Single Instance Storage in Windows® 2000

by William J. Bolosky, Scott Corbin, David Goebel*, and John R. Douceur
*Microsoft Research, *Balder Technology Group, Inc.*
{bolosky, scottc, v-davidg, johndo}@microsoft.com

Abstract

Certain applications, such as Windows 2000's Remote Install service, can result in a set of files in which many different files have the same content. Using a traditional file system to store these files separately results in excessive use of disk and main memory file cache space. Using hard or symbolic links would eliminate the excess resource requirements, but changes the semantics of having separate files, in that updates to one "copy" of a file would be visible to users of another "copy." We describe the Single Instance Store (SIS), a component within Windows® 2000 that implements links with the semantics of copies for files stored on a Windows 2000 NTFS volume. SIS uses copy-on-close to implement the copy semantics of its links. SIS is structured as a file system filter driver that implements links and a user level service that detects duplicate files and reports them to the filter for conversion into links. Because SIS links are semantically identical to separate files, SIS creates them automatically when it detects files with duplicate contents. This paper describes the design and implementation of SIS in detail, briefly presents measurements of a remote install server showing a 58% disk space savings by using SIS, and discusses other possible uses of SIS.

1. Introduction

Some applications generate many files that have identical content. These files are separate from one another, in the sense that they may have different path names, owners, access control lists, and may charge different users' disk allocation quotas. Most importantly, because the files are separate, writes to one file do not affect any other files. However, the fact that the files have identical contents presents an opportunity for the file system to save space on the disk and in the main memory file cache. The Single Instance Store (SIS) is a pair of components in Microsoft® Windows® 2000 Server [Solomon 98] that automatically takes advantage of this opportunity. This paper describes the design and implementation of SIS, and discusses its use in Windows 2000 as well as other potential uses for the technology.

SIS is used in Windows 2000 to support the Remote Install Server [Microsoft 00]. Remote Install is an application that allows a server owner to configure a server with a set of machine images (installations of an operating system and an arbitrary set of applications), and to use these images to set up new machines quickly, without having to go through the usual installation processes. Different machine images stored on a remote install server will have different files, but there will be some duplicate files in the images. Running SIS on a remote install server removes space used by the duplicate files, and allows a server to host more images with the same amount of disk and main memory file cache space.

SIS is built in two components. First is a kernel-level file system filter driver (called the *SIS filter* or just the *filter*) that transparently implements files that have identical contents but are stored only once on the disk; second is a user-level service (called the *groveler*) that is responsible for automatically finding identical files and reporting them to the filter for merging. The filter's basic technique is to handle reads by redirecting them to common files, and to handle writes using copy-on-close. The groveler runs as a low-importance service (daemon) that tracks changes to the file system, maintains a database of hashes of files in the file system, checks files with matching hashes to see if they are identical, and reports matching files to the filter for merging. Unlike traditional file linking, SIS (via the groveler) automatically finds and merges files with duplicate contents. This is sensible in SIS because SIS links (unlike hard or symbolic links) are semantically identical to independent files.

SIS's copy-on-close technique is similar to copy-on-write, which has been used in various forms in computer systems for quite some time, most notably in virtual memory [Rashid 81] and database [Todd 96] systems. In copy-on-write, at the time of a "copy" a link between the source and destination is established, and the actual copying of the data is postponed until either the source or destination is modified. Crucial to the concept of copy-on-write is that it is semantically identical to a normal copy, unlike linked-file or shared memory techniques. SIS's copy-on-close differs from copy-on-write in that the copy is delayed beyond even the time of the first write until the complete set of updates are made to the

المنارة للاستشارات
www.manaraa.com

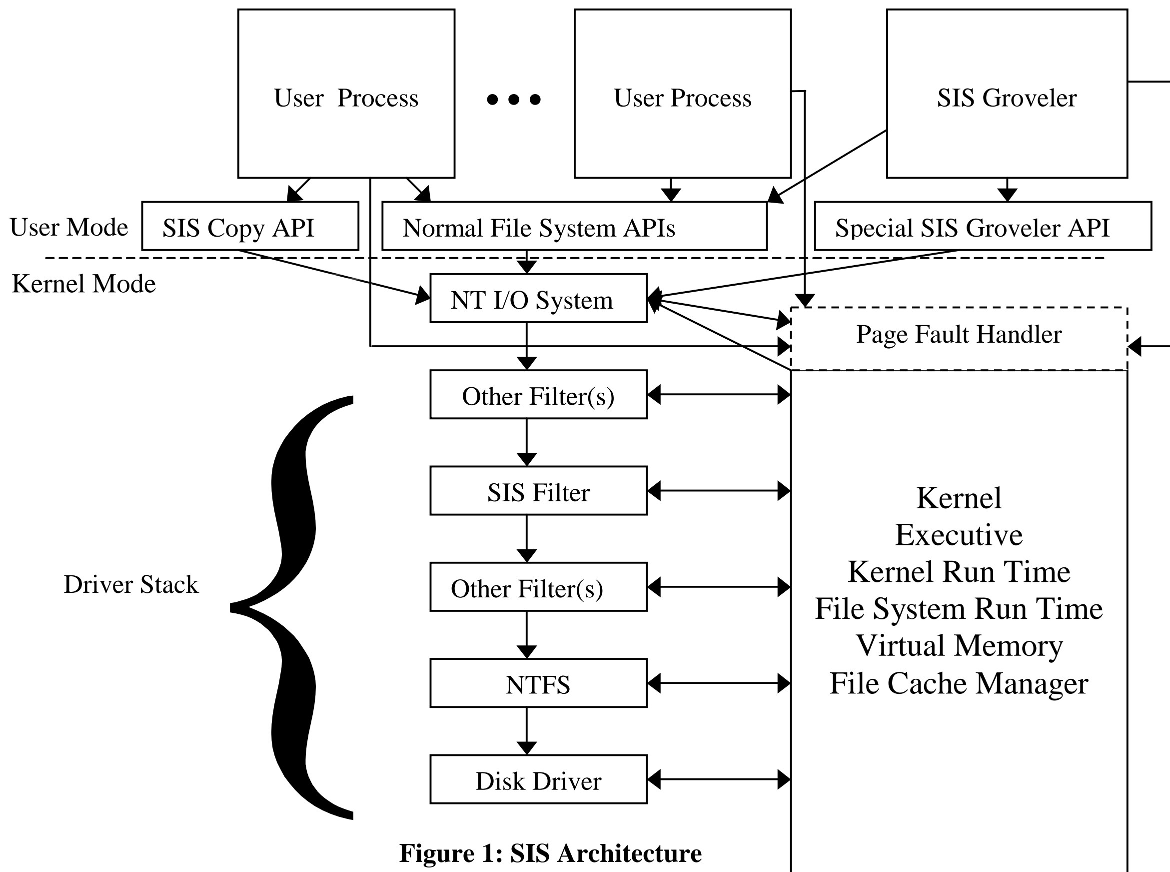

Figure 1: SIS Architecture

file, and then only the portions of the file that haven't been overwritten are copied. This has two advantages over copy-on-write: there is no delay for the copy at the time of the first write, and there is no need to copy the portions of the file that are overwritten.

While saving disk space is valuable, disk storage has been rapidly dropping in price per byte. Some may argue that the disk space savings afforded by SIS, which in most cases will be much less than an order of magnitude, are uninteresting for all but the most space-consuming applications. While this may be true, the reduction in size of the main memory file cache can have large performance effects. This will become more pronounced as the ratio of processor, memory and network speeds to disk latency increases.

The next section presents some background describing features in Windows 2000 on which SIS depends, and then goes on to describe SIS's architecture and implementation in detail. Section 3 briefly presents some performance measurements relating to the time and size overheads of SIS, and the disk space savings that was realized on a remote install server. The section then discusses some potential uses for SIS beyond the remote install server. Section 4 presents related work, section 5 summarizes and section 6 describes the availability of the software and raw data.

2. SIS Architecture and Implementation

SIS has two responsibilities: implementing SIS links, and discovering files with identical content in order to merge them. These two functions are provided by a kernel-level file system filter driver and a user-level service (daemon) respectively. Figure 1 shows the top-level architecture of SIS. This section lays out the basic concepts and terms used in describing SIS and its design, provides brief background on the Windows NT underpinnings used by SIS, and then describes the SIS architecture and design in detail.

A user file managed by SIS is called a *SIS link*. The SIS filter (the kernel mode portion of SIS) is responsible for assuring that users see appropriate behavior when accessing SIS links. The filter keeps the data that backs SIS links in files in a special directory called the *SIS Common Store*. SIS links may be created in two ways: A user may explicitly request a SIS copy of a file by issuing the SIS_COPYFILE file system control, or SIS may detect that two files have identical contents and merge them. The groveler (so called because it grovels through

المنارة للاستشارات
www.manaraa.com

the file system contents) detects duplicate files. All SIS functions are local to a particular NTFS volume, which may be accessed remotely in the same way as any other NTFS volume.

The rest of this section describes how SIS is designed and constructed. The first two subsections provide necessary background about unusual Windows 2000 and NTFS [Custer 94] facilities used in the implementation. Section 2.3 describes the SIS filter in detail, and section 2.4 covers the groveler.

2.1 The Windows 2000 Driver Model

The privileged mode portion of Windows 2000 consists of a base kernel containing services such as scheduling, virtual memory support, thread and multi-processor synchronization, etc.; and a set of loadable *drivers* [Solomon 98;Baker 97;Nagar 97]. Windows 2000 drivers may have more broad functionality than device drivers in traditional systems. They implement file systems, network protocol stacks, RAID/Mirroring disk functionality [Patterson 88], virus protection, instrumentation, off-line file migration, and other similar functions as well as simply operating normal devices [Fisher 98]. Groups of these drivers are called *stacks*, although that term can be somewhat misleading, because their organization is not necessarily linear (imagine a RAID driver that talks to more than one disk driver). A driver that is inserted between the NT I/O system and the base file system driver is called a *file system filter driver* (or sometimes just a *filter*). There are also other kinds of filter drivers that sit below the file system, but they are not relevant here.

To first approximation, NT driver stacks work by passing around I/O Request Packets (IRPs), which are requests to do a specific operation, such as read, write, open or close. For instance, a file system driver might receive a read IRP specifying that a range of a file should be read into a certain virtual address in a process. The file system would use its metadata to find the region(s) on disk holding the data for the given part of the file, modify the IRP to tell the disk driver what action to take, and then send the IRP down the stack to the disk driver, which would perform the actual I/O. When a driver marks an IRP as completed, any drivers above it on the stack have the opportunity to inspect the IRP, see if it completed successfully, and take action including aborting the IRP completion or changing the completion status.

The NT I/O manager sits at the top of all file system driver stacks. It accepts system calls from user processes (or function calls from kernel-level components), translates handles into pointers to file objects, generates IRPs and sends them to the top member of the appropriate driver stack. After the IRP completes, the I/O manager completes the initial request by completing the system or function call, or by other appropriate means in the case of asynchronous calls. The I/O Manager (and SIS) also support the FastIO function-call driver interface, but that is beyond the scope of this paper.

Windows 2000 supports memory mapped access to files. In this mode, a process or system component asks the system to map a portion of a file to a region of virtual memory. When the process or component accesses a virtual address in a mapped range, it may take a page fault, which will result in the virtual memory manager generating an IRP for reading the appropriate data into a page, and sending this request to the file system driver stack. This IRP is marked to indicate that it was generated by a page fault and that the file system should read data from the disk, rather than trying to obtain it from the system file cache. Once the file system completes the request, the virtual memory manager maps the page at the appropriate virtual address in the process or system address space and restarts the faulting thread. When an access to a memory mapped file doesn't result in a fault, the system is not immediately aware of it. This has an important implication for SIS: If a user maps a page, takes a read fault on the page and then later writes to it, SIS will not have any way of knowing that the page has been written, and so will not be able to take any consistency actions at the time of the write. Eventually, the virtual memory system will notice that the page is dirty and write it to disk, but the delay may be large. A different memory manager could mark such a clean, mapped page read-only and send a notification to SIS when a write happens (much like the technique used in distributed shared memory systems [Li 86] or Accent and Mach [Rashid 81; Accetta 86]), but there is no support for this in the NT memory manager.

The NT cache manager is a system component that maintains an in-memory cache of file contents. There is a single cache manager and single pool of cache pages for all of the different file systems and volumes on a particular Windows 2000 system. The cache manager operates by memory mapping files that are cached, and then using memory copy operations to/from the caller's memory in response to read/write calls. If the mapped page isn't present, the copy results in a page fault, which retrieves the appropriate data from the file system. The pages to which the cache manager has mappings are the same as the pages to which a user's memory mapped file would point, so memory mapped files and regular read/write based IO are coherent with one another. Misses in the cache result in page faults that are identical to those generated by user-level mapped accesses.

المنارة للاستشارات
www.manaraa.com

Because of the NT driver model, it is possible to develop filters with complex functionality independently of the other components with which they interact. The SIS filter was developed without any changes in NTFS, the NT I/O manager, virtual memory manager, cache manager or any other NT components.

2.2 Sparse Files and Reparse Points in NTFS

The Windows 2000 version of the Windows NT File System (NTFS) provides some new functionality that is used in the implementation of SIS: sparse files and reparse points. This section briefly describes these features.

A *sparse file* is a file that does not have physical disk space allocated for the entire file. Parts of the file that are not allocated are logically filled with zeroes. A file may be marked as sparse and extended without reserving disk space for the extension. An existing sparse file may have regions within the file deleted by a special IO control call, releasing the disk space and (logically) filling the deleted region with zeroes. A user can issue a different IO control that returns a description of the allocated and unallocated regions of a file. A write to an unallocated region causes disk space to be allocated. Unless a user specifically looks at a file to determine if it is sparse, it appears to be a normal file, possibly with much of the file being filled with zeroes. Users' disk quotas are charged for the sparse files as if they are fully allocated, regardless of how much disk space is actually used. Unallocated regions within files have a minimum granularity; the current implementation restricts them to aligned 64 Kbyte chunks.

A *reparse point* is a generalization of a symbolic link. A reparse point is placed on a file or directory by calling an IO control function. The reparse point consists of two parts: the reparse tag and the reparse data. The reparse tag is a 32 bit number that specifies the type of reparse point, and the reparse data is a variable size area that is not interpreted by the file system, but rather is used by a filter driver above NTFS that implements the functionality associated with the reparse point (or by the IO system in some special cases). SIS has a reserved 32 bit reparse tag.

When NTFS receives an open file request for a file with a reparse point, instead of doing a normal file open, it fails the request with `STATUS_REPARSE` and returns the reparse tag and data along with the completed (failed) IRP. Filters that use reparse points look for STATUS_REPARSE, and then check to see if the reparse tag is implemented by the filter. If not, the filter passes the completion up the driver stack. If the filter owns the tag, it can take whatever action is appropriate, based on the reparse data. If no driver claims the IRP and so the `STATUS_REPARSE` is passed all the way to the top of the driver stack, an error is returned to the caller.

There is an option flag bit for opens, `FILE_OPEN_REPARSE_POINT`, that specifies that reparse behavior should be suppressed. Unlike Unix symbolic links, a file with a reparse point on it is still an otherwise normal file. Specifying the flag tells NTFS that the file under the reparse point should be opened rather than returning a `STATUS_REPARSE` and letting the filters take action.

To illustrate the reparse point functionality, imagine implementing symbolic links using a filter driver and reparse points. The filter would have a reparse tag type allocated specifically for itself. The contents of the reparse data for a symbolic link would be the pathname component to be substituted for the file or directory in question. When the filter driver saw an open IRP complete with `STATUS_REPARSE` and the symbolic link tag, it would halt the completion process, modify the open request to have the pathname component from the reparse buffer replace the file name in the original open request, and send the request back to the file system for further processing. If an application wanted to create a symbolic link, it could simply place the appropriate reparse point on the file in question. To delete a symbolic link, it would open the link using `FILE_OPEN_REPARSE_POINT` (which would cause the open to complete without STATUS_REPARSE and thus prevent the filter driver from redirecting the open to the link target rather than the link) and then delete it in the normal way*.

2.3 The SIS Filter

The kernel portion of SIS is a filter driver that sits above NTFS. It handles all normal file operations that happen on SIS links, such as read, write, open, close and delete. It also implements a pair of special IO controls for creating new SIS links: `SIS_COPYFILE`, and `SIS_MERGE_FILES`. `COPYFILE` makes a SIS copy of a file, possibly turning the source file into a SIS link in the process. `MERGE_FILES` is used by the groveler to tell the filter to merge two files together. SIS_COPYFILE is unprivileged and is available to any user who has read permission to the source file and write permission to the destination. SIS_MERGE_FILES is privileged and only available to the groveler.

* Although the Win32 DeleteFile API [Microsoft 00a] takes a file name as its parameter, at the NT system interface level files are deleted by opening them and then sending down a delete call on the opened file handle.

www.manaraa.com

The remainder of this section describes the details of SIS link files, implementing reads and writes, handling memory mapped accesses to SIS files, copy-on-close, volume check and backing up SIS links.

2.3.1 SIS Links

SIS links usually do not contain any file data, but rather contain a reference to another file called a *common store file*. Common store files contain the data for files managed by SIS, and are located in a protected directory. By having the data for SIS files located in the common store rather than in any particular link file, SIS avoids the problems that would arise when such a "primary" SIS file was deleted or overwritten.

A SIS link is implemented as a sparse file of the size of the file it represents with (usually) no regions allocated. Because there are no regions allocated, the file uses only as much space as is needed for its directory entry. A SIS link has a reparse point with a SIS tag. The contents of the data portion of a SIS reparse point are the name of a common store file that backs the contents of the link, a unique identifier for the link, a signature of the contents of the common store file backing the link, and some internal bookkeeping information. The purpose for the signature is described below.

Creation of a SIS link is fairly straightforward. A user issues a `COPYFILE` request. If the source file is not already a SIS link, its contents are copied to a newly allocated file in the common store, and the source file is converted into a link to that common store file. The destination file is then created as a link to the (either pre-existing or newly created) common store file. SIS keeps some out of band information (called *backpointers*) associated with the common store file that contains the set of links that point to the common store file. A COPYFILE request adds such a backpointer for the destination, and also for the source if it was not already a SIS link.

The reason that SIS copies the contents of a non-SIS file into the common store rather than renaming the file is that it is possible to open NTFS files by file ID, which is a number associated with the file, somewhat akin to a Unix i-number. When a file is renamed, its file ID stays the same. Therefore, if SIS renamed the source file into the common store, users of the file ID would attempt to open the common store file rather than the link file. By doing a real copy, the SIS filter avoids the problem, although this means that a SIS copy of a non-SIS file takes effort proportional to the size of the file. An extension to NTFS that allowed moving the contents of a

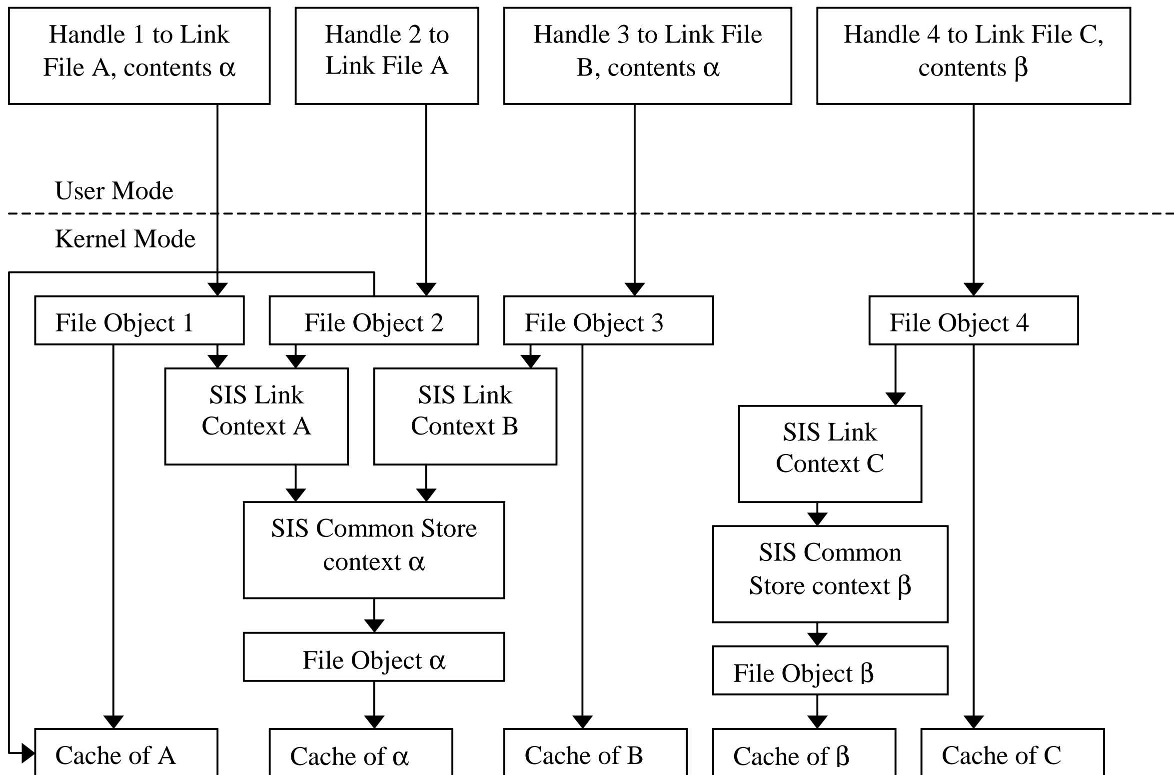

Figure 2: Example layout of SIS Contexts and Caching

المنارة للاستشارات
www.manaraa.com

file without moving its file ID would solve this problem, but it is not available at this time.

2.3.2 Normal Operations on SIS Links

When a user opens a SIS link, the filter intercepts the `STATUS_REPARSE` completion and resubmits the open request with the `FILE_OPEN_REPARSE_POINT` flag, resulting in the user's handle pointing to the link file (rather than the common store file). SIS also opens the common store file (if it has not already done so) and attaches some context to the user's handle indicating that this is a handle to a SIS link, and which common store file backs the link.

SIS must properly maintain coherency of SIS link files, meaning that users of a particular link must see all updates to that link, while seeing none of the updates to other links that are backed by the common store file. It is also desirable for SIS to conserve file cache space when possible by having different links to the same common store file backed by the same cache memory. Associating the cache with the common store file would violate coherence, while associating it with the link file would not conserve file cache space. Instead, SIS uses a hybrid of the two, where the caches associated with the links contain dirty (or potentially dirty) data and the caches associated with the common store files contain data that is known to be in common and clean.

Figure 2 shows an example of the in-memory structures created when a SIS link is opened. File objects are in-memory kernel objects that correspond to file handles, and are created by the kernel when handles are opened. Associated with a file object is a cache (maintained by the kernel cache manager), some file system state that is used to find the file's blocks on the disk (not shown), and (in the case of SIS links) a special context attached by SIS. The SIS link context contains a map of the file showing each region as clean, dirty or faulted. It also contains a pointer to a SIS common store context and some other various housekeeping information. The common store context contains a pointer to a file object that corresponds to the particular common store file, which in turn has its own cache. There is one handle and file object per successful CreateFile [Microsoft 00a] call, one SIS link context per open SIS link file (regardless of how many handles refer to it), and one common store context, file object and cache per common store file that has one or more open link(s) pointing to it.

As Figure 2 illustrates, there are two separate data caches associated with each opened SIS link: one for the link itself and one for the common store file that backs the link. In Figure 2, examples of these two caches for SIS link A are the boxes labeled "Cache of A" and "Cache of α" respectively. As long as there are no writes and no mapped file accesses, the cache associated with the link file is empty, and all of the cached file data is in the cache associated with the common store file. This is important in situations where more than one link to a particular common store file are is use at one time, because it reduces the cache requirements by up to a factor of the number of links to the common store file. This reduction in cache usage and the resultant ability to cache more file data in the saved memory is one of the main benefits of SIS to a file server.

Reads and writes initiated by ReadFile and WriteFile system calls (as opposed to by mapped file accesses, including those by misses by the system cache manager) are simple for SIS to handle. All writes are sent to the cache associated with the link file, and result in the written region of the file being marked dirty in the map in the SIS context associated with the link. When a ReadFile call tries to read a portion of a SIS file, the filter checks the map and sends the read to the common store file if the region is clean, and to the link file if the region is dirty or faulted. If the read spans multiple regions that are in different states, SIS splits the read into pieces and handles the pieces as appropriate.

2.3.3 Memory Mapped Access to SIS Links

Mapped file accesses present more of a challenge. When a user (or the system cache manager) maps a file and touches a page for the first time, it will generate a page fault which will be translated by the NT IO system into a read request that is sent to SIS. Read requests generated by page faults are specially marked, and the SIS filter can differentiate them from normal reads. Once SIS provides the data, the virtual memory system will map the appropriate page, and future accesses (either reads or writes) will not generate any action that will be seen by SIS. Other users who map and touch the same portion of the same file will be provided with a mapping to the same page, but the virtual memory system will not send a read to SIS, since it already has the data. The virtual memory system can unmap and throw away any clean (unwritten) pages without notifying SIS, and it will asynchronously generate writes for dirty pages at its own pace.

Because SIS cannot take any action when a mapped page is first written (because it doesn't get any notification of the write), in order to maintain coherence mapped pages must be associated with the link file cache and not the common store; if not, then two users could map two different SIS links that share a common store file, and they would see each others changes, violating the basic SIS semantics. We were unwilling to change the NT

المنارة للاستشارات
www.manaraa.com

memory manager to generate write faults when a page is first written.

Neither SIS's clean nor dirty states provides correct behavior in the case of a page that has taken a mapped read fault. Treating pages that have seen page faults as clean would result in normal (ReadFile) reads to those pages going to the common store file, which violates the coherence of normal reads with mapped writes. Treating them as dirty causes another problem: if there has been no write to the page the virtual memory system will see the page as clean, and may discard it. Sending a read to such a page would result in the user seeing a zero-filled portion of the link file, which is clearly wrong. To handle this problem, SIS has a third page state, *faulted*. When a page is in the faulted state, it sends normal reads to the cache associated with the link file, and page-fault reads to the common store file. Since a page-fault read will only happen if the page has been discarded by the virtual memory manager, when one occurs SIS concludes that the page has not been written and so it is safe to use the data from the common store file. SIS directs all accesses (reads and writes, page-fault and normal) to dirty file regions to the link file.

2.3.4 Copy-on-close

After all users close a SIS file that has had writes to it, the filter fills in the remaining clean regions (if any) with the data from the common store file. The actual copying happens in a system worker thread so as to not delay the close operation. If disk space is exhausted during this process, the filter simply leaves the remainder of the file unfilled, and leaves the reparse point intact. A subsequent open of this file results in the allocated regions of the file being marked dirty. We could have chosen to forego filling the clean regions of a file, and instead left the link backed in part by the common store file. If we had done so, we could have wound up in the odd situation of having SIS potentially increase the amount of disk space used in the system, by having files that are almost completely dirty backed by large and mostly unused common store files. To avoid this situation, SIS does the copy-on-close when possible.

Using copy-on-close rather than the more common copy-on-write has several advantages. The main one is that in many cases the file will be wholly overwritten, and so the copy can be avoided entirely. In his recent study of Windows NT 4.0 file system behavior, Vogels [Vogels 99] found that 79% of accesses to files were read only. Of the remaining accesses, 18% were write only and 3% were read/write. Of the write only file accesses, 78% of them were whole file writes. This means that of all files write accesses at least 67% were whole-file overwrites (conservatively assuming that none of the read/write files were wholly overwritten), and so copy-on-close would do no data copying. A second advantage is that the straightforward implementation of copy-on-write would stop all accesses to the file after the first write until the copy is completed, which could be a long time for large files. In an earlier paper [Douceur 99], we report that the distribution of files by size in Windows file systems is heavy-tailed, and so there are significant numbers of very large files; both we and Vogels found that about 20% of bytes are in files 4MB and larger, so it is reasonable to believe that writes to large files would not be unusual, and copy-on-write delays unacceptable. A final advantage of copy-on-close is that it allows sharing of file cache space for file regions that are unwritten.

2.3.5 Implementation Details and Backup

When a SIS link is eliminated, either by deletion of the link or because of an overwrite, the filter removes the corresponding backpointer in the common store file. When all of the backpointers for a common store file are removed, the filter deletes the common store file.

In certain circumstances, it is impossible for the SIS filter to prevent a user from writing an arbitrary reparse point. In theory, the user could write a SIS reparse point, which the filter would then use to read data for the user from the common store. This would be a violation of security, since the user did not have to prove access permission for the common store file before writing the reparse point. In order to address this problem, SIS includes a signature of the common store file contents in the reparse data. The signature is a hash of the entire contents of the common store file; it is easy to compute given the data in the file, but impossible to compute without the contents of the file (and 64 bits in length so randomly guessing is difficult). Each link file contains a copy of this signature in its reparse point. The filter will refuse to open a link file that contains an invalid signature. A reparse point that contains an appropriate signature proves that the user already knows the contents of the common store file (or has seen another reparse point that refers to the same contents, and so could have known the contents of the file). Because all that a user gets by creating SIS link is the ability to read the common store file contents, there is no security breach.

The SIS filter includes a facility called *volume check* for repairing inconsistencies in metadata, similar to the Unix `fsck` and Windows `chkdsk` programs. Unlike these programs, however, SIS is able to do its repairs while the system is running, without having to disrupt service in any major way. One limitation during a volume check is that if a user deletes what appears to be the last reference to a common store file the filter will not delete the common store file until the volume check completes,

المنارة للاستشارات
www.manaraa.com

because it does not trust its backpointers during a volume check. The volume check process will repair the backpointers for all common store files, and will delete any common store files that have no references. NTFS provides a method for efficiently finding all SIS links on a volume, so the time to complete a volume check is proportional to the number of SIS links on the volume, not to the size or total number of files on the volume. A volume check is initiated any time that the SIS filter notices an inconsistency.

SIS provides a special interface to backup/restore applications that allows them to behave appropriately with SIS links [Bolosky 99]. The goal is to have exactly one copy of the SIS file content on the backup tape for each backed up set of SIS links that refer to a given common store file. SIS provides a dynamically loaded library (DLL) for the backup/restore application. The backup application calls the DLL when it encounters a SIS link, and the DLL tells backup if it needs to back up a common store file in response. On restore of a SIS link, restore calls the DLL, which in turn looks to see if the appropriate common store file already exists or if it's already reported that file to restore. If not, then it reports the common store file corresponding to the link being restored. Because common store files have universally unique file names, and their content never changes once the file is created, if the common store file still exists on the volume there is no need to restore over it; simply linking to it suffices.

2.4 The Groveler

The groveler is a user level process that finds duplicate files in the file system, and reports these files to the SIS filter for merging. The essence of its task is efficiently to find the sets of matching files on a volume, and to keep the sets up to date as the volume changes. It maintains a database of signatures of files on the volume, and uses NTFS 5.0's update journal feature* to track files that have changed and to update their database entries.

The groveler database contains two structures: a work queue, and a mapping of signatures to files. The work queue contains work items of two types. The first is to compute the signature of a file, and the second is to compare a file to any others with matching signatures. The groveler has a thread that runs periodically, reads entries from NTFS's update journal and if appropriate makes entries on the groveler's work queue. If the groveler misses entries in the update journal, it detects this fact and re-scans the entire file system. A second thread drains items from the work queue, either computing a signature or comparing a pair of files for each item. It then updates the database, possibly instructs the filter to merge two files, and removes the item from the queue.

The groveler uses a 128 bit file signature. The first 64 bits of the signature are the size of the file. It is inexpensive to obtain the file size, and files with differing size obviously cannot be identical. The remaining 64 bits are computed by running a hash function on a fixed portion of the file's contents. The groveler hashes two 4 kilobyte chunks of file contents from the middle of the file (unless the file is less than or equal to 8 kilobytes in size, in which case it hashes the entire file). Hashing only part of the file means that any differences in the portion of the file not hashed will not be detected by comparing the signatures. However, restricting the hash limits the amount of work that the groveler does for large files. Even if the groveler hashed all of the file, the possibility of hash collisions would still exist, and in order to guarantee that it does not corrupt data it would be required to compare files byte-by-byte before linking them together. Therefore, unless hashing a smaller portion of a file greatly increases the probability of hash collisions for unequal files, the optimization is worthwhile. We have noted very few cases wherein files' hashes match and the files' contents do not, even though we only hash a constant amount of data per file.

The groveler is set up to run as a low importance background task (unless disk space is tight). The groveler is not particularly CPU intensive; most of the work it does is disk I/O. As a result, simply running it at low scheduling priority does not prevent it from interfering with other, more important tasks. The groveler uses a scheme where it tracks its own rate of progress, and slows down its run rate when its progress slows. The premise is that if the groveler is running more slowly, it must be contending with some other, higher priority process for some resource. The toolkit that implements this technique is called "MS Manners" and is described in detail in [Douceur 99a].

In order to do its work the groveler needs to open files, and in some cases to hold them open for a long time. This could cause problems for other users of the files who might want to have exclusive access to a file for some reason, and would have their opens fail with a sharing violation. In order to mitigate this problem, the groveler takes a batch oplock on each file it opens immediately after opening it. Oplocks [Microsoft 00a]

* The update journal [Microsoft 00a] is a feature whereby NTFS maintains a record of all recent updates to a volume in a ring buffer. Each entry in the journal has a sequence number (USN, "update sequence number"), so it is possible for a user of the journal to determine if it has missed any entries because of a gap in the sequence numbers.

المنارة للاستشارات
www.manaraa.com

are a facility by which the opener of a file can be notified when another user wants access to the file, and can take action before the other user's action is completed or failed. They were designed to allow effective local caching of files that are shared over a network while maintaining coherence. However, in the case of SIS they allow the groveler to close a file that it is using without generating a sharing violation for another user of the file.

3. Performance of and Uses for SIS

In Windows 2000, SIS is deployed only on remote install servers, and then only on the volumes that contain the remote install system images. While the initial release of SIS is used only for remote install servers, we believe that in the future it may be put to other uses. We briefly present measurements that show that the per-link space overhead of a SIS link and the time to make a copy of a SIS link. We measured a remote install server at Microsoft, and report on the disk space (though not file cache) savings that SIS provided for that server. We also discuss using SIS on file servers that back the files of multiple users, and using SIS in a serverless, distributed file system that we have proposed elsewhere [Bolosky 00].

We measured the time to make a SIS copy of a file that is already a SIS link using the SIS_COPYFILE API on a Gateway 2000 E-5000 Pentium II 300MHz personal computer running Windows 2000, with 512 MB of memory, and a 9 GB ST39173N 7200 RPM Seagate Barracuda disk drive containing an aged file system and a 9 GB ST19171W Seagate Barracuda 7200 RPM disk drive containing a clean file system. We ran all tests with the network disconnected. The clean file system was almost empty and newly formatted while the aged file system was 70% full and had been in use for several years, although it had been defragmented a few months before we ran the test.

We determined the time to make a SIS link by creating a file, making one SIS copy of it to make the source file a SIS link, making 10,000 SIS copies of the file, noting the wall clock time for the execution and dividing by 10,000. We ran the 10,000 copy measurement 100 times on each of the two file systems. In each instance, we discarded the first 10,000 copy run so that we would be running with the file caches hot.

In our tests (using a file about 1.6MB in size), a SIS copy took 4.3ms (+/- 290µs at 99% confidence) on the clean file system and 8.6ms (+/- 220µs at 99% confidence) on the used file system to copy the file. It is difficult to say how much of the difference is due to the state of the file system and how much is due to the fact that the dirty file system is running on an older model of disk. The reason that the copy is this slow is that there is a synchronous disk write in the SIS backpointer update, which is necessitated by the inability of several NTFS metadata updates to be grouped into a single atomic transaction. Copying the same 1.6MB file using a normal file copy took about 260ms per copy on the clean file system.

We also measured the space cost to create a SIS link by noting the amount of free disk space, making a large number (10,000) of SIS links, noting the amount of free space again, and dividing by the number of links. The overhead was about 300 bytes/link for all power-of-ten file sizes from 10^0 to 10^8 bytes.

We measured a remote install server at Microsoft that is used to install various versions of Windows NT for testing purposes (of the systems being installed, not the server). This server contains 20 different images of Windows NT of various flavors: both Windows 2000 Professional and Server, and different internal builds of the system, including the last five builds before the final Windows 2000 product, the release version of Windows 2000, and a build from after the Windows 2000 release. The remote install volume on this server contained about 112,000 files, and a logical 7.5 GB of file contents. Of this, 45,000 files (39%) and 6.0GB (80%) were in SIS links, backed by 1.6GB in 13,000 common store files. The overall space savings from SIS was 58%. In a field deployment, one would expect to see fewer different versions of the operating system with more different sets of applications installed, which we expect would result in better space savings.

We would have liked to measure the advantage in buffer cache usage for a remote install server running SIS. Unfortunately, we did not have the ability to take traces of a real remote install server in action. While we could have used synthetic workload traces to simulate such a server, barring real measurements there is no good way to determine the parameters for the workload generator. Because cache performance is strongly (and non-linearly) influenced by the working set size, the exact workload parameters would almost wholly determine the results of such a simulation. Therefore, lack of real parameters makes the synthetic workload exercise at best meaningless, and more likely misleading.

SIS could be used for file servers that store the files of groups of users, such as is typically done with NFS [Pawlowski 94; Sandberg 85], AFS [Howard 88], Coda [Satya 90; Kistler 91] or any of a number of comparable systems. In fact, the original reason for building SIS was to support the remote boot server, a similar idea for Windows 2000, but remote boot was cut from the final product for schedule reasons.

المنارة للاستشارات
www.manaraa.com

In [Bolosky 00] we measured the contents of a number of desktop personal computer file systems at Microsoft and observed the duplication of contents among them. We found that the level of duplication depends on the number of file systems grouped together, with the percentage of reclaimable space growing roughly with the log of the number of file systems. Grouping 100 randomly selected file systems gave a little better than 30% space savings. At 1000 file systems the savings was just under 50%. Our model predicts a little better than 60% savings at 10,000 file systems. File systems of people with similar job functions (e.g. software developer, secretary, manager) were more alike than the randomly selected groups of file systems quoted above. We are unaware of any comparable study or raw data for any other environment, but we expect that there may be significant differences on other operating systems or even for institutions other than Microsoft.

The file system that we propose in [Bolosky 00] is intended to provide a shared name space and common access to storage for tens of thousands of users using only the desktop workstations of those users. One issue in such a design is providing availability of files on machines that are much less available than managed servers. Our approach is to replicate the file contents across the machines in the hope that the system can find at least one copy of a file on a machine that is up. The number of replicas that can be made strongly (exponentially) influences the probability that a file will be available. By using SIS (or a similar technique) the total size of the stored file content can be reduced, and hence the number of copies that will fit on the available disk space will be increased, greatly improving the overall availability of the system. That is, even if SIS only provides modest space savings, these savings can result in greatly improved system performance.

4. Related Work

There are a number of different uses of copy-on-write in computer systems. Mostly they share the same characteristics: A traditional copy would be expensive in time or space (or both); the semantics presented to the user are those of a copy, rather than a link; and, the expectation of the system designer is that the copy-on-write will rarely happen. Typically, the cost of the initial "copy" followed by the copy-on-write is higher than just eagerly evaluating the copy, but this is made up for by the common case in which the copy-on-write never happens [Fitzgerald 86].

Copy-on-write has been used in virtual memory systems as least as far back as Accent [Rashid 81] and Mach [Accetta 86;Young 87]. These systems allowed processes (including file systems) to send messages to one another with copy semantics, but used the virtual memory system to map the same memory into both processes' address spaces. If a process subsequently wrote into the memory, the system took a protection fault, made a copy of the page in question, and mapped the newly copied page into the faulting process's address space with write permission.

The Microsoft Exchange Server [Todd 96] (a multi-user mail server) uses copy-on-write techniques for mail messages that are sent to multiple recipients, and even calls the technique "Single Instance Store." This mail system allows recipients to modify mail messages after they're received, which triggers the copy-on-write. It does not have an equivalent of the groveler; if two identical messages are in a server, but they were not generated as copies of one another, the server will never merge them.

Apollo systems used access control lists (ACLs) that described the lists of users who had access to particular files. These ACLs were immutable. The `salacl` command looked through the sets of ACLs existing in the system, and combined those that matched [Leach 98], functionality that is similar to the groveler, but the because the ACLs were immutable there was no need for the copy-on-close function of the SIS filter.

Many file systems support various types of links with semantics differing from SIS links. In particular, Unix file systems [McKusick 84] typically support both hard and symbolic links. These types of links differ from SIS links in that writes through one link to an object are visible through a different link. NTFS [Custer 94] supports hard links that are similar to those in Unix. None of these systems automatically detect and merge files with identical contents, because such an action does not make sense when links have different semantics from separate files.

It is common practice to aggregate files of many users onto a central file server, which may be implemented as a single machine or a cluster. NFS [Sandberg 85], Sesame [Thompson 85], AFS [Howard 88], the Sprite file system [Ousterhout 88], Coda [Sayta 90], Ficus [Guy 90] , Swift [Cabrera 91], Zebra [Hartman 93], the Microsoft file systems using the Server/Redirector network remoting services [Solomon 98], and a host of others too numerous to mention all are variations on the theme of a centralized network file server that often will combine the files from many users on many client machines onto a single file system.

There are number of serverless (ie., decentralized) distributed file systems similar to the one that we mention at the end of Section 3. Chief among them are

المنارة للاستشارات
www.manaraa.com

Frangipani [Thekkath 97], and xFS [Anderson 95]. They differ from our proposed file system in a number of ways, among them that they assume that the machines that implement the system are secure, and are more highly available than is reasonable to expect from desktop workstations.

5. Summary

SIS is a component of Windows 2000 that detects files that have identical contents and merges them into SIS links, special links that present the semantics of separate files while in most cases using the disk and file cache space of a single file (plus a small disk overhead per link that does not depend on the size of the underlying file). In Windows 2000 SIS is used as part of the remote install server, which is a way of setting up machines to a pre-determined configuration without having to go through the normal set-up process.

SIS is implemented as a file system filter driver and a user-level service. The filter driver implements the links and copy-on-close when a file is modified, presenting the semantics of an independent copy to the user of the link. The user-level service watches changes to the file system, computes signatures for newly-changed files, compares files with matching signatures and reports matching files to the filter for merging.

The cost of making SIS copies of files that are already SIS links is small and independent of the size of the file. The disk-space overhead of a SIS link is about 300 bytes regardless of the size of the file to which the link refers. In most instances, copying a SIS file takes about 8.6ms on the (pretty slow) machine and thoroughly aged file system on which we took our measurements. We were unable to measure the impact of SIS on file cache usage in a real installation, but hypothesize that in some cases it may significantly improve performance by reducing the cache working set below the available memory size.

We speculate that SIS could be useful in contexts other than the remote install server, in particular a distributed, serverless file system built on ordinary workstations. Because that file system's performance is strongly influenced by the amount of free disk space available, using SIS to reduce the effective disk space usage provides large benefits in availability.

6. Availability

SIS ships with Microsoft Windows 2000 as part of the Remote Installation Services. The source code is available with a Windows 2000 source license, which is available from Microsoft on a case-by-case basis. A suitably sanitized version of the raw data used in [Douceur 99] and [Bolosky 00] is available by request from John Douceur, johndo@microsoft.com, on a set of CD ROMs. It is much too big to place on the net for download.

Acknowledgments

We would like to thank Cedric Krumbein, Mihai Popescu-Stăneşti, Drew McDaniel, Steven West, Galen Hunt and Yi-Min Wang for their help in implementing and testing the SIS components. Some of the early conceptual work on SIS was done in conjunction with Rick Rashid, Nathan Myhrvold and Terri Watson Rashid. Rob Short and Chuck Lenzmeier were invaluable in navigating the politics of the Windows NT group. Brian Andrew provided help in understanding NTFS and its behaviors, and provided useful suggestions as to how to implement SIS. Steven West and Matthew Stevens provided access to remote install servers so that they could be measured. Felipe Cabrera commented on the SIS/backup interactions.

References

[Accetta 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M.Young. "Mach: A New Kernel Foundation for UNIX Development," In *Proceedings of the Summer USENIX*, July, 1986.

[Anderson 95] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. "Serverless Network File Systems," In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 109—126, December 1995.

[Baker 97] A. Baker. *The Windows NT Device Driver Book*, Prentice Hall PTR, 1997.

[Bolosky 99] W. Bolosky. "The SIS/Backup Interface," available upon request from Steve Olsson, solsson@microsoft.com.

[Bolosky 00] W. Bolosky, J. Douceur, D. Ely and M. Theimer. "Evaluation of Desktop PCs as Candidates for a Serverless, Distributed File System," to appear in *Proceedings of ACM SIGMETRICS 2000*.

[Cabrera 91] L. Cabrera and D. D. E. Long. "Swift: Using Distributed Disk Striping to Provide High I/O Data Rates," *Computing Systems*, 4(4):405—436, Fall 1991.

[Custer 94] H. Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[Douceur 99] J. Douceur and W. Bolosky. "A Large-Scale Study of File System Contents," in *Proceedings of ACM SIGMETRICS '99*, pp. 59—70, May 1999.

[Douceur 99a] J. Douceur and W. Bolosky, "Progress-based regulation of low-importance processes," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 247—260, December, 1999.

www.manaraa.com

[Fisher 98] L. Fisher. *The Windows NT Installable File System Kit*. A Microsoft product that can be ordered from http://www.microsoft.com/hwdev/ntifskit.

[Fitzgerald 86] R. Fitzgerald and R. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems*, 4(2):147—177, May, 1986.

[Guy 90] R. G. Guy, J. S. Heidemann, W. Mak, T. W. P., Jr, G. J. Popek, and D. Rothmeier,. "Implementation of the Ficus Replicated File System," Proc. of the Summer 1990 USENIX Conference, pp. 63—71, June, 1990.

[Hartman 93] J. H. Hartman and J. K. Ousterhout. "The Zebra Striped Network File System," *In Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 29—43, December, 1993.

[Howard 88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, 6(1):51—81, February 1988.

[Kistler 91] J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System," In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 213—225, October, 1991.

[Leach 98] P. Leach. Personal communication. He said this is only documented in the manuals, not any other published sources, and these manuals are now hard to find.

[Li 86] K. Li and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems," In *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pp. 229—239, August 1986.

[McKusick 84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. "A Fast File System for Unix," *ACM Tranactions on Computer Systems*, 2(3):181—197, August, 1984.

[Microsoft 00] Microsoft Windows 2000 Server online help file. Microsoft Corporation, February 2000.

[Microsoft 00a] Microsoft Developer Network Library. A product available from Microsoft. See http://msdn.microsoft.com/. January, 2000.

[Nagar 97] R. Nagar. *Windows NT File System Internals*. O'Reilly, 1997.

[Ousterhout 88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch. "The Sprite Network Operating System," *IEEE Computer* 21(2):23—36, February, 1988.

[Patterson 88] D. A. Patterson, G. Gibson, and R. H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)," In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, pp. 109—116, June 1988.

[Pawlowski 94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel and D. Hitz. "NFS Version 3 Design and Implementation," In *Proceedings of the Summer USENIX Conference*, pp 137—152, June 1994.

[Sandberg 85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. "Design and Implementation of the Sun Network Filesystem," In *Proceedings of the Summer USENIX Conference*, pp. 119—130, June 1985.

[Rashid 81] R. Rashid and G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel," In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pp. 64—75, December, 1981.

[Satya 90] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. "Coda: A Highly Available Filesystem for a Distributed Workstation Environment," *IEEE Transactions on Computers*, 39(4), April 1990.

[Solomon 98] D. Solomon. *Inside Windows NT, Second Edition*. Microsoft Press, 1998.

[Thekkath 97] C. Thekkath, T. Mann and E. Lee. "Frangipani: A Scalable Distributed File System," In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 224—237, December, 1997.

[Thompson 85] M. R. Thompson, R. D. Sansom, M. B. Jones, and R. F. Rashid. "Sesame: The Spice File System," Carnegie-Mellon University Computer Science Technical Report CMU-CS-85-172, Carnegie-Mellon University, Pittsburgh, PA. 1985

[Todd 96] G. Todd, *et al. Microsoft Exchange Server Survival Guide*. Sams Publishing, 1996.

[Vogels 99] W. Vogels. "File system usage in Windows NT 4.0," In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 93—109, December, 1999.

[Young 87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 63—76, November, 1987.

المنارة للاستشارات
www.manaraa.com