Implementing File Systems and Object Databases in a Microstorage Architecture*

Dawson Dean
Richard Zippel

TR 93-1393
October 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Rsearch Contract N00014-88-K-1989, the Office of Naval research through contract N00014-92-J-1839 and in part by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University.

Implementing File Systems and Object Databases in a Microstorage Architecture*

Dawson Dean and Richard Zippel
Department of Computer Science
Cornell University, Ithaca, NY 14850
dawson@cs.cornell.edu
rz@cs.cornell.edu

October, 1993

Abstract

A *microstorage architecture* consists of a microstorage kernel and several storage servers. Each storage server implements a storage model that defines a client's view of all the data in the system, how it is stored, retrieved and manipulated. The storage servers are built on top of the microstorage kernel and rely on it to perform the actual data storage and retrieval. The microstorage kernel implements a mechanism and the storage servers each implement specific policies defined by their storage models. Several different storage servers, each implementing a different storage model, may run concurrently over the microstorage kernel and all data in the system is concurrently visible to all the different storage servers. Different application programs, or different parts of the same application program, can use different storage models, to manipulate the same data. Microstorage architectures provide a flexible interface and a smooth transition from traditional file systems to more powerful object oriented storage models. Existing applications continue to work correctly unchanged because they use a storage server that implements a traditional file model while new applications may gradually adopt more powerful storage models.

1 Introduction

Modern application programs support increasingly rich documents that contain many different types of data, such as text, pictures, sounds, and simulations. Different programs, written by different developers, may manipulate a single document, and no single program may understand all of the data in such a complex document. Different programs, or different parts of the same program, may organize the contents of a document differently to support different operations. For example, a database program views a document as a collection of records, and perform operations on individual records, while a word processor may view the entire document as a stream of characters.

At the same time, the boundaries between documents are diminishing. Documents may share common sections, and end users may jump between related sections of different documents. A single piece of data, such as a component description in a mechanical design, may appear in several

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-1989, the Office of Naval Research through contract N00014-92-J-1839 and in part by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University.

different documents, such as a parts database, an illustration of the design, and a marketing plan. Increasingly, data in the system is organized according to how people use it, not according to file boundaries in a traditional file system.

Modern operating systems should provide better support for modern application programs. They should provide more powerful storage architectures that provide several different views of the same data and not impose artificial limits on sharing data. Different programs, or different parts of the same program, should be able to view the same data in a variety of ways, such as files, objects, or database records. Data should be seamlessly shared between documents and programs. Operating systems should also provide backward compatibility, so existing programs may continue to view data as organized in traditional files, while new programs may gradually adopt more sophisticated models.

Microstorage architectures provide this functionality. Microstorage architectures consist of a microstorage kernel and several storage servers. Each storage server provides a different view of all of the data in the system. This paper introduces microstorage architectures, and describes the Vista microstorage system as a specific example. Section 2 describes the high level design goals of the system and section 3 describes how these goals might be achieved with different storage architectures. Microstorage architectures provide a better implementation of these design goals and section 4 introduces microstorage architectures. Section 5 explains how microstorage architectures maintain consistency between different storage models. The next sections explore specific parts of a microstorage architecture: section 6 discusses the microstorage kernel, section 7 discusses a file storage server, and section 8 examines an object storage server. Section 9 reexamines the original design goals, and discusses how they are implemented in a microstorage architecture. Section 10 discusses related work, and section 11 discusses open questions and future work.

2 Design Goals

Microstorage architecture are designed to support a new generation of applications that provide more powerful end user features. Microstorage architecture do not directly implement these end user features, rather they provide technology that allows applications to easily implement them. To understand microstorage architectures, it is important to first understand these end user features. Although most of these can be implemented with a file system, they are not simply extensions to files. They also apply to other storage models, and some fit more closely with other, more powerful, models.

2.1 Perform Queries On Any Document

End users should be able to use queries to find information in any document. Currently, database applications allow end users to query database files, but the same functionality should be available for all documents and applications. For example, an end user might use a query to find certain components in a CAD document, or certain cells in a spreadsheet, or all blue images in a graphics document. End users should be able to use a single query language for all documents and all applications although the specific query interface should not be important. Different end users may use different query languages and interfaces with the same documents.

End users should be able to use the output of a query to create a new file or extend an existing file. For example, an end user may query an existing document to find specific bibliography references and then create a new bibliography file, such as a BIBTEX file, that contains these references. The new file should behave like any other file, so programs like BIBTEX can use it without any change to the program or the file. The user may request that the file be continuously updated, so when new interesting entries are added to the bibliography database they will also be automatically included in the file.

End users should be able to use queries to locate entire files. For example, an end user should be able to name a file with the query, "the letter I sent to Bob last week that contained a picture of the engine design", rather than be forced to use a file name such as:

/usr/me/letters/old/bob/engdes2.txt

Applications should accept a query anywhere they currently accept a file name.

2.2 Navigational Links

Users should be able to follow links between related sections of documents. For example, an end user reading a technical paper that contains a reference to a related paper should be able to open the related paper with a simple gesture, such as clicking on a bibliography entry with the mouse. The referenced document will be immediately opened and displayed by the appropriate application, so if the referenced document was created by a CAD program, the CAD program runs and displays the referenced document. When the referenced document is opened, a specific section may be highlighted. For example, if an end user follows a link to an entry in an online encyclopedia, the encyclopedia document opens and scrolls to display the specific entry. Links between documents should be persistent, so they remain valid even when the referenced document is edited, moved, or renamed.

Several hypertext applications currently offer this kind of functionality, but it is typically limited to documents created by that single program. End users should be able to create links in any program, and link files created by different programs.

2.3 Rich Documents

Documents should be rich, consisting of words, pictures, sounds, movies, simulations and much more. End users do not want to create separate files for each part of a document, nor do they want to jump between several different programs to edit a rich document. No single program, however, can support all possible kinds of data in a way that pleases all users. A word processor may be good at editing and formatting text, but it may lack the sophisticated drawing features to correctly render and edit a CAD illustration. As a result, programs may not know the format or semantics of all parts of the rich documents they display and edit. Instead, programs will have to cooperate with other programs that understand the various parts of a document. For example, a word processor may ask a CAD program to edit a design drawing that appears in a text document. This type of inter-application communication requires both a powerful storage model and a standard for inter-application communication.

Separate documents should also be able to share a common section. For example, a mechanical design document and an inventory database may share a section that describes the parts list for a product. If the shared section is changed in one document, then it should be automatically changed in all documents. Updates to shared sections should be propagated whenever a file that contains the shared section is used, whether that file is opened by a user or by another program such as an agent. This means updates must be propagated by the storage system, and not rely on user intervention. Moreover, if several files that share a section are currently open, their contents should be updated immediately. Some applications may go further, and propagate updates to shared sections even when they have been copied from a file into virtual memory. For example, a word processor might copy part of a rich document into virtual memory when it is displayed. If a shared section in that document is changed by another program, then the changes may update both the copy of the section on disk and the copy in memory.

2.4 Context Sensitive File Contents

A file should be able to adjust its contents to fit different situations. For example, a file may be opened by various users with various levels of access. Typical file systems grant read and write access to users and groups for the entire file. If a user can read or write part of a file then he can read or write the entire file. Files should instead provide more selective access controls that allow a user to read only portions of the file, or read only a summary of the file contents or read some parts and write to different parts. For example, a research paper may display review comments from colleagues when the owner opens the file, but not when other people open it. Granting access to part of a file is similar to granting access to a view of a database. The access controls must be implemented in the storage system, not applications, or else a single untrusted application might fail to enforce them.

Documents may also hide sections that an end user is not currently interested in. For example, documents may include previous versions, and end users may specify that they want to read a specific version when they open the file. Documents may also hide or reveal different sections depending on the application program that opens them. For example, a source code file may have sections written in different programming languages, or sections built for different operating systems, or include sections like a drawing of the data structures and a bug database.

An extreme example of context sensitive files is to completely create new file contents every time it is opened. For example, a file may contain the latest values of a variety of sensors, such as the load average of several machines or network traffic statistics. Each time the file is opened, a variety of system and network management programs create the entire file contents by collecting the latest statistics. Files can also be updated when some event other than opening the file occurs. For example, when a source code file is changed, the corresponding object file may be automatically recompiled.

2.5 Compatibility, Extensibility and Smooth Transitions

As the software industry has grown, it has developed a massive installed base of programs that use traditional file systems. Any storage architecture should be fully compatible with these existing programs; existing programs should continue to work correctly without change and existing files should continue to be fully available without change. Programs that use the new features of a storage architecture should be able to seamlessly share data with existing programs that use traditional files.

Most programmers are also familiar with traditional file interfaces, so there should be a smooth transition when adopting the new features of a storage architecture. New programs may continue to use existing file interfaces, or adopt some new storage features while continuing to use files, or adopt many new features and abandon the file model. All types of programs should be able to share data seamlessly.

2.6 Distribution and Heterogeneous Systems

A storage architecture should be able to work in a large distributed environment. This means resources may be stored on different machines, and different machines may have different standards for data representation and alignment. Some operations, such as distributed garbage collection, may be costly due to communication delays. The system has to handle more complicated error cases, such as some machines crashing and rebooting at various times during a long sequence of operations. Although these are not specific end user features, they do impact the overall utility of the system and influence the design.

