CMU-ITC-83-032

ITC File System Design

29 September 1983

Information Technology Center
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

PREFACE

This is the proposed ITC file system design, representing a consensus of the file system group. Its basic architectural elements are unlikely to change much, but you can expect that the design decisions will change as we begin implementation and find oversights and errors. The intent of distributing the proposal at this stage is to elicit comments and help in avoiding such errors.

The discussion of the design falls into two parts: the base design and refinements. The latter typically constitute optimizations or alternatives which are consistent with the base design, but are not essential for the functioning of the system. The decision to incorporate these (and other) refinements can be deferred until the base design has been implemented and used.

The structure of this document reflects the division into a base design and refinements in two ways. First, the early chapters (particularly those on the VICE and VIRTUE interfaces) delineate the core parts of the design, leaving later chapters to fill in details, support packages, and applications. Second, each chapter presents a base case followed by refinements within its particular functional area.

This document is not yet complete, and may never be. A number of open questions are identified at the ends of the various chapters. These and other issues must be resolved before the skeleton presented here is adequate to serve as the basis for an implementation.

The file system group consists of:

John Howard (group leader)
Mike West
M. Satyanarayanan
Dave King
Dave McDonald
Alfred Spector (consultant)
Rick Rashid (consultant)
Dave Gifford (consultant)
Mike Young (consultant)

We thank the entire ITC staff and management, who have helped review the proposal as it evolved and contributed numerous ideas and observations to the design and our discussions.

Introduction **1**
Overview 1
Scenarios 2
  Ordinary file access 2
  Sharing 3
  Mail and Bulletin Boards 3
  Local Files 4
  Independent Operation 4

The VICE File System **5**
VICE System Calls 5
  ConnectFS 6
  DisconnectFS 7
  Lock 8
  Unlock 10
  WhereIS 11
  Fetch 12
  Store 13
  Remove 14
  GetFileStat 15
  SetFileStat 17
  MakeDir 18
  RemoveDir 19
  Link 20
  Unlink 21
  Rename 22
VICE Implementation 23
VICE Refinements 23
  PreFetch 24
  Cfetch 24
  Cluster caching 25
  Checkin/Checkout 26
Unresolved VICE Issues 26
  Locating custodians 26

The VIRTUE File System **29**
VIRTUE System Calls 29
  Open 30
  Close 31
  Fsync 32
  Read 33
  Write 34
  Lseek 35
  Remove 36
  GetFileStat 37
  SetFileStat 38
  MkDir 39
  RmDir 40
  Partition 41

UnPartition 42
SetCacheMode 43
VIRTUE Implementation 44
Caching Files 44
Local files 45
Locating Custodians 45
VIRTUE Refinements 45
Partition 46

Authorization 49
Login service 49
Check Authorization 50
Logout 50
Maintenance 50
Issues 50

Accounting and Disk Accounting 51
Accounting 51
Data bases 52
Login 52
Server processes 52
Logout 53
Daily billing 53
Monthly billing 53
Offline billing 54
Maintenance 54
User-to-User transactions 54
Disk Accounting 55
Disk quotas 56

Protection 57
Introduction 57
Protection Domain Management 58
Protection Subsystem Rights 60
Protection Subsystem Calls 61
Sketch of Protection Subsystem Implementation 62
File System Protection 64
Simulating Unix Protection 65
Implementation Phases 66

Additional Topics 69
Paging and Disk Servers 69
Archiving and Backup 69

Mail, Bulletin Boards, and Printing 73
Mail 73
Storage 73
Usage 74
Bulletin Boards 74
Storage 75
Usage 75
Accounting 75
Maintenance 76

Printing 76

INTRODUCTION

In this chapter we give you an overview of the ITC file system. Its purpose is to tell you what using the VICE file system will be like in general, without getting into all the detail you'll see in later chapters.

If all you want is an overview, read only this chapter. The later chapters go into much more detail and are intended to be used as reference material and as implementation guidelines, not for general consumption.

OVERVIEW

VICE provides communications and shared file storage for the ITC network of workstations. Communications, including the local area network and also the set of "gateways" to existing campus computers, is discussed elsewhere, as is the VIRTUE workstation. Here we describe the file system, including storage and movement of files, protection and authentication, accounting and space control, dealing with network failures and independent operation, and certain applications such as mail and bulletin boards.

A *file* is simply a sequence of bytes. It need have no particular character codes or record structure, although one may expect that many files will contain ASCII text separated into lines by New Line characters.

Files are stored in *directories*. A directory contains a collection of files and other directories, each identified by a unique (within the directory) component name.

You can identify any file or directory by stringing together component names, each preceded by the ASCII character "/". Such a string is called a *pathname*. The pathname "/" refers to the *root* directory, whose name and location are built into VICE. As you append components to a pathname, you identify subdirectories until eventually you arrive at a file. For example, to find the file associated with "/usr/bovik/jokes" VICE looks up "usr" in the root directory, then "bovik" in "/usr", and finally "jokes" in "/usr/bovik".

Ordinarily you can write programs and manipulate files as if the file system were stored on a single very large computer system. The workstation will contain interfacing software whose job it is to insulate you from the details of network interfacing, copying files to and from the workstation, and so on. However, the network won't be completely invisible:

- You'll notice performance differences because locally stored files are faster to get to.

- You can, if you wish, control file placement, bypass the interfacing code, or even write your own operating system.

- The file system will attempt to provide (possibly limited) services even if the workstation is disconnected from the network or the network itself is partitioned.

Therefore, a little terminology about the structure of the network is in order.

The network will be organized into interconnected "clusters" of workstations. Each cluster includes some number of workstations and at least one "cluster server" machine, which stores files and runs other VICE applications. For security and reliability reasons, cluster servers run only VICE applications. The hardware and software of the attached workstations may differ, but they can all use the services provided by VICE if they support the underlying communication and application protocols.

Within this framework, there are two file system interfaces:

- the VIRTUE interface, through which programs and user commands running in the workstation see files stored in VICE and/or locally.

- the VICE interface, used by the workstation (and potentially by any machine connected to the network) to store, share, and retrieve files.

You will seldom deal directly with the VICE interface, which is really intended for machines, and will usually deal with the VIRTUE interface by running programs which read and write files.

SCENARIOS

What will using the file system be like? This section attempts to give you simple examples, with most of the detail suppressed, of what things you can do with the ITC file system.

Ordinary file access

When you start using a workstation, you "log on" by providing a password and your name. The workstation goes through an authentication procedure to establish a secure connection with VICE. Since only your workstation can use the connection, and you're in control of the workstation, VICE treats the connection as your authorized representative. (It will be up to you to "log off" when you're done, to prevent somebody else from misappropriating the connection and masquerading as you.)

When you first use a file, the workstation will copy it automatically to its local "cache" of files. Thereafter it will use the cached copy directly, checking with VICE to verify that nobody else has updated the master copy. If you modify the file, the workstation will copy it back to VICE automatically. If you stop using the file, the workstation will eventually discard its local copy in order to make space for other files.

Some widely-used files, such as compilers or popular applications programs, will be replicated at every cluster server. VICE will update such files periodically, for example once a day. You can force an update if you really must have the most recent version.

Sharing

You can use your files from any workstation, regardless of where they're actually stored. For example, you can log on to a public workstation anywhere on campus and use your files normally, although possibly with a performance penalty if the files must be moved a long distance.

Only you have access to your files to start out with. You can, however, allow other users or groups of users to perform selected operations on your files. For example, a research project can set up a shared set of files which any member of the project can read and update.

Mail and Bulletin Boards

Mail and bulletin boards are a widely-used example of file sharing. VICE will support them through its tree-structured directory. Every user who wishes to receive mail will have a "mailbox" subdirectory into which other users are permitted to store messages in the form of files. Only you can remove and read messages from your mailbox.

Bulletin boards will resemble mail, except that a whole group of users can read the messages. Stale messages will be removed by the bulletin board's owner.

VIRTUE will provide application programs to generate, categorize, and read mail and bulletin boards; the file system simply stores them.

Local Files

You can force files to be stored only in your workstation by placing them in a special "/local" subdirectory of your workstation's file system. This provides some additional security for very sensitive files, but you won't be able to get at them from any other workstation. A more important application is to store the programs and data needed to get the workstation started and connected to the network when its power is turned on.

Independent Operation

If your workstation has sufficient local storage to hold your files and programs, you should be able to use it even if you take the workstation off-campus, or if there is some failure in the network or cluster server. This is a special case of what is called *partitioned* operation, meaning that the network has been divided into two or more independently operating parts which can't talk to each other.

Your workstation will retain file updates until reconnected to VICE, then merge your local changes with any that happened in the rest of the network. We expect that conflicting updates will be relatively rare, but will attempt to detect them by keeping some sort of update log.

The ability to operate independently requires that your workstation have a local disk with reasonable performance. We may also provide for diskless workstations (which would not be able to run independently) in order to reduce the minimum cost of a workstation. This decision depends on technological and performance considerations not yet available. Diskless workstations would depend on the cluster server to store almost all of their files.

The VICE file system stores simple sequential files, which it names using a hierarchical directory system. In many respects it resembles a UNIX[1] file system, and indeed it is designed to be easily used as a remote file server for UNIX-like systems such as VIRTUE.

Since VICE is a distributed system, different parts of this file hierarchy will be stored at different places. A VICE file location is called a *cluster server*. Every file and directory is stored in some unique cluster server called its *primary custodian*. Closely related files, such as the entire set of files owned by a particular user, usually will have a single custodian.

Custodians synchronize and control access to files. Usually you must check with the custodian before using a file, even if you have a copy. You can ask the custodian to lock a file, giving you temporary control so you need not check with the custodian each time you use it. Control reverts to the custodian when you unlock the file or disconnect from VICE.

Files may be replicated within VICE in several ways. First, some widely-used but seldom changing files or directories may have copies stored in *secondary custodians*, which cooperate with the primary custodian for updates and share the labor of fetching the files. Second, a PreFetch operation lets a non-custodian cluster server cache files as if it were a workstation. The backup mechanism also makes secondary copies. The difference is in updating the secondary copies, VICE always protects against inconsistent updates.

VICE SYSTEM CALLS

Although the VICE interface is really a protocol for messages between a workstation and VICE, it is described here in the style of procedure calls. We feel this gives a clearer picture of the intended operations than a direct list of message types and contents. You should realize that some of these VICE calls will require long exchanges of messages, and that most require at least a message from the workstation and a reply from VICE.

One implication of this style of description is that it deliberately omits certain important information. In particular, every system call is addressed to some particular cluster's file server, implicitly requiring an active connection. It is assumed this connection, which is initially set up using ConnectFS, is made secure by encryption or other mechanisms in the transport layer.

[1] UNIX is a trademark of Bell Laboratories.

ConnectFS

Format

ConnectFS(userid, token, clusterID)

Input Parameters

userid	The userid that is used to LOGIN to the system.
token	This is the token returned from LOGIN that is used to verify the identity of the user.
clusterID	The ID of the cluster to connect to. This can be obtained from LOGIN or from a WhereIS call.

Description

ConnectFS establishes a connection with a cluster's file server, identifying you for file protection purposes and authenticating the workstation as being under the your control. It is up to the workstation software to keep track of connections and to use the appropriate one for each instance of the other VICE calls listed here.

It is assumed that authentication has already been done as described in "Authorization" on page 49, returning a validation token and the local clusterID. The validation token establishes who you are, and the local clusterID identifies the cluster server to try first for new files.

Errors

Invalid token or userid

Cluster server not available.

DisconnectFS

Format

DisconnectFS()

Input Parameters

NONE

Description

DisconnectFS breaks the connection between a cluster's file server and a workstation. When it is issued, any outstanding locks held by the workstation or on behalf of the workstation are released. More generally, all state information concerning the particular workstation, such as checked out files or open file status, is reset by DisconnectFS.

Errors

Not currently connected

Lock

Format

Lock(pathname, READ|WRITE|NONE) returns (timestamp)

Input Parameters

pathname	names the file or directory to be locked.
READ	Requests a read lock on the file, preventing others from changing it.
WRITE	Requests a write lock on a file, allowing you to change it.
NONE	Sets no lock (just returns the timestamp.)

Output

timestamp	identifies the most recent version of the file.

Description

Locks prevent unexpected changes in files. READ locks prevent other users from setting WRITE locks. The Store operation, described later, requires a WRITE lock. Thus a READ lock protects a file from being changed. Several users may have READ locks simultaneously. WRITE locks exclude all other locks, both READ and WRITE. You should use a WRITE lock when you fetch a file with the intent of updating it.

Read locks allow you to synchronize access to files. You can fetch a file without locking it, accepting the risk that your copy will be made obsolete if somebody else stores the file.

The timestamp is returned so that you can verify that a cached copy of a file is still current. Note that the only function of Lock with the NONE option is to return the timestamp.

It is not necessary that a file actually exist for its pathname to be locked, but at least the containing directory must exist. This allows a new pathname to be reserved while the file is being created.

You must have write authorization to set a WRITE lock on an existing file, and create authorization to set a WRITE lock on a pathname for which no file exists. READ locks require read authorization; NONE requires lookup authorization.

A lock on a directory ensures only that the directory's meta-information does not change. It does not prevent new files from being added to the directory.

Errors

Cluster is not a custodian of pathname

Not authorized for requested kind of lock

Containing directory does not exist

File is read locked

File is write locked

Unresolved symbolic link

Unlock

Format

Unlock(pathname)

Input Parameters

pathname	The string representing the file or directory to have its lock released.

Description

Unlock releases locks. Once a write lock is released, it is no longer possible to store a copy of a file or update meta-information.

Errors

Cluster is not a custodian of pathname

No lock held

Unresolved symbolic link

WhereIS

Format

WhereIS (pathname) returns (clusterID, prefix)

Input Parameters

pathname	The string naming the file or directory whose location is to be determined.

Output

clusterID	A list of the clusters that are custodians of this file or directory.
prefix	The briefest prefix of pathname all the descendents of which have the returned clusterID.

Description

WhereIS identifies the custodian(s) of a file or directory. It also returns a prefix which the workstation may use to avoid further WhereIS calls for closely related files.

The list of custodians returned will always have at least one entry. Whether or not it ever has more than one is up to the implementation.

Errors

Pathname unknown.

Unresolved symbolic link

Fetch

Format

Fetch(pathname) returns (file contents,status)

Input Parameters

pathname	The string representing the file to be fetched.

Output

file contents	This is the byte string that makes up the requested file.
status	This is the file status, as described under GetFileStat.

Description

Fetch obtains a copy of a file. Although no explicit locks are required, the copy returned will not contain partial updates. In other words, Fetch operations never overlap Stores. If the file is to be written, or it is necessary that the file not change while the copy is examined, you should lock it before fetching it.

The file contents and status returned by Fetch are mutually consistent, and it is expected that the workstation will retain a copy of the status in order to verify that it has an up to date copy of the file.

Errors

Cluster is not a custodian for this file

Pathname is not a file

Pathname does not exist

Unresolved symbolic link

Store

Format

Store(pathname, file contents)

Input Parameters

pathname	The string representing the file to be stored.

Output

file contents	This is the byte string that makes up the file to be stored.

Description

Store creates and replaces files.

You must have a write lock on the pathname before writing a file. You must also have appropriate access privileges (discussed later.)

If you intend to read, modify, and update a file, and if other users also have access to the file, you should write lock it before you fetch it. Otherwise you could either overwrite somebody else's changes or be unable to obtain the necessary write lock in order to store your version.

Errors

No write lock held

pathname not a file

Cluster is not a custodian of the file

Unresolved symbolic link

Remove

Format

Remove(pathname)

Input Parameters

pathname	The string representing the file to be removed.

Description

Remove gets rid of a file. You must have a write lock before you can remove a file.

Errors

Cluster is not a custodian

Write lock not held

File does not exist

Not authorized to remove file

Unresolved symbolic link

GetFileStat

Format

GetFileStat(pathname, LINK) returns (status)

Input Parameters

pathname	The string representing the file or directory to queried.
LINK	This requests that any link information be returned explicitly

Output

status	This is the status data returned.

Description

GetFileStat obtains information about a file or directory. The status includes the following information:

- File name
- Size
- Protection Information
- Time stamps of creation, access, modification
- An optional file description or title
- Other information as needed

A particular format which allows easy extension and retrieval of specific items will be defined later.

The status returned for a directory includes information about all of the children of the directory, and if the LINK keyword is included will contain explicit, rather than resolved link information.

The file description is intended to make it easy to locate old files without actually fetching them. This may be particularly important for archived files.

Errors

Cluster is not a custodian

Pathname does not exist

Unresolved symbolic link

SetFileStat

Format

SetFileStat(pathname, status)

Input Parameters

pathname	The string representing the file or directory to be changed.
status	The new status information.

Description

This call is used to update the status information kept about pathname. A write lock on pathname must be held to update the information. To ensure consistency from a GetFileStat to a SetFileStat, a write lock would have to be obtained before the get, and not released until after the set.

Errors

Cluster is not a custodian

Pathname is invalid

Status information is invalid

Write lock not held

Not authorized to write file

Unresolved symbolic link

MakeDir

Format

MakeDir(pathname)

Input Parameters

pathname	The string representing the directory to be created.

Description

This creates an empty directory at pathname. Pathname cannot already exist, a write lock must be held on pathname and the parent of pathname must be a directory.

Errors

Cluster is not a custodian

Write lock not held

The parent of pathname is not a directory

File or directory already exists

Unresolved symbolic link

RemoveDir

Format

RemoveDir(pathname)

Input Parameters

pathname	The string representing the directory to be removed.

Description

This removes an empty directory at pathname. A write lock must be held on pathname.

Errors

Cluster is not a custodian

Write lock not held

Pathname is not a directory

Directory is not empty

Unresolved symbolic link

Link

Format

Link(source pathname, target pathname)

Input Parameters

source pathname The string representing the source pathname to be created.

target pathname The string representing the target pathname to which the source pathname will resolve.

Description

Link defines the source pathname to be a synonym for the target. All references to the source pathname are (ultimately) treated as if they were references to the target pathname. This is the symbolic link facility that currently exists in UNIX.

A write lock must be held on the source pathname. No checking will be done on the target pathname to ensure that it exists, uses valid directories, or refers to accessible files.

In the base proposal, VICE does not pursue symbolic links across cluster boundaries, and in fact may not pursue them at all. Instead it makes an error return whenever it arrives at a symbolic link, leaving further action up to the workstation.

This may lead to undue performance impact. Possible ways to reduce or eliminate this impact would be to cache symbolic links somehow in VIRTUE, or to follow some or all of them in VICE automatically.

Errors

Cluster is not a custodian for source pathname

Write lock not held

Unresolved symbolic link

Unlink

Format

Unlink(pathname)

Input Parameters

pathname	The string representing the source pathname of a symbolic link.

Description

This removes a link. A write lock must be held on the pathname.

Errors

Cluster is not a custodian

Write lock not held

Pathname is not a link

Unresolved symbolic link

Rename

Format

Rename(pathname, componentname)

Input Parameters

pathname	The string representing the file or directory to be renamed.
componentname	The replacement for the last component of the pathname.

Description

This renames a file or subdirectory within its containing directory. The last component of its pathname is replaced by the given component name, producing a new pathname. Write locks must be held on both names and the requests must be directed to the custodian of the old name.

Errors

Cluster is not a custodian

Old pathname does not exist

New pathname already exists

Write lock not held on old pathname

Write lock not held on new pathname

Unresolved symbolic link

VICE IMPLEMENTATION

In this design, the file system component of a VICE cluster server may be implemented on top of an unmodified UNIX system. Each cluster server has mechanisms to map VICE pathnames for which it is a custodian to the local UNIX file system, and to locate the true custodians of other files. VIRTUE is expected to locate custodians using WhereIS and to direct requests to the proper cluster server.

Some files must be replicated throughout VICE. Examples include not only system software and public programs, but also the root portion of the directory hierarchy. Fortunately most such files change seldom. This should make it much easier to update them when necessary without excessive internal overhead.

The secondary custodian mechanism is intended to take care of widely used and/or frequently read files. Secondary custodians service fetch requests independently of the primary custodian in order to avoid centralized bottlenecