also a numbers of indices to the file system that are used in similar ways between reads and writes. The biggest difference between system reads and writes is that the write operation makes changes to the indices.

Another difference between system reads and writes is that the write operation is greatly impacted by the way in which disk space is allocated. In order to do this efficiently the operating system maintains several indices as mentioned previously, collectively referred to as *metadata*. Rather than describe the step by step process of a UNIX File System write this next section will provide information on the various metadata tables that are used in a typical UNIX file system, along with several disk allocation methods.

Metadata Tables

One important metadata index is the one which keeps track of the available disk blocks, generically referred to as a *free block list*. In UNIX free block lists are usually implemented as a set of linked lists, and stored in a table called the superblock. The superblock is unique to each file system and contains static parameters including total file system size, the file system individual block size, and assorted parameters that affect the allocation policies.

Another important index is the file system directory. The file system directory is the user's logical view of the file system's contents. This is represented as a tree structured directory where the individual files are stored. Each directory is represented by an inode, in the same manner as a file, in the file system. When a file is created, (or read), its location in the directory is defined either explicitly or implicitly. When searching for a file or directory, the file system traverses the directory structure by doing directory to inode translations.

For performance reasons, the superblock is always stored in system memory and synchronized with the on disk copy via a background process running in the UNIX system. Additionally, newer versions of UNIX added a *directory name cache* to speed up the process of directory to inode translations by keeping the most recent translations in memory. In fact in many UNIX implementations, there are a number of metadata indices that are always kept in system memory and synchronized with their on disk copies, in order to improve overall file system performance.

When a UNIX system performs a file write, it must access and update all of the relevant metadata tables. As mentioned above, these writes are oftentimes buffered to improve performance. This buffering can provide performance increases due to the fact that metadata changes occur frequently in file systems. However this design also decreases data integrity. If the system goes down prior to metadata changes actually being written to the disk, the system, once restarted, can suffer corrupted file information, and corrupted file system structure (metadata) information.

Note: When a UFS file system is in the process of a file write, the writes are buffered in the system cache. Once the writes are stored in memory, they are eventually synchronized on disk. If the computer is interrupted before this synchronization is completed, then the system, upon restart, must first manually check every disk block against the on disk free block list and reject any incomplete operations. This is done with the UNIX utility *fsck*, (file system check / repair).

Journaling

This problem of increasing performance while decreasing file system integrity creates a design challenge that is addressed with something called file system logging, or journaling.

File system logging is a technique which involves committing system writes to a sequential log file. The benefit is that the writes are stored on disk, not in system memory, and the sequential nature in which they are written speeds up disk write activity. File system log implementations differ in what types of system writes are stored in the log, some storing just the file system structure information or metadata, others storing all metadata and data in the log. Implementations also differ in whether the log is permanent, or a temporary space, with the files eventually being written to their permanent disk blocks.

The VERITAS File System employs a variation on the general logging technique by employing a circular *intent log*. All file system structure changes, or metadata changes, are written to this intent log in a synchronous manner. The file system will then periodically flush these changes out to their actual disk blocks.

Journaling has a tremendous impact on file system availability. In a normal UNIX environment the metadata changes are stored in the system memory buffer cache. If the computer were to go down prior to the changes being written to disk, the system would have to perform a file system check, using the utility *fsck*. This utility scans all of the data blocks including the metadata structures to determine the correctness of their information. If a disk block had been marked as in use, when in fact it was not, fsck would update the appropriate metadata table, and add the disk block to the list of free disk blocks. This is a very time intensive process due to the fact that every block must be examined.

By using an intent log, the VERITAS File System can recover from system downtime in a fraction of the time. When the VERITAS File System is restarted in the same scenario, the system simply scans the intent log, noting which file system changes had completed and which had not, and proceed accordingly. In some cases, the VERITAS File System can roll forward changes to the metadata structures, because the changes were saved in the intent log. This adds a great degree of both availability and integrity to the overall file system. Some older block based file system implementations have added journaling technology. The VERITAS File system differs from these in that it is designed as a journaling file system.

Disk Allocation

As mentioned in the previous section, the allocation of disk blocks for the storing of files is an important concept for understanding not only how file systems operate, but also how file system performance is affected by allocation policies.

As we described previously, disks are divided into units called physical disk blocks. The allocation of disk blocks for file storage greatly impacts file system performance. There are several methods for allocating disk blocks. The most commonly used disk allocation methods are known as *contiguous* , *linked*, and *indexed* allocation.

Contiguous Allocation

The *contiguous allocation* method allocates file space on a contiguous basis. This means that all files are allocated disk blocks in a contiguous manner. A file would be allocated their first block, as disk block *n*, and then be allocated all subsequent blocks until the file was completely stored. This would mean that disk blocks n+1, n+2, n+3, etc. would be occupied by the one file. The largest advantage of contiguous allocation is that since all blocks of a file are located next to each other, retrieval of the file requires much less disk head activity. The disk seek times are greatly reduced using this method of allocation.

There are two problems with this type of allocation. The first involves something that affects all allocation policies and that is once a file has been deleted from the system, “holes” begin to appear in the disk block allocation map. This is commonly know as *external fragmentation*. In a contiguously allocated file system, if there is not a big enough contiguous set of disk blocks for a file, the file system will return with an error stating that there is not enough disk space available. At this point the user must intervene to defragment the file system. This is done by reorganizing all the current files on disk, to reclaim all the file system “holes” into a single large set of free disk blocks. In most cases this requires taking the system off-line to perform this operation, thereby making the system unavailable to users.

The second problem with contiguous allocation comes when trying to determine the size of the allocation. For example, if the initial allocation does not have enough empty blocks on either side of it to accommodate growing the file, then when the system attempts to grow the file allocation, the system will typically look for another larger set of free disk blocks, and copy the entire existing file with the new allocations to this larger space. This results in performance slowdowns as the OS manages the disk space in this manner. If the initial allocation is too large for the actual file stored, this will result in *internal fragmentation* where disk space is wasted being allocated to a file and not storing any file data. IBM’s VM/CMS is an example of an OS that uses contiguous allocation.

Linked Allocation

Linked allocation is a method that does not require contiguous allocation of disk blocks. Instead the first block in a file, contains a pointer to the next block in the file. This allows files to be allocated, grown, and shrunk, by having at least the minimum required free disk blocks be available anywhere on the disk. This type of allocation works around external fragmentation by being able to use any free blocks on the disk. However external fragmentation has a great impact on performance in this system. If disk blocks are allocated randomly across the disk surface, the system takes up more time in searching through a file, as each pointer in the file’s disk blocks must be followed until the required block is found. Also performance is impacted due to the fact that the disk heads must move to random points on the disk surface to follow the file links. This type of file allocation also requires that more disk space be used up by the pointers themselves, allowing less disk space for files. MS-DOS and OS/2 use a modified version of linked allocation, using a table for more efficient file link searches, called a File Allocation Table, or FAT.

Indexed Allocation

The last method of allocation is the one employed in the UNIX OS, and is called *indexed allocation*. In an indexed allocation system every file is associated with an index block. This block contains the pointers to the actual disk blocks used to store the file. This provides a performance boost over linked allocation by allowing a file’s disk blocks to be accessed directly by simply accessing their address in the index block. This type of allocation does incur disk overhead for the storing of the index blocks but it typically is less overhead than a comparable linked allocation system. This is due to the fact that in an indexed allocation system one index block points to multiple physical disk blocks, as opposed to a linked allocation system where each file block contains a single pointer to another file block.

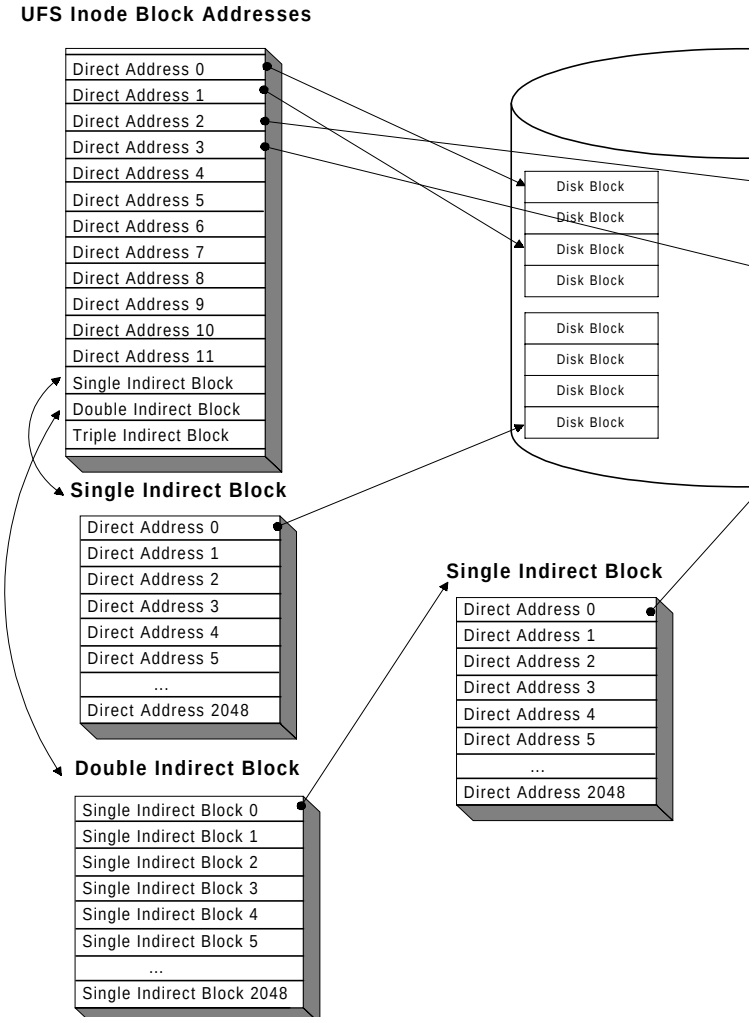
Indexed allocation also works around external fragmentation by allowing the use of any free blocks

on the disk. However external fragmentation also impacts the performance of this type of system due to the fact that as a file's blocks are stored farther and farther away from each other, more and more seek time is introduced into the system.

UNIX's Indexed Allocation

UNIX's implementation of indexed allocation rests with their index block or inode. The UNIX inode contains information about the file including a list of physical disk blocks where the file resides. Each inode can store 15 addresses, or pointers, for these physical disk blocks. The first 12 of these pointers point to direct blocks, these are the addresses of physical disk blocks where the data for the file is stored. If a file system is using a 4K block size, then up to 48K of the data can be accessed directly from this first, or primary, inode.

The next 3 pointers in the primary inode point to *indirect blocks*. This allows the UNIX system to support and address larger files. The first indirect block pointer provides the physical disk address of a *single indirect block*. This block does not contain any of the file's data, but like the first 12 pointers in the primary inode, it contains pointers to the physical blocks that do. The second indirect block pointer provides the physical disk address of a *double indirect block*. This double indirect block contains addresses of single indirect blocks, which again contain pointers to the files' physical disk blocks. The third indirect block pointer is not currently used.



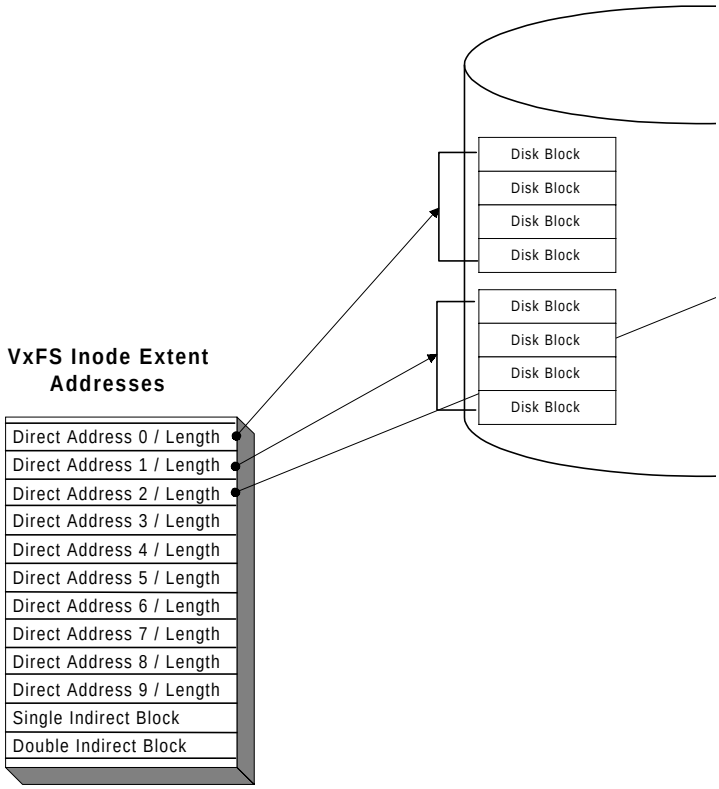
Using this method of block addressing with a 8K block file system, UNIX could logically address files up to 32GB in length. Since the file offset in UNIX is kept in a 32-bit word, and since the file pointers are signed integers, for seeking backward and forward in a file, the actual maximum file size in UNIX is currently 2^{32-1} bytes, or 2GB.

Note: While all methods of allocation have certain advantages and disadvantages it is important to note that contiguous types of disk allocation typically provides the highest throughput performance for file systems.

Extent Based Allocation

Extent based allocation takes the contiguous allocation concept a step further. Extent based allocation is actually a modified contiguous / indexed allocation method. This method capitalizes on the performance advantages of the contiguous allocation method while avoiding several of its drawbacks.

Extent based allocation works by allocating a large contiguous set of disk blocks initially when a file is created. This is known as an extent. In most implementations there is an index block associated with the extent allocation, but unlike the UNIX inode, an extent based inode simply needs to have a pointer to the first block of the extent, and then a size. This lets the file system directly access any block of the file no matter how large the file.



When the file needs to grow larger than the extent size, the operating system allocates another extent of disk blocks and in some implementations the inode is updated to point to the first block of the next extent, along with its size. This type of extent based allocation has a number of direct benefits. One is that this method can address files of any supported size, directly and efficiently. Another benefit is that this method requires fewer pointers and less indirection to access data in large files.

The biggest detraction given for extent based allocation methods is that the extent allocation size can lead to inefficiencies in the file system. Choosing the right extent size is important for an extent based allocation system. If the size is too small, then the system will lose some of the performance benefits of extent allocations, and act more like an indexed allocation system. If the extents are too large, the file system will have disk space allocated that is not actually in use, wasting disk space. This is known as *internal fragmentation*. In order to address this issue, some extent based systems implement a method for choosing the extent size based on the I/O pattern of the file itself.

The VERITAS File System is an example of an extent based file system.

File System Administration

Commercial deployments of computers typically demand availability at very high rates. As computer file systems have been employed in more environments that require commercial levels of availability, several aspects of the administration of file systems have been shown to be unsatisfactory. Commercial system availability is quickly approaching continuous uptime in more and more implementations. This is sometimes referred to as 24 by 7 availability, defined as the system being available 24 hours a day, 7 days a week, 365 days a year.

Commercial environments that require this level of availability exceeds the traditional levels achieved by typical UNIX systems. While disk array redundancy techniques address issues of unscheduled downtime caused by system or disk failure, they do nothing to address scheduled or user-driven downtime issues. In order to make 24/7/365 operation schedules feasible under UNIX, it is necessary to provide tools to perform common administrative and corrective tasks without interrupting access. These tasks range from consistent backup to defragmentation to increasing or reclaiming space.

Stable Backup Support

Non-stop operation poses significant problems for administrators who wish to implement a backup-recovery strategy. Without some consistency enforcement, backups performed while users are accessing data may have structural inconsistencies in the file system, and data inconsistencies between blocks of a file, between files in a file system, and between files in different file systems.

The traditional technique for addressing this problem is to back up data at scheduled down periods. However, for some applications, any appreciable downtime may be unacceptable. Administrators have often chosen to perform backups at known low-activity periods and run the risk of inconsistency, but this is generally unacceptable as well. Several other techniques are available to facilitate data stability.

The VERITAS Volume Manager and the VERITAS File System incorporate the first and third methods for achieving stable backup support described here.

The first common technique for stabilizing backup involves using a stabilized mirror of the data, while one or more instances of the data remain accessible.

There are, however, several deficiencies to this method:

- Backing up a mirror requires an extra mirror. If the data is initially unmirrored, this requires finding enough free space to create an additional copy; if it is already mirrored, a mirror may be taken off-line, but redundancy will be reduced and the data will be more vulnerable to loss.
- The mirrored copy is potentially unstable; outstanding writes to the file system may still be in the page cache. While a sync operation will flush these blocks, there is no way to determine that new operations will not occur between the sync and the actual disabling, or snapping, of the mirror.

The VERITAS Volume Manager / VERITAS File System combination supports a freeze/thaw interface which addresses this issue. Before snapping the mirror, the VERITAS Volume Manager instructs the VERITAS File System to flush all outstanding writes and hold all pending writes. After the mirror is snapped, the I/O is allowed to continue, yielding a stable copy.

- The additional mirror requires either initial synchronization (if it is an additional one) or resynchronization (if it is a “borrowed” permanent one).

The second technique uses locks to disable some or all operations on some or all parts of the file system during backup. While this method has the advantage of not requiring additional storage, it is also very limited in utility. Its drawbacks are:

- Use of locks disables access for some operations during the progress of a backup. While a file is being backed up, it may not be changed. Most metadata operations are inhibited during the backup phases that archive the metadata. Some metadata operations (rename, for example) are inhibited for the duration of the backup.

- Locking does nothing to facilitate consistency between the contents of files, whether on the same or different file systems. Where the temporal relationship between the data in multiple files is significant, locking cannot guarantee application consistency.

The third method which may be used for providing stable backups is called a *snapshot*. Snapshots are virtual stable read-only copies of a file system, which appear to be separate trees or file systems. When a snapshot is created, all subsequent writes will modify the data in the live file system, allowing application access to continue. The initial access to each block after the snapshot is established will cause a copy of the original block to be stored in a holding space. Access to the live file system will see up-to-date data, and access to the virtual snapshot copy will see a stable view of the file system at the instant the snapshot was established. When the backup is complete, the snapshot can be removed, and the snapshot holding space reallocated.

VxFS supports stable snapshots.

This method requires additional space to store only the blocks which change while the snapshot is in effect. But this is generally 2% to 20% of the total data space, depending on the file system size and activity level, rather than a full second copy. For total stability, application access need only be interrupted for the few seconds it takes to establish the snapshot. This method guarantees the consistency of all data and metadata in a single file system; if multiple file systems must be backed up with mutual consistency, they must be stabilized at the same time before establishing their snapshots.

The deficiencies of this method are:

- Some additional storage is required.

- Backup and restore utilities must be capable of backing up data from one location (the snapshot point) and restoring it to another (the actual file system).

Most common backup software, including the VERITAS utilities vxdump and vxrestore, can do this.

- Reads through the snapshot and initial writes of a block are slower than reads and writes of a live file system without snapshot.

It seems that the snapshot method offers the best opportunity for combining stable backup and non-stop operation, while adding minimal overhead.

Another interesting use of snapshot technology is the establishment of frequent snapshots to implement a user-driven recovery or “undelete” capability. For instance, if file server directories have a snapshot established every two to three hours and left in effect for a day or two, it is very simple for a user who accidentally deletes a file, or modifies it incorrectly without keeping a copy of the original version, to “roll back” changes by using standard UNIX utilities to copy the file back from the snapshot

to the live file system.

As administrators in most file server and time-sharing environments report that the major requirement for backup recovery is not to address media or system failure, but rather, to respond to user error, this snapshot use is an attractive option.

In conjunction with database files, the snapshot mechanism may be used to enhance concurrent access by mixed communities of transactional and batch updaters and read-only queries and reports. This is a common job mix, involving operational and decision support users, and often creates problems when long-running queries and reports cause update transactions to be rolled back unnecessarily.

If a snapshot is established, and read-only applications are invoked against the snapshot instance of the database, there is no conflict between the decision support users, who see a stable and consistent database version without the overhead of additional locking, and the operational users, who are freed from the nuisance of rerunning their applications because of unwanted rollbacks.

On-line Resizing

As application and user community data requirements change and grow, so do the characteristics of the file systems they use. Traditional methods of disk and file system allocation have led administrators to configure file systems that were bounded by disk sizes, and to attempt to balance capacity and I/O requirements between them. Of course, this is not a very predictable situation, and often results in imbalances in capacity (excess unused space in one file system and frequent capacity crises in another), or in activity (I/O bottlenecks in one file system, with unused bandwidth in another).

In order to address this problem, administrators would need to move users and directory trees between disks. This is a manual process, and is likely to be only a temporary correction. It also requires some interruption of access for the users whose files are being relocated. The file system itself must be capable of supporting a resizing capability for this to be effective.

Virtual disk managers like the VERITAS Volume Manager allow the growth and shrinkage of devices. The VERITAS File System supports file system growth and shrinkage, though shrinkage may not be possible for specific file systems if structures close to the end of the file system are already allocated. It may be done on-line with no interruption or locking of access.

On-line Defragmentation

As files are created, extended, truncated, and removed in a file system, the storage used for each file may become fragmented. This inhibits effective file I/O. Without an on-line defragmentation utility, the only way to restore efficient layout is to back the file system up, recreate it, and reload it. This yields unacceptable downtime.

While a file system that uses extent based allocation in general creates a more contiguously allocated file system, the file system itself can still suffer from fragmentation under some circumstances.

The VERITAS File System provides an on-line defragmentation utility that may safely be run on live file systems without interfering with user access. It brings the fragmented extents of files closer together, groups them by type and frequency of access, and compacts and sorts directories. This utility, which is typically run as a recurring scheduled job, is an effective tool for the management of a high-performance on-line file store. Even if database software used on top of the file system has its own defragmenter, this additional defragmentation is necessary to make the storage that the database engine sees as contiguous more truly so.

A computer's file system has traditionally been a very cost intensive component of the overall computer system. This trend along with overall storage demands are increasing for commercial server environments. The file system is also the component that represents a significant amount of the overall value of a computer system. This is evidenced by the fact that a system's data is the one component of a computer system that is continually grown, not replaced. While computer CPUs, memory, and software are continually upgraded and replaced, a company's data is always brought forward and grown, not replaced.

We have illustrated that a computer file system is composed of multiple layers interacting with one another to perform computer file I/O. Each layer in a computer file system is responsible for a specific set of functions, and each layer typically operates by receiving requests from one specific layer, processing the request, and then handing the result to another specific layer.

Finally, while some areas of the computer file system industry have scaled well to increasing user demands, for example disk drive manufacturers have steadily increased the size of disk drives, other areas of the industry have not kept pace. Even with their advancements, operating system vendors have been slow to improve the scalability of their legacy file system software components with backwards compatibility being a large factor.

The VERITAS Software File System fills this gap to provide scaleable, commercial class, next generation file system technology.

For further file system technical information consult other VERITAS White Papers, such as the *File System Performance* and *VERITAS File System Performance* white papers, available from VERITAS Software.