3 Storage Architectures

3.1 Traditional File Systems

Application programs typically store data in files[1]. When programs implement some of the above features, such as linking and embedding sections [4, 57], they currently do so with traditional file systems. Unfortunately, files are not the most natural container for the functions described above. Those functions typically manipulate individual sections of a document: a rich document consists of different kinds of sections, and operations such as linking and embedding relate sections between documents. Files, however, store data as a linear sequence of bytes so application programs first have to parse files into sections and then associate different behavior and semantics with each section.

A typical file system does not provide any mechanism for persistent and globally unique section identifiers. For example, a link between sections in different files must uniquely identify the section in the referenced file, even after both files have been saved to disk and opened again much later. The task of assigning and storing section identifiers is left to each application program, and applications that share sections must agree on a standard format for these identifiers. If a section is shared between files, then there must be some controls on the sharing that protects individual sections, such as access controls and locking. If users are able to set access controls on sections, then there must be user-visible names to identify sections and applications will have to map these names to section identifiers.

Normal file systems do not propagate updates to shared sections, so applications must assume this responsibility and inform each other of any changes to shared sections. Programs must communicate and cooperate with each other when they display and edit rich documents that contain different kinds of sections. For example, a word processor may have to rely on other programs to draw or edit some sections in a document it created. The word processor must locate these other programs, send messages they understand, and gracefully pass control to the other programs while they draw or edit the sections.

A file system does not help programs manipulate sections after they have been copied from a file into virtual memory. The file system does not convert links to sections on disk into pointers to sections in memory. The file system does not propagate updates or enforce consistency rules for sections in virtual memory.

Finally, file systems do not support data translation between different architectures. For example, Sun and MiPS processors have different representations of integers. This does not effect a text file whose contents is simply ASCII bytes, but it does effect many other files, such as any file that contains floating point numbers, 16 or 32 bit integers, or pointers. Without support from the file system, it is up to the application to convert data to the representation appropriate to the local processor. For example, if several programs running on different workstations share a section, they must all perform data translations when they read the section.

Because of the limitations of file systems, application programs must implement many of the features of an object store on top of a file system. Instead of using the file model, these programs hide files under their own ad hoc object store and use the file system as little more than a high level disk interface. This makes applications extremely difficult to build, hurts performance by adding more layers of software, and makes standards for sharing sections between different applications difficult to define.

Instead of parsing files into sections, applications instead can implement each document section with a separate file and implement documents with directories of files. This is done in PenPoint [19] and solves the problems of parsing documents into sections and implementing sec-

[1] Database applications are an important exception and often implement their own private storage architecture, bypassing the operating system's file system and directly managing disk storage. Most programs, however, like word processors, mailers, CAD and multimedia packages use the operating system's native file system.

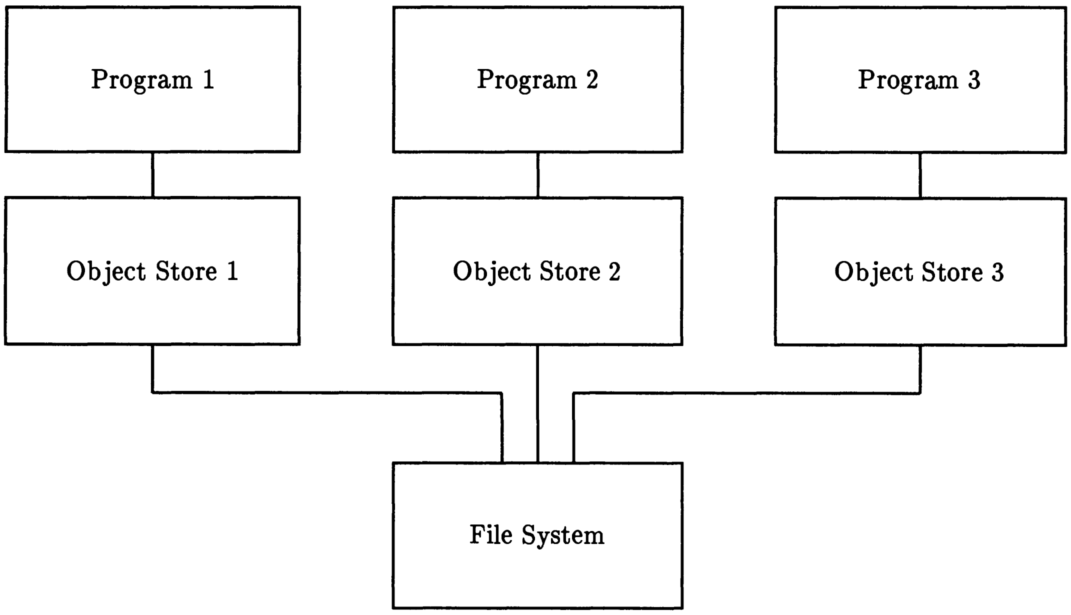

Figure 1: Object Storage Built On A File System

tion identifiers. Unfortunately, the resulting rich documents are not compatible with existing programs; at best the end user runs existing programs like word processors on individual sections of a document. The system also does not support data translation or support sections in memory. Such a system really treats the underlying files as objects or disk segments, and thus resembles an object oriented file system. This is described in the following section.

3.2 Object Oriented File Systems

An alternative design is to implement the file system on top of objects. Each file is implemented as a collection of objects, and each object stores a different section of a document. For example, a text file might be implemented as a collection of paragraph objects. Application programs continue to manipulate files with a traditional file interface, but they may also manipulate the individual objects that make up a file.

This architecture implements the specific end user features described above, but it may be unsatisfactory for other unexpected storage features. Different applications may use data models very different than objects and files. For example, a database application views all data in the system as records in a table, while an application from the Apple Newton views all data as a data soup and applications built on segment oriented systems like Multics use a segment model. Object oriented file systems are biased toward the file model. They specify that data is principally accessed through files and objects are simply a mechanism for manipulating the contents of files.

4 Microstorage Architectures

Microstorage architectures recognize that files and objects are simply two of many different ways of viewing data. Each view is appropriate for some operations, and applications may use different views to perform different operations on the same data. A microstorage architecture consists of a *microstorage kernel* and several *storage servers.* Each storage server implements a storage model

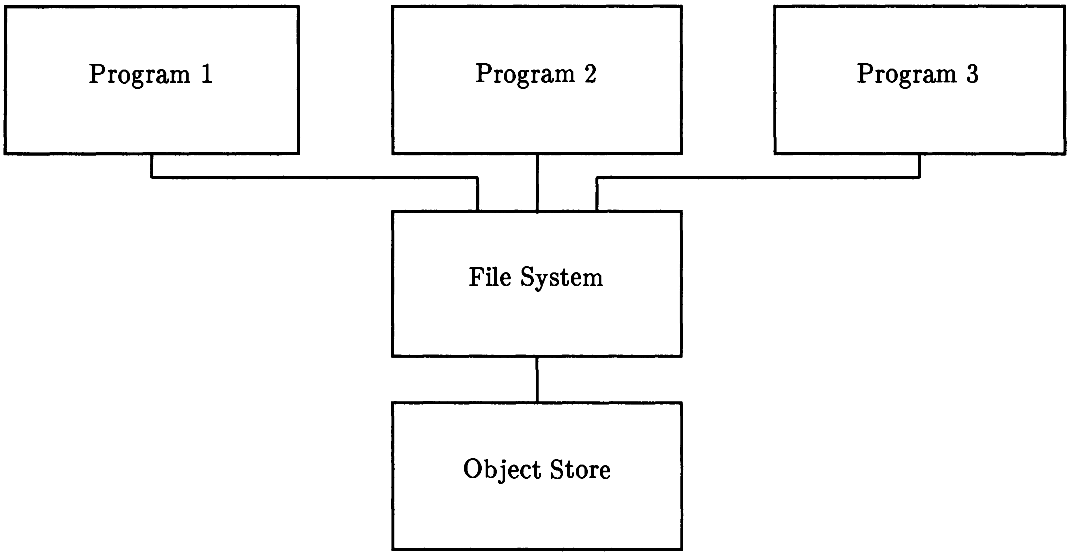

Figure 2: Object Oriented File System

that defines a client's view of the data in the system, how it is stored, retrieved and manipulated. For example, one storage model might define a traditional file system such as the Unix file system. This model specifies that data is stored in files that are sequences of bytes. This model also defines the routines that manipulate files, such as open, close, read and write. Another data model might specify that data is stored as objects in an object oriented database. This model defines how objects are typed and bound to methods that enforce encapsulation. Other data models might specify that data is stored in a semantic net, or define a relational database model, or declare that data is stored as persistent program variables. Each of these storage models is an abstract definition, and the corresponding storage server is the actual implementation.

Several different storage servers, each implementing a different storage model, may run concurrently over the microstorage kernel. All data in the system is concurrently visible to all the different storage servers, so the same data may be accessed and manipulated through any combination of storage models, such as the file and object models. Different application programs, or different parts of the same program, may use different storage models to access any data in the system. For example, a program may use a file system storage server to open and read a file, and then use an object database storage server to read and write individual objects contained within the file. In this example, the same data is viewed as both a stream of characters in a file, and as a group of paragraph objects. Application programs use the most appropriate model for each operation, even if they share data with applications that use different models.

The storage servers are built on top of the microstorage kernel and rely on it to perform the actual data storage and retrieval. The microstorage kernel implements the mechanism while the storage servers each implement specific policies defined by their storage models. The role of a microstorage architecture in data storage is analogous to the role of a microkernel [62] in an operating system and microstorage architectures are designed to fit easily in a microkernel operating system. New storage servers, implementing new storage models and specialized for different tasks, may be added at any time, so the system can be easily extended and specialized.

Because it concurrently supports several storage models, a microstorage architecture also

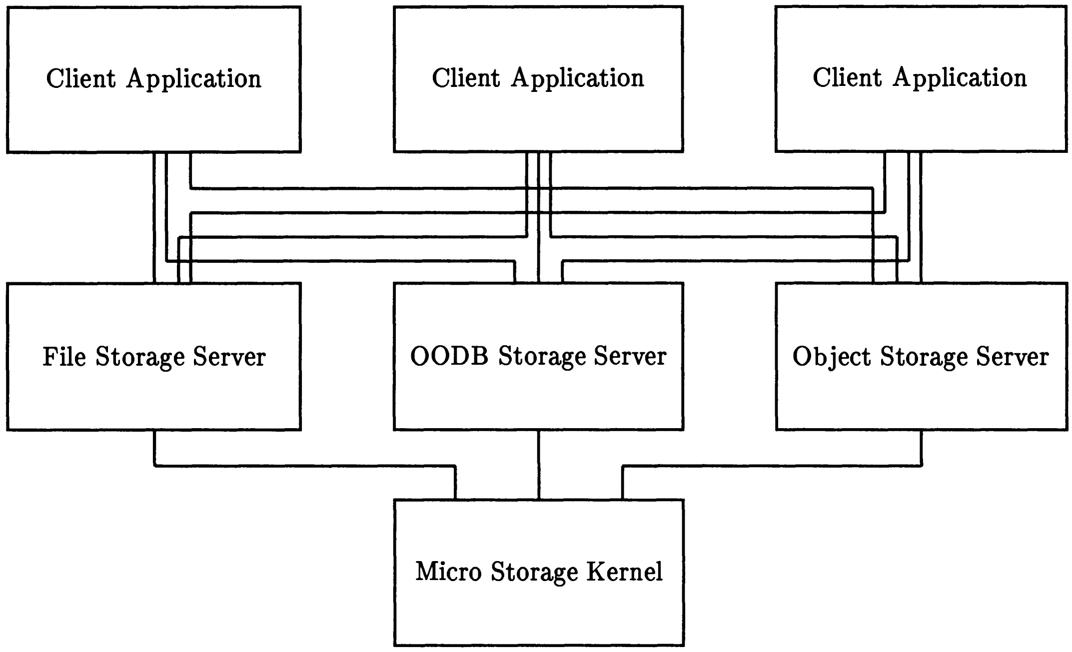

Figure 3: Microstorage Architecture

provides backward compatibility and a smooth transition from traditional file systems to more powerful object oriented storage models. Existing applications continue to work correctly without any modification because they use a storage server that implements a traditional file model, such as the Unix file system. New applications may incrementally use new features from new, more powerful, storage models while still using a traditional file system. Finally, new applications may completely abandon a traditional file storage model and exclusively use new storage models. All of these types of programs may continue to share data seamlessly.

A microstorage architecture, however, does not by itself implement all of the functionality needed for all of the user features described above. For example, it does not implement inter-process communication or a standard set of messages used by several cooperating applications to display a compound document in a window system. These functions are necessary, but they fall outside the domain of a storage architectures since they describe how applications are structured and how they communicate. These functions are implemented instead by an application architecture. A microstorage architecture, however, is an important part of implementing the user features described above.

5 Consistency Between Models

The storage servers in a microstorage architecture must guarantee that data remains valid in all data models. For example, if data is read and written as a group of objects in the object model and as a single file in the file model then it must continue to be both a valid file and a valid group of objects. The object storage server must not perform operations on individual objects that would make the data an invalid file and the file storage server must not perform operations that would make the data invalid objects. The microstorage kernel and individual storage servers work together to enforce consistency.

Each storage server has a unique 4 byte signature that is assigned either when the server is compiled or when it is installed in a particular system. Storage servers may attach their signature to any piece of data in the system and any piece of data may have any number of signatures. Signatures are persistent and remain with a piece of data even when it is read and written by different storage servers. For example, a piece of text data may have the signatures of a particular file storage server and an object storage server. The text data retains these signatures even when it is viewed and manipulated by other storage servers. Signatures must be explicitly removed from a piece of data.

Signatures are used to control the way storage servers access data and are similar to access flags that control how end users access files. When a storage server tries to read or write data that has attached signatures, the microstorage architecture notices the signatures and intercepts the access. The kernel locates the storage servers identified by the signatures and obtains permission from each of them to complete the read or write operation. Storage servers whose signatures are attached to a particular piece of data are called *interested servers* and the storage server trying to read or write the particular piece of data is called the *requesting server*. The kernel asks each interested server the following questions:

- Is the requesting storage server allowed to access the data?
- Will the interested server provide a translated copy of the data for the requesting server? Each data may include optional information describing its current format, such as ASCII text or portable bitmap. The kernel only asks this question when the requesting server provides an ordered list of data representations it is willing to accept.
- Is the requesting server allowed to change the format of the permanent copy of the data? This question is only asked if the requesting storage server wants to change the data and specifies a new format.

Any interested server may refuse access to data, or it may provide an edited version of the data that preserves other access controls. Different storage servers will use different criteria to regulate access. For example, a file storage server may use user access controls and the current user identity to enforce security. A database storage server may use the same access controls and user identity, but it may grant access to a restricted view of a database, rather than the underlying table. An object storage server may ask method procedures to grant access or convert data representation, which guarantees that the underlying data is only accessed through the method procedures, and preserves encapsulation.

The kernel implements a basic mechanism and different storage servers use it to implement very different access control policies for protection, concurrency and encapsulation. Because the mechanism is implemented by the kernel, it is reliably enforced, so it can be used for access controls even if some storage servers are not trusted. The kernel only intercepts access to data that has attached signatures, so there is a performance overhead only when storage servers require it. The kernel also implements an authentication scheme to prevent servers from impersonating other servers and prevents a rogue storage server from denying access to all data in the system. Authentication, and other issues regarding consistency are discussed in more detail in [26].

6 The Microstorage Kernel

The microstorage kernel stores and manipulates the basic unit of data that is used by storage servers to create files, objects, databases and other data structures. This unit of data is called a *segment* and is designed to be independent of any specific storage model so it does not force policy decisions.

6.1 Segments

A segment is a persistent piece of data that is stored on disk and may be loaded into virtual memory. When a segment is loaded in memory, it is accessed with a normal pointer and may be read and written like any other piece of memory. Segments in memory must be explicitly saved back to disk to save any changes. This is the only way to read or write a segment: first load it into memory, read or write its contents like any other piece of memory, then save it back to disk if it has been changed.

When a segment is being loaded into memory, virtual memory is allocated for the entire segment, so programs may not load individual disk blocks. Segments, however, may or may not be paged. If a segment is not paged, then all of its contents are copied from disk into memory when the segment is loaded. If a segment is paged, then disk blocks are only copied into memory when a client program first tries to read or write each page in memory. Loading a paged segment is analogous to mapping a file; it sets up system tables and allocates virtual memory but the actual data is demand paged. Generally paging improves system performance, but there are good reasons why some segments should not be paged that will be explored in [26]. Any segment may be individually marked as paged or non-paged, though we will assume in all examples that segments are paged.

Segments may be virtually any size, ranging from 100 bytes to 1GB. This means storage servers may use segments for small program variables like structures and arrays or for large things like a multimedia movie with stereo sound. This wide range of sizes also means the implementation must be efficient for both very small and very large segments.

Segments are identified by a persistent globally unique identifier. This identifier is an opaque handle that is used by storage servers to specify a segment to be loaded into memory or to link segments. An identifier is associated with each segment when the segment is created and is always valid, even when the segment is loaded in memory. While a segment is loaded in memory, however, it is usually named by a memory pointer. Segment identifiers are globally unique so they can specify segments on other networked machines. Kernels on different machines can communicate, so accessing a remote segment appears no different to the storage server or application program than accessing a local segment.

Segment identifiers are a simple naming mechanism and each storage server may implement its own name space with its own mechanism for associating names to segment identifiers. Segment identifiers are location dependent, and include information about the location of the segment on disk. They resemble an inode number in Unix, which simplifies the microstorage kernel, but means the microstorage kernel cannot transparently migrate segments. If a particular storage server needs to implement segment migration, then it must create its own location independent name space and map those names onto location dependent segment identifiers.

The microstorage kernel implements library procedures that manipulate segments, such as load, save, create and destroy. There are no methods or types for segments, although a storage server, the object storage server, implements these features and is discussed below. The microstorage kernel is designed to implement a basic mechanism and types and methods require policy decisions that should be defined by the storage servers.

6.2 Segment Properties

A property is a piece of data that is associated with a segment. There is no limit to the number of properties a segment may have. Properties are not stored in the memory occupied by a segment when it is loaded, instead storage servers and client programs use kernel library procedures to read and write properties. A segment must be loaded into memory before any of its properties may be accessed, but reading or writing properties does not cause pages of a paged segment to be swapped into memory. Properties are persistent, so they remain associated with a segment until explicitly removed, even if the segment is saved to disk and later loaded again into memory.

Properties are identified by two 4 byte values, called the property type and property id. For example, a particular segment might have several properties of the same type that contain the locations of backup copies of the segment. Each of these properties must have a different id number to distinguish them. A type-id pair must be unique for a particular segment, so a segment may have properties with the same type but different id's, or the same id and different types. Properties do not have to be unique across segments, so different segments may have properties with the same type and id.

Most properties are created by the storage servers or application programs that use segments, but a few special properties, those with type "systemProperty", are created by the microstorage kernel when a segment is first created. These system properties contain information used by the system to maintain segments, such as the size of the segment and when it was created. Although they are used by the system, these values may also be accessed by storage servers through the same library routines that manipulate all other properties.

6.3 Segment Links

The microstorage kernel also supports a special type of property called a *segment link*, or just a link, that may be attached to any segment. A link is a persistent reference from one segment to another that survives across loads and saves to disk. For example, if a segment contains a link to another segment and is saved to disk then loaded again into memory (possibly much later), the link still refers to the same segment. The microstorage kernel provides library procedures that allow client programs to create and delete links between segments and to use a link to load a segment into memory.

Each segment maintains a reference count of the number of links that refer to it so a segment cannot be deleted while there are links to it. Deleting a segment decrements this count and when the count reaches zero the segment storage is reclaimed. The microstorage kernel provides a garbage collector that reclaims garbage segments that have cycles of links and makes sure the reference count of each segment is accurate.

Storing links as properties allows any program to store and retrieve persistent pointers to segments. The object storage server, described below, provides a more powerful pointer swizzling mechanism, but that relies on programs providing type information for a segment. In particular, the object storage server has to know where the pointers are stored in a segment before it can update them. Links avoid this problem. Client programs explicitly create links, storing them as properties, so the kernel and other client programs do not have to read the segment body to read and write links. This allows the kernel to implement and use links and provide garbage collection without examining segment bodies, which depend on the data format.

6.4 Segment Attributes

When segments are loaded in virtual memory, they may have memory attributes associated with them. Memory attributes are a generalization of the concept of paging. When a client program tries to read a word of memory, the data may be unavailable for several reasons: the page has not been swapped in from disk, the page is part of shared memory and the latest version of the page is in another address space or on another machine, another process has locked that page while it is in its critical section, or the page contains a future that is the result of an asynchronous procedure call that has not yet returned. Memory attributes allow all of these conditions and many more to be cleanly implemented and used by the kernel, storage servers and client programs.

Memory attributes may be associated with an entire segment in virtual memory or any page of virtual memory. Each segment and page may be valid or invalid with respect to a particular attribute. For example, suppose that there is an attribute that signifies that the page contents have been swapped in from disk. A page is valid with respect to this attribute if its contents have been swapped in. Another attribute may signify that a page is unlocked by cooperating processes so any process may read or write it. A segment is valid with respect to this attribute if it is unlocked and available for use. Before any program may reference a word in virtual memory, all attributes associated with the segment and page that contain that word must be valid. If any of these attributes are invalid, then a pager procedure associated with each invalid attribute is called.

There is a single pager procedure for each attribute and the pager makes the attribute valid. For example, the pager for the attribute that signifies that a page has been swapped in from disk just copies the page data from disk. The pager for the distributed shared memory attribute will consult a system dictionary, find the latest version of a page and copy it in from the correct remote machine. A pager procedure for a lock attribute may just wait until the lock is released, causing the program to block until the process that holds the lock releases it. Vista defines several attributes and their pager procedures, but storage servers and client programs may define their own attributes and provide their own pager procedures. Attributes enforce a constraint before every access to memory, so they are a powerful mechanism for defining protocols for sharing segments in memory between programs. Storage servers may use attributes to maintain consistency when data is simultaneously being updated in several different models.

When an attribute is initially defined, it specifies whether it applies to individual pages of memory or to entire segments that may be several pages long. For example, an attribute that specifies demand paging disk blocks may apply to individual pages. After the pager for this attribute completes, the rest of the pages of the segment are still invalid, and the pager will be called again if the client program tries to read and write to any of those other pages. Other attributes, like the lock attribute, may apply to an entire segment. Once a shared segment is unlocked, then the entire segment is unlocked, and a client program may access any other page of the segment. In general, most attributes apply to the entire segment and only low level attributes (typically those provided by Vista) are aware of the underlying storage and page layout and can manipulate individual pages of a segment.

Attributes apply to a segment only when it is loaded in virtual memory, so attributes are not preserved across saves to disk. When a segment is saved to disk, its updated contents must be read from virtual memory and copied back to disk. By definition, all attributes must be valid before a segment may be read, so reading the segment makes all attributes valid.

7 The File Storage Server

The file storage model is important because it provides backward compatibility with existing applications, and because it provides block IO operations that allow separate pieces of a file to be individually loaded into memory. With block IO, applications can read and write files one piece at a time so they can read and write very large files, like multimedia files, that are larger than virtual memory.

The Vista file storage server implements files similar to Unix files, and provides standard library routines like creat, open, close, read, and write. Other storage servers could implement different file interfaces, such as the Macintosh, Windows, Amoeba, or Cedar file systems.

7.1 File Interfaces

The Vista file storage server implements files on top of the microstorage kernel so files do not exist by themselves, but are stored as trees of segments. The file server maps each file name onto a single segment and a segment that has an associated file name is called a *file segment.* Any segment may have a file name, and segments with file names may still be manipulated by normal segment operations just like any other segment. Typically, file segments will contain links

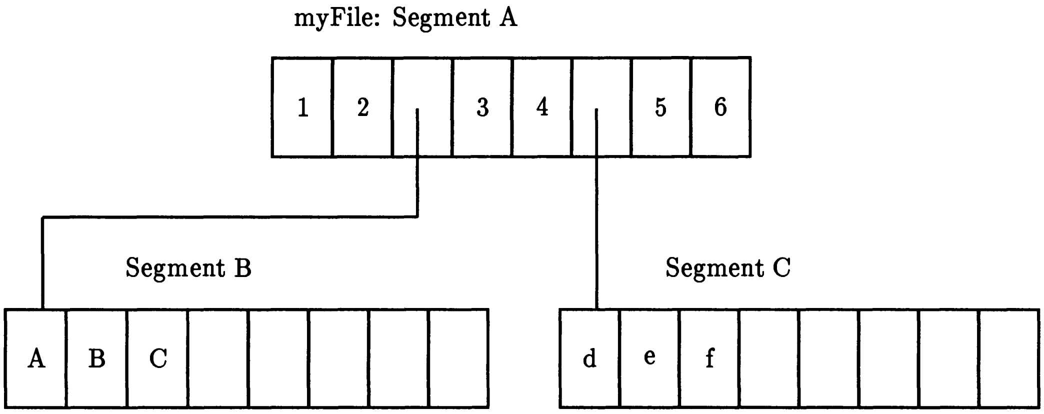

Figure 4: File Made of Segments

to other segments, although this is not necessary. If a file segment does contain links to other segments, then the contents of the file if the sum of the contents of all segments that are directly or indirectly linked to the file segment. A file is the entire tree of segments rooted at the file segment.

The file storage server implements file system routines such as read and write that find data in various segments and create the illusion that the file data is a seamless byte stream. These routines treat a segment similar to a disk block in a traditional file system such as Unix [6, 51], and a segment loaded in memory is similar to a disk block in a block cache. The only difference between implementing a file out of segments rather than disk blocks is segments consist of multiple disk blocks, so the Vista file system loads several blocks at a time when it loads each segment into memory. In effect, the only difference is the Vista file system prefetches and loads clusters of disk blocks into cache while a traditional file system loads one block at a time.

For example, suppose that the file storage server implements a particular file with three segments: a file segment and two linked segments (see Fig. 4). A client program could access this data as segments by first loading the segments into memory and then navigating the links. Alternatively, a program could open myFile, which maps to segment A, and read the same data as a serial stream of bytes. Because this serialization occurs in the file storage server, applications cannot tell the difference between a file that is made up of segments and a normal file made up of disk blocks in a traditional file system.

This stream could be in several forms, depending on the order in which the linked segments are traversed. In this example, the contents of segments B and C could appear in the byte stream in three places depending on the traversal order.

Pre-Order Before the contents of segment A. The stream would look like: A B C d e f 1 2 3 4 5 6

In-Order At the link position, that is in the middle of the contents of A. The stream would look like: 1 2 A B C 3 4 d e f 5 6

Post-Order After the contents of A. The stream would look like: 1 2 3 4 5 6 A B C d e f

Intuitively, the in-order traversal order is the most natural, but different programs and different files may prefer different formats. The default is in-order, but programs may override this when they open the file and specify a traversal order. Notice that the link to segment B appears before segment C in the file segment, so segment B always preceded segment C.

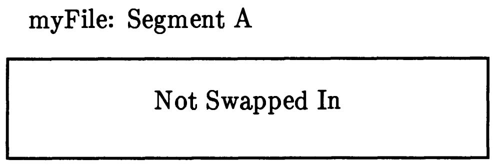

Figure 5: File In Memory After Load

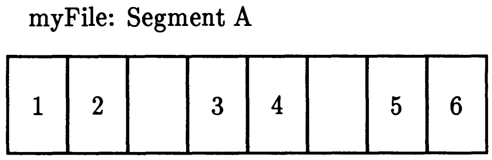

Figure 6: File In Memory After First Read

7.2 An Example

Consider a simple example of a program reading myFile described above. For this example, the segments are marked as paged, although they could be marked otherwise. Initially, no segments are loaded into virtual memory. When the application opens the file with the command:

```
fd = open("myFile", RDONLY);
```

the file storage server initializes some internal state and loads the file segment. Because the file segment is paged, none of its contents have been swapped in from disk yet.

Next, the client program reads the first two bytes of the file with the command:

```
result = read(fd, buffer, 2);
```

The file storage server tries to read these bytes in the file segment and this causes a page fault. To service this page fault, the microstorage kernel swaps the first page of the file segment from disk into memory. The file segment is only a page long, and it consists of the text, "123456".

The file attributes specify in-order serialization of the segments contents, so the file storage server starts to read from the file segment. The file segment also contains two links, and the file storage server uses the type and id of the links to decide where the linked segments should appear in the byte stream. For example, the link id could be interpreted as a byte position. This first read operation, however, does not require that linked segments be loaded into memory. The file storage server reads the requested bytes, "12" from the file segment and returns.

The program then reads the next 2 bytes, that includes text from the first linked segment. The file storage server loads the first linked segment into memory. The linked segments are also paged, so the file storage server causes a page fault when it first tries to read segment B. The microstorage kernel services this page fault and swaps in the text for segment B. The file storage server completes reading this segment and returns the text.

The client program reads the rest of the file and the storage server returns the remainder of the segments in a similar manner. The file storage server reads segments, and the microstorage kernel pages in their contents as they are read.

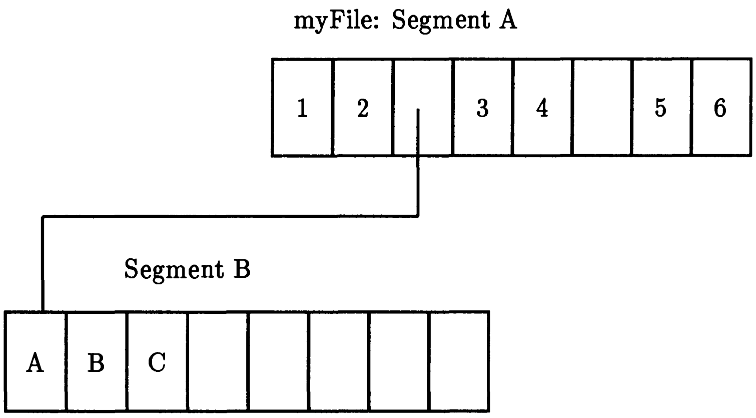

Figure 7: File In Memory After Second Read

7.3 Creating and Manipulating Files

A client program may create a file in one of two ways: declare an existing segment to be a file segment, or use file operations such as the Unix creat system call. The first method only requires that the client program identify a segment and provide a name and directory for the file. The segment becomes a file segment, and appears in the file storage server name space as a file. A single segment may appear in the file name space in several places, and, like links in the Unix file system, all of these file names refer to the same segment, not copies of the segment. A segment is never deleted until all file names and links that refer to it have been removed.

Identifying segments as files requires a system routine not normally found in traditional file systems. The file storage server provides several such special routines that may be used to manage the segments that make up a file. A normal application program does not have to use any of these special routines; they are provided only for programs that wish to manipulate individual segments. These routines include making a segment into a file segment, making a file segment into a normal (non-file) segment, and returning a pointer to the segment associated with a file name. A program that uses these routines will also likely use other routines provided by the microstorage kernel or other storage servers to manipulate segments.

When application programs create files through the standard file interface, the file storage server must create these files out of segments. Segments may be any size, so it is possible to create a file out of a single segment. Reading or writing a segment, however, requires that the entire segment be loaded into virtual memory so segments should not be too large. Files may realistically be very large, up to several hundred megabytes for files such as multimedia movies, and this will often be too large to load into virtual memory. To solve this problem, when a file is created with a file operation, the file storage server always stores the file data in a tree of segments. Segments at the leaves of the tree contain the actual file data and are limited to a reasonable maximum size. The internal tree nodes are segments that contain links to other internal nodes and leaf segments. This organization treats segments like disk blocks, and the file segment is similar to a Unix inode.

8 Object Storage Server

The object storage server implements a storage model that organizes all data into objects. Each object is implemented by a single segment, but it includes type information and is manipulated by method procedures. Objects resemble data in an object oriented programming language; objects are typed in a class hierarchy, objects are only manipulated by method procedures, methods are bound to objects at runtime and methods support operator overloading and inheritance. Objects are designed to provide a more powerful interface for application programs that manipulate individual segments. They are more powerful because they enforce the policy decisions of a specific storage model. Of course, no application program must use this model, it is simply one of several storage models that programs may use.

8.1 Classes and Formats

Object types perform two functions: they describe the semantics of an object and they describe the representation of an object. Object semantics define the concept an object models, such as a list of numbers or an employee record. The semantics do this by defining which operations may be performed on each object, effectively binding method procedures to a class of objects. The object representation describes how the contents of an object is represented in terms of primitive data types such as integer and character. Some, but not all, semantics may infer a representation, for example employee records may always have the same format.

Typing objects lets the object storage server perform some powerful functions on behalf of client programs, but it may also be unreasonable to require a program to type all of its data. For example, consider typing every byte in every file in a typical file system serving a community of users. Even though application programs do not have to use objects at all, the model should be flexible enough to encourage their use. Moreover, some objects may also change their representation every time they are stored, yet still maintain the same object semantics and use the same operations. For example, some graphic objects are really bags of tagged data, and two different objects may have very different contents and internal representations even though they are manipulated by the same procedures.

Because of these issues, object types are designed to be flexible and non-obtrusive. Each object has two separate pieces of type information: the class, that describes the operations that may be performed on the object, and the format, that describes the actual representation of the object. Every object must have both a class and a format, but the object storage server provides simple defaults for both. Besides convenience, separating type and format preserves abstraction. A program does not have to know the internal representation of an object to use it or call its methods.

8.2 Object Classes

Classes are organized in a hierarchy, as they are in most object oriented programming languages, although each class must have exactly one parent, enforcing single inheritance. The object storage server provides some standard classes, and client programs may define new object classes by calling library routines. Each method is identified by its name and the classes of it arguments, so a single method name may be defined differently for different classes. Classes inherit methods from parent classes, so an object of a particular class may be manipulated by a generic method from an ancestor class. When a method is applied to an object, the object storage server dynamically locates the appropriate method definition using the class of the object, the method name and the class hierarchy. If there is a method with the specified name that takes an argument of the object's class, then that method is invoked. Otherwise, the method with the same name that takes an argument of the nearest ancestor class is used.

Methods are stored locally on each machine that uses the object storage server, even if the actual data is stored remotely on a central server in a distributed system. Many distributed systems contain machines with different architectures and different versions of different operating systems. Each machine will need its own implementation of the object methods that works with its processor and operating system. These are then made available to the clients running on each machine. If methods were stored with the data, as they are in many object oriented databases, then the objects could only be used by the particular architectures and operating systems supported by the methods. Installing new methods in a central store to support each different client is difficult, especially in a large rapidly changing system.

Instead, each client loads a local copy of the methods it uses. There is no global namespace for classes or methods, so each client may have a different version of the same methods or completely different methods. For example, one client running on a Sun workstation may have a **length** method defined for stacks, while another client on a DecStation may not. Because classes are stored in a local name space, each client must also create its own class hierarchy. The local name space is also volatile, so client programs must recreate it every time they run. Typically, clients install classes and methods by loading a precompiled library, like the utility libraries in any operating system, and these libraries add to the local class hierarchy as part of their initialization. The number or types of methods in a class is not fixed, so new methods may be dynamically defined at any time.

Application programs should also use methods when they need help from other programs to display or edit data in a rich document. For example, a text file may contain an embedded CAD illustration. A word processor could ask the operating system to run a CAD program and send the program a message asking it to draw the illustration. The CAD program, however, would have to behave differently than if it were launched by a user. In particular, the CAD program would not display its own user interface, such as a document window, or do all of the expensive state initialization, when it is only being asked to display a small illustration. For example, LATEX should not run, with all the associated overhead, every time a program asks LATEX to locate a paragraph in a LATEX object.

Application programs should instead call method procedures that manipulate embedded data. For example, a CAD program may define a class called "CADillustration", and provide method procedures that display and edit objects of this class. Other programs, such as a word processor, may either load this class into their local name space, or else call a central dispatch program that has loaded the libraries for several different classes. Methods support overloading and inheritance, so generic methods may manipulate objects of descendent classes and different programs may supply different methods for a single class. Method libraries also cleanly separate the functionality of manipulating individual objects from the functionality of manipulating entire documents in the main application program. Commercial software companies may distribute a runtime package, that simply installs the methods for manipulating embedded objects separately from their main application program. This means users do not have to own the main program to perform simple operations on objects managed by that program. Companies can also enforce software licenses, since each program may have to install a local copy of each method library.

8.3 Object Formats

Object formats specify the representation of an object, and allow the object storage server to perform byte swapping and other conversions when an object is copied between disk and memory. These operations guarantee that the object contents are correctly represented in the local architecture and programming language. For example, Sun and MiPS processors use different representations for 32 bit integers while the Motorola 68000 and 601 processors align 32 bit integers differently. Besides byte swapping, object pagers may also recursively load in linked segments. Object formats can include a type for *persistent pointers*, which specify segment links that are converted to pointers every time an object is loaded into memory. Converting a link into a pointer means loading the linked segment, and storing its pointer somewhere in the body of the object that contained the persistent pointer. The effect is a pointer to a segment that is always valid, even when both segments are saved to disk and loaded again later. The object format specifies which link to use for each persistent pointer, as well as where the pointers belong in the object body.

The object storage server stores the format description with the object, so it is available whenever the object is loaded into memory. Some systems, like Sun's XDR, may use format information to translate data as it is passed between networked machines, but then discard the format information when the data reaches its destination. Storing the format with the object on disk means programs do not have to declare the format to load an object, which speeds loading and preserves abstraction. Saving the format on disk also makes the secondary storage behave like a tagged architecture. Any program may read the contents of any object on disk, even an object it did not create and, for example, find all pointers or all floating point numbers.

A storage server or an application program declares an object's format when the object is created or its format changes. The object storage kernel provides a collection of library procedures to define formats, similar to the XDR procedures in SunOS [70]. Each routine is used to declare a different primitive data type in the object; the routines take as an argument a pointer to the location of the primitive data type in the object. For example, if a client program is defining the format of an employee record that consists of three integer fields and a character array, it would call the integer library procedures three times with pointers to each of the integer fields, then the string type procedure for the character array. Classes and application programs may also use the declaration routines to define an object schema, which is a template that defines the format of objects like an abstract type in programming languages. When an object is defined, its format may be specified by an existing schema instead of using the type declaration procedures.

Different object stores use different mechanisms, such as formal grammars, for declaring formats. No one strategy is optimal and each will please some application programmers and displease others. The Vista object storage server uses a mechanism similar to XDR because it is familiar to many programmers and it makes it easy to create many different types if many objects have unique formats. It is also not difficult to parse commands written in a data definition language or format grammar and compile the specification into calls to the library procedures. Finally, different object storage servers may use different strategies.

8.4 Standard Object Methods

Although each object must have both a class and format, the object storage server provides convenient defaults that can be used for each. The default format is an array of characters, so the object is treated as a sequence of bytes like a Unix file. The default class is called *object* and is the ancestor of all client-defined classes. This default class provides methods that perform basic operations on an object. Because methods may be overloaded, other classes may define more specialized versions of these standard methods. In fact, several methods for the *object* class do little by themselves and are intended to be overloaded by each new class.

The methods for class *object* fall into three categories: low, medium, and high level. The low level methods include routines to create, duplicate and delete an object. These methods treat an object as a sequence of bytes, and actually just call corresponding microstorage kernel routines. Middle level methods include loading and saving objects. Depending on the object's format, these methods convert the object between its representation on disk and its representation in memory, which may include byte swapping and pointer swizzling. The high level methods include parse, display and edit, which have different semantics for each class. Parse converts an object into different representations, such as changing between graphics standards, and is used when different storage servers want an object stored in different representations. The display operation draws an object on an output device and is designed for displaying rich documents in a window. The edit method processes user actions, and creates an embedded editor for an object so different objects in a rich document may be edited differently.

Finally, the *object* class includes a method for enumerating objects that is a generalization of the query mechanism in most object oriented databases. Besides supporting query, enumeration is used for such tasks as garbage collection and backup. First, an enumeration specifies a set of input objects. This can be:

All objects of a specified class that are stored on a specified machine. This may be the local or a remote machine.

All objects that are directly or indirectly pointed to by an object. This means all objects that are descendents of a top level object. Typically, this top level object will be a file segment, meaning it implements a file in the Vista file system. In that case, this is the same as all objects of a specified class in a specified file.

All objects in an index. Object indexes are implemented as a special class of object that is provided by Vista.

Next, the enumeration performs some action on some object of the input set. A simple query may specify calling a predicate method for every object, while other routines, like garbage collection, may mark the object as live, or backup the object or call some other method. Finally, the enumeration may, depending on the result of the action performed on the object, place the object in the output set. The output set is an object that contains links to all the member objects. This output is also a legal input to another query, so queries may be nested.

9 Implementing End User Features

Microstorage architectures are designed to support a new generation of applications that provide end user functions like those described earlier. It is useful to review how a microstorage architecture implements those specific functions.

To implement each of these functions, an application program first uses a file name to locate a file segment and then manipulates the file segment and other segments in the file as objects. If the objects belong to the *object* class, then the methods perform simple operations. For example, a query on an object of class *object* searches for byte patterns in the object. Existing programs that create files with Unix file operations are not aware of objects or segments, so they create files that consist of *object* objects. New applications will use segments and objects, and can create files of objects that belong to richer classes. For example, a word processor may create a document of paragraph objects, and a database program may make a document of record objects. Methods on these richer objects will be more powerful. For example, a query may find all resistors in a circuit design that have a specific resistance.

Rich Documents Documents are trees of segments, and different segments may correspond to objects of different classes. To display a rich document, a program invokes the display method on each object. New applications and segment types are easily accommodated.

Navigational Links Segments contain links to other segments, so end users can easily follow links between documents. An application program can graphically represent a link, and the user follows that link with some command, such as clicking on the link with the mouse. The application program asks the kernel to load the linked segment, and then asks the file system to open the file that contains that segment.

Context Sensitive File Contents The display and parse methods may filter the contents of an object depending on the current application, user, access controls or other conditions. An application may explicitly call the display and parse methods on segments. If an application instead views data as a file, the microstorage kernel still guarantees consistency between models. The kernel may still call the object server as an interested server, so all data is still be filtered through the parse and display methods.

Perform Queries On Any Document File sections are implemented by segments, so to find interesting sections in a file, an application program submits a query to the object server, asking it to find all matching segments that are directly or indirectly linked from the file segment. To create a file from the output of a query, an application creates a file segment that contains links to all segments returned by a previous query. To find a file with a query, an application performs a query on all segments in the file system, and returns the files that contain matching segments.

Compatibility and Extensibility Segments and objects may still be viewed as streams of bytes in a file, so data is backward compatible.

Distribution and Heterogeneous Systems The microstorage kernel supports remote segments, and they appear no different than local segments. The object storage server performs byte swapping and alignment and pointer swizzling when an object includes format information.

10 Related Work

Microstorage architectures and the Vista system in particular support a variety of different storage servers and some of these resemble other storage systems, such as research and commercial object oriented databases, file systems, and object stores. Vista should be seen not as a specific alternative to these systems, but rather as an architecture in which these systems might be implemented as storage servers. If they were implemented as storage servers, then these systems would be able to use various features of Vista and seamlessly share data with other storage servers supporting different models.

10.1 Object Oriented Databases

There is not yet widespread consensus on which specific features must be included in an object oriented database, but there does seem to be some agreement [5] that it should combine the features of an object store and a traditional database system. In particular, this means its storage model should define data as objects that are organized in a hierarchy of classes and are bound to methods at runtime. It should also support data definition and manipulation languages and transactions. There have been several such systems built [8, 9, 31, 43, 44, 59].

A microstorage architecture like Vista is an excellent platform for object oriented databases. The microstorage kernel implements segments that provide the storage for persistent objects. An object oriented database might also use the facilities of the Vista object storage server, which implements object classes and formats, late method binding, inheritance, operator overloading, and a general query mechanism. The type system in Vista's object storage server is flexible, and the separation of class and format is similar to the distinction between types and classes in systems such as O2. An object oriented database system would still have to implement additional policy decisions on top of Vista, however, such as the semantics and syntax of data definition and manipulation languages, as well as implementing transaction support.

10.2 Related Object Stores

There have been many different objects stores, and different systems use objects to address different problems. Some object stores, such as LOOM [42], provide a persistent storage mechanism for programming languages. LOOM implements a persistent object base for Smalltalk that runs on a Xerox Dorado. LOOM was designed to provide a large virtual address space for objects and focuses on memory management issues, such as compacting physical memory. Each LOOM object has a unique 32 bit object identifier, even though the Dorado has a smaller address space. Clients only access objects with references that behave like a pointers and LOOM intercepts every read or write to a reference. This allows LOOM to load objects from disk into memory when they are first accessed and relocate objects once they are in memory.

LOOM provides a large address space on a machine with a small memory, but other object stores provide programming languages with persistent storage that has different design goals. Some object systems implement an object model with a compiler and attack problems in parallel and distributed programming [12,17,55]. For example, the Eden operating system [15,48], written in the Eden language, and the Emerald [16,41], and Amber [20] languages as well as Distributed Smalltalk [11] use this approach to implement distributed objects. Eden, Emerald, and Amber define strongly typed objects and operations on objects are performed by sending a message to the object. All three languages identify objects with global persistent identifiers that are location independent. Objects may migrate through the system and an object invocation mechanism in the languages will find the (local or remote) object and invoke a method associated with it. These systems solve problems in distributed computing such as remote execution, concurrency control, and migration.

Still other object stores are not associated with any language, but instead resemble a simple database without sophisticated data definition and manipulation languages or transaction support. For example, the Apple Newton implements an global object store that is used by all applications that run on each Newton. Programs use this as a general data store instead of a file system. Unfortunately, there is no way to seamlessly share data between a Newton *data soup* and Macintosh files. A microstorage architecture would provide this seamless connection, by implementing both files and Newton data objects, called *frames*, with different storage servers.

10.3 Application Architectures

Application architectures implement protocols for inter-application communication that allow several cooperating programs to display and edit a compound document and propagate updates between files that share sections. For example, Microsoft's OLE [57] supports rich documents with object linking and embedding and allows documents to share sections. Similarly, the Apple Macintosh Edition Manager [4] allows documents to share sections. Neither system, however, includes a storage architecture so neither addresses how the data that is shared between files or applications is actually identified and stored. Currently, both systems are built on top of a traditional file system; OLE on the Windows file system and Edition Manager on the Macintosh file system.

Application architectures such as these are excellent candidates to work with a microstorage architecture. The microstorage architecture provides a mechanism for storing, retrieving and manipulating persistent segments and storage servers can implement various file systems, such as the Macintosh or Microsoft file systems, or new file systems that directly implement object linking and embedding. Existing programs continue to access files through a traditional file system interface, while new programs directly manipulate the individual segments in a file either as segments or objects.

10.4 Related File Systems

As the Vista file storage server demonstrates, a microstorage architecture easily implements traditional file systems like the Unix file system. Microstorage architectures are also well suited for implementing more sophisticated file models, such as the PenPoint file system [19], that support embedding and navigational links. A user document in PenPoint may contain embedded objects, such as a movie inside a text document. Like OLE, PenPoint implements an application architecture, which PenPoint calls an application framework, but PenPoint also implements a storage architecture. PenPoint documents are implemented as DOS directories and the document contents are stored in individual DOS files. An embedded document in PenPoint is just a subdirectory in the parent document's directory. PenPoint uses DOS files in the way Vista files use segments. Vista could implement DOS files and directories, or PenPoint could be changed to directly embed objects with explicit link properties.

Other file systems, such as the Amoeba file service [58], save old versions of file blocks so previous versions of the file remain available. The Amoeba file service also supports *optimistic concurrency control*, and implements each file as a tree of pages. This closely resembles a tree of segments, and suggests both that the Amoeba file service could be easily implemented as a storage server and that microstorage architectures may use the ideas of optimistic concurrency control.

Several files systems, like the Intelligent File System [33], support locating files based on their contents and attributes. The intelligent file system then creates virtual directories comprised of all files that meet certain criteria. The intelligent file system, however, relies on file transducer programs that parse files and determine whether certain attributes apply. Like application architectures, this uses the standard flat file architecture and relies on parsing a file to implement new functionality. Finally, the GOOSE operating system [7] implements a form of dynamic file; files can have a program associated with them that is run when the file is opened normally. This program can dynamically generate the contents of the file or perform any related function. This is similar to opening a program in Plan 9 [61] through a file interface.

10.5 Related Segment Architectures

Multics [13, 24, 27, 60], implements a general purpose segment architecture. In Multics, client programs create persistent segments and access their contents with memory pointers. Segments do not have to be explicitly loaded into memory, because Multics implements a single level store. This means all stored data, whether it is in memory or on disk, is in the same global address space; a pointer to a segment is always valid even if the segment is on disk. Links between segments can simply be pointers, and since pointers are always valid there is no need for pointer swizzling. More importantly, access controls in a single level store are applied to segments so programs can share data and enforce access controls even when data is in memory. Multics does not implement some of the more modern ideas of object oriented programming like type hierarchies and operator overloading but it does support runtime binding of segment pointers and many more ideas in operating systems.

The Opal operating system [21] also implements a single level store, but it runs on 64 bit processors and uses 64 bit addresses as persistent global identifiers. Opal supports threads that execute in a protection domain, which is like an address space. Threads may attach segments to a protection domain, which is like mapping a segment into an address space. Programs may just use threads and segments, but Opal also supports an object model on top of the segment architecture.

Both Multics and Opal rely on hardware support for segmentation, and it would be difficult to port them to different architectures[2]. In Multics, for example, segments are demand loaded into *virtual memory* when a pointer to a segment is first dereferenced. Once the Multics segment has been loaded, the contents of the segment may be paged, so each is swapped in when it is first referenced. Vista can use memory attributes to page segments after they have been loaded into memory, but it cannot intercept every pointer access and demand load segments into memory without hardware support. Memory attributes are applied to a specific piece of memory, not any

[2] Multics uses segment registers, which are found in some of the most popular microprocessors.

uninitialized pointer and memory attributes have no means of mapping an uninitialized segment pointer to a segment identifier. Despite this limitation, Vista's object storage server resembles a single level store. Objects are persistent and memory resident and may contain persistent pointers that are valid across saves and loads.

10.6 Related Memory Management

Managing data in memory is an important part of a microstorage architecture, and there have been many related projects [71]. Memory attributes have been independently developed in several projects [3, 29] as a general purpose memory management tool. MENTAT [35, 36] implements futures for objects but did not define general client pagers. Mach [1, 32, 62, 64, 74] supports a notion of user level pagers that handle all page faults in a designated area. In Mach, client programs may associate any section of virtual memory with a user level pager procedure. Of course, Mach's most important influence on Vista is the idea of building a simple kernel that can support different policies in servers. This idea was also explored in systems such as Amoeba [72], V [22, 75], and Hydra [52, 78].

11 Current Status and Future Work

Vista is currently under development on a collection of Sun workstations under SunOS 4.x. The memory attributes are implemented with Unix memory management facilities, but the rest of the system is extremely portable and could be easily moved to systems as diverse as Macintosh, NT, DOS, Windows, VMS, and PenPoint. At the time of this writing, the microstorage kernel is running and work is underway on a file storage server that compatible with the Unix file system. Vista will support an NFS server, so workstations running Unix can use files stored on Vista without changing their local operating system. This works well for development, but Vista is designed to eventually replace the local file system.

To demonstrate its full potential, Vista needs several more storage servers, such as ones that implement a persistent programming language, a semantic net, a relational database, and a complete object oriented database. Each of these introduce open questions in their respective domains, as well as questions of how they work with a microstorage and other storage servers.

The object storage server also has open issues. For example, it currently locates methods in in the class hierarchy using only the the class of the object a method is applied to. This resembles C++ but another language, CLOS, might use a more suitable method. CLOS uses the classes of all the arguments to a method to find the most closely matching inherited method. For example, suppose that a program applies the insert method to arguments of type stack and boolean. CLOS may find two methods named insert, one that takes arguments of type set and integer and another that takes arguments of type set and list. Stack is a subtype of set and boolean is a subtype of integer, not list, so CLOS calls the first method. This is a flexible mechanism for inheritance and is currently under investigation.

The object storage server also does not currently support remote object invocation. The kernel can load and save remote objects, so programs can invoke local methods on remote objects, but they cannot invoke remote methods on local or remote objects. Remote method invocation would resemble a remote procedure call [14] and could use object format information to marshal arguments. There are also a number of different issues when objects are paged in a heterogeneous computing environment. Different processors and programming languages have different rules for data format and alignment, and this means that data on disk will have different sizes and formats in the memory of different systems. That makes it difficult to predict how large data will be when it is paged in, so it is difficult to locate where the data for a specific memory page is stored on disk. There are several possible solutions under consideration, including providing more elaborate format information and not paging typed objects.

Finally, there are several open issues that effect the microstorage kernel. A microstorage architecture may support many links throughout a large internet of computers, some of which may be disconnected at any time. This makes distributed garbage collection difficult. A microstorage architecture also shares design issues with microkernels, such as the performance penalty from crossing address spaces if several servers implement a single system call. Microkernels have begun to address this question, but more work needs to be done. Finally, fault tolerance has not been addressed, but it is essential in a storage architecture. Ideally, a microstorage architecture will provide a mechanism, and different storage servers implement various guarantees as individual policies. Fault tolerance typically has a performance penalty, and often adds implementation complexity, so it should be left to each storage server how to support it.

12 Summary and Conclusions

Current operating systems separate storage policy from mechanism by supporting a simple file system that makes few assumptions about how application programs use their data. A user document seldom looks like the stream of bytes that is stored in a file, so different programs implement their own data storage architecture on top of files. The result is each application program becomes an isolated world with its own private data representation and storage architecture, and it is difficult to move and share data between programs.

Each program has different storage needs, but common themes emerge. Instead of providing one simple mechanism, the operating system can support many different applications by providing many different policies. A microstorage architecture implements multiple storage servers, each implementing the policies of a different storage model, on a common mechanism, the microstorage kernel. This provides more powerful data storage tools for application programs, and makes the system extensible since new storage servers can be added at any time. Multiple storage models also support sharing between programs, since different programs, that perform different functions and use different storage models, may use the same data. Multiple storage models also provide a smooth transition from a file based view of data to an object based view.

References

[1] M. Accetta. Mach: A new kernel foundation for Unix development. In *Proceedings of the Summer 1987 USENIX Conference*. USENIX, 1987.

[2] R. Ananthanarayanan, Sathis Menon, Ajay Mohindra, and Umakishore Ramachandran. On the integration of distributed shared memory and virtual memory management. Technical Report GIT-CC-90/40, Georgia Institute Of Technology, 1990.

[3] A. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[4] Apple Computer. *Inside Macintosh, Volume VI*, chapter The Edition Manager. Addison-Wesley, 1991.

[5] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Proceedings of the First DOOD Conference*, December 1989.

[6] M. Bach. *The Design of the Unix Operating System*. Prentice-Hall, 1986.

[7] K. Bailey, L. Boynton, P. McKenney, G. Oliver, and D. Regan. User defined files. *Operating Systems Review*, 15(4), October 1981.

[8] F. Bancilhon. Object oriented database systems. *Proceedings of the Seventh Symposium on Principles of Database Systems*, 1988.

[9] Francois Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database System.* Morgan Kaufmann, 1992.

[10] Gilles Barbedette. Lisp O2: A persistent object-oriented lisp. In *Proceedings of the 2nd EDBT Conference*, March 1989.

[11] John Bennett. The design and implementation of distributed Smalltalk. In *ACM SIGPLAN Notices, Proceedings OOPSLA 1987.* ACM SIGPLAN, 1987.

[12] E. Bensley, T. Brando, and M.J. Prelle. An execution model for distributed object-oriented computation. In *ACM SIGPLAN Notices, Proceedings OOPSLA 1988*, pages 316–322. ACM SIGPLAN, 1988.

[13] A. Bensoussan, C.T. Clingen, and R.C Daley. The Multics virtual memory: Concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972.

[14] Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[15] Andrew Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 181–193. ACM SIGOPS, 1985.

[16] Andrew Black, Norm Hutchinson, Eric Jul, and Henry Levy. Object sructure in the Emerald system. *ACM SIGPLAN Notices, Proceedings OOPSLA 1986*, pages 78–86, 1986.

[17] J.D. Bos and C. Laffra. PROCOL a parallel object language with protocols. In *ACM SIGPLAN Notices, Proceedings OOPSLA 1989*, pages 35–47. ACM SIGPLAN, 1989.

[18] B. Caplinger. An information system based on distributed objects. In *ACM SIGPLAN Notices, Proceedings OOPSLA 1987.* ACM SIGPLAN, 1987.

[19] R. Carr and D. Shafer. *The Power of PenPoint.* Addison-Wesley, 1991.

[20] Jeffrey Chase, Franz Amador, Edward Lazowska, Henry Levy, and Richard Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth Symposium on Operating System Principles*, pages 147–158. ACM SIGOPS, 1989.

[21] Jeffrey Chase, Henry Levy, Edward Lazowska, and Miche Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *ACM SIGPLAN Notices, Proceedings OOPSLA 1992*, pages 397–413. ACM SIGPLAN, 1992.

[22] David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):313–333, March 1988.

[23] Helen Custer. *Inside Windows NT.* Microsoft Press, 1993.

[24] Robert Daley and Jack Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[25] Partha Dasgupta, R Ananthanarayanan, Sathis Menon, Ajay Mohindra, and Raymond Chen. Distributed programming with objects and threads in the Clouds system. Technical Report GIT-CC-91/26, Georgia Institute Of Technology, 1990.

[26] Dawson Dean and Richard Zippel. The vista object storage server. Technical Report In Preparation, Cornell University, 1993.

[27] Jack Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM*, 12(4):589–602, October 1965.

[28] J. Dion. The Cambridge file server. *Operating Systems Review*, 14(4), October 1980.

[29] D. Edelson. Fault interpretation: Fine-grain monitoring of page accesses. In *Proceedings of the Winter 1993 USENIX Conference*, 1993.

[30] David Redell et. al. Pilot: an operating system for a personal computer. *Communications of the ACM*, 23(2):81–91, February 1980.

[31] O. Deux et al. The story of o2. In *Transactions on Knowledge and Data Engineering*, March 1990.

[32] Alessandro Forin, Joseph Barerra, Michael Young, and Rick Rashid. Design, implementation, and performance evaluation of a distributed shared memory server for Mach. In *Proceedings of the Winter 1988 USENIX Conference.* USENIX, 1988.

[33] David Gifford and J. O'Toole. Intelligent file systems for object repositories. In *Proceedings on the International Workshop on Operating Systems of the 90s and Beyond*, Munich, 1991. Springer-Verlag.

[34] I. Greif and S. Sarin. Data sharing in group work. *ACM Transactions On Information Systems*, April 1987.

[35] Andrew Grimshaw. An introduction to parallel object oriented programming with Mentat. Technical Report TR-91-07, University of Virginia, 1991.

[36] Andrew Grimshaw and Jane Liu. MENTAT: An object oriented macro data flow system. In *ACM SIGPLAN Notices, Proceedings OOPSLA 1987*, pages 35–47. ACM SIGPLAN, 1987.

[37] Sabine Habert and Laurence Mosseri. COOL: Kernel support for object-oriented environments. *ACM SIGPLAN Notices, Proceedings OOPSLA 1990*, 1990.

[38] Robert Hagman. Reimplementing the Cedar file system using logging and group committ. In *Proceedings of the Eleventh Symposium on Operating System Principles.* ACM SIGOPS, 1987.

[39] Robert Halstead. MultiLisp: A language for concurrent symbolic computation. *ACM Transactions On Programming Languages And Systems*, 7(4), October 1985.

[40] Michael Jones and Rick Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. *ACM SIGPLAN Notices, Proceedings OOPSLA 1986*, 21(11):67–77, 1986.

[41] Eric Jul, Henry Levy, Norm Hutchinson, and Andrew Black. Fine grained mobility in the Emerald system. In *Proceedings of the Eleventh Symposium on Operating System Principles.* ACM SIGOPS, 1987.

[42] T. Kaehler. Virtual memory on a narrow machine for an object-oriented language. *ACM SIGPLAN Notices, Proceedings OOPSLA 1986*, 1986.

[43] W. Kim. Research directions in object-oriented database systems. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, 1990.

[44] W. Kim, N. Ballou, H. Chou, and J. Garza. Integrating an object-oriented programming system with a database system. In *ACM SIGPLAN Notices, Proceedings OOPSLA 1988*, pages 142–153. ACM SIGPLAN, 1988.

[45] W. Kim, J. Banerjee, H. Chou, J. Garza, and D. Woelk. Composite object support in an object-oriented database system. In *ACM SIGPLAN Notices, Proceedings OOPSLA 1987*. ACM SIGPLAN, 1987.

[46] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun Unix. In *Proceedings of the Summer 1986 USENIX Conference*. USENIX, June 1986.

[47] Butler Lampson. Hints for computer system design. In *Proceedings of the Ninth Symposium on Operating System Principles*. ACM SIGOPS, 1983.

[48] Edward Lazowska, Henry Levy, G. Almes, M. Fisher, R. Folwer, and S. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth Symposium on Operating System Principles*. ACM SIGOPS, 1981.

[49] Rodger Lea. COOL: an object support environment co-existing with Unix. *Proceedings of the AFUU Convention Unix 1991*, 1991.

[50] C. Lecluse, P. Richard, and F. Velez. O2, an object-oriented data model. In *Proceedings of the ACM SIGMOD Conference*. ACM, June 1988.

[51] Samuel Leffler, Marchall McKusick, M. Karels, and John Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley, 1989.

[52] Roy Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the Fifth Symposium on Operating System Principles*. ACM SIGOPS, 1975.

[53] Barbara Liskov. Preliminary design of the Thor object-oriented database system. *Proceedings of the Software Technology Conference 1992*, 1992.

[54] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*. USENIX, 1988.

[55] Steven Lucco. Parallel programming in a virtual object space. In *ACM SIGPLAN Notices, Proceedings OOPSLA 1987*. ACM SIGPLAN, 1987.

[56] Marshall McKusick, William Joy, Samuel Leffler, and Robert Fabry. A fast file system for Unix. *ACM Transactions on Computer Systems*, 2(3):182–197, August 1984.

[57] Microsoft. *Windows Programmer's Reference*. Microsoft Press, 1992.

[58] S. Mullender. A distributed file service based on optimistic concurrency control. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 51–62. ACM SIGOPS, 1985.

[59] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in the ObjectStore database system. *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 1992.

[60] Elliot Organick. *The Multics System: An Examination of its Structure.* The MIT Press, 1972.

[61] Bob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from bell labs. In *Proceedings of the Summer 1990 UKUUG Conference*, July 1990.

[62] Rick Rashid. From RIG to Accent to Mach: The evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society 1986 Fall Joint Computer Conference.* ACM, 1986.

[63] Rick Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 64–75. ACM SIGOPS, 1981.

[64] Rick Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computer Systems*, August 1988.

[65] L. G. Reed and P.L. Karlton. A file system supporting cooperation between programs. In *Proceedings of the Ninth Symposium on Operating System Principles.* ACM SIGOPS, 1983.

[66] David Rosenthal. Evolving the Vnode interface. In *Proceedings of the Summer 1990 USENIX Conference.* USENIX, June 1990.

[67] J. Saltzer. *File Indexing and Backup.* Springer-Verlag, Munich, 1991.

[68] Singleton and Bennett. A single model for files and processes. *Operating Systems Review*, 20(1), January 1986.

[69] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1986.

[70] Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043. *Network Programming*, 1990.

[71] Ming-Chit Tam, Jonathan Smith, and David Farber. A taxonomy-based comparison of several distributed shared memory systems. *Operating Systems Review*, 24(3):40–67, July 1990.

[72] Andrew Tanenbaum and S. Mullender. An overview of the Amoeba distributed operating system. *Operating Systems Review*, 15(3), July 1981.

[73] A. Tannenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, 1987.

[74] Avadis Tevanian, Rick Rashid, David Golub, David Black, Eric Cooper, and Michael Young. Mach threads and the Unix kernel: The battle for control. In *Proceedings of the Summer 1987 USENIX Conference.* USENIX, 1987.

[75] M. Thiemer, K. Lantz, and David Cheriton. Preemptable remote execution facilities for the V system. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 2–12. ACM SIGOPS, 1985.

[76] Robbert van Renesse, Hans van Staveren, and Andrew Tanenbaum. Performance of the world's fastest distributed operating system. *Operating Systems Review*, 22(4), October 1988.

[77] Paul Wilson. *Uniprocessor Garbage Collection Techniques.* Springer-Verlag, Munich, 1992.

[78] W. Wulf, Roy Levin, and C. Pierson. Overview of the Hydra operating system development. In *Proceedings of the Fifth Symposium on Operating System Principles.* ACM SIGOPS, 1975.

[79] Michael Young, Avadis Tevanian, Rick Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, D. Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 63–76. ACM SIGOPS, 1987.

[80] Roberto Zicari. A framework for schema updates in an object-oriented database system. In *Proceedings of the 7th IEEE International Conference on Data Engineering*, April 1991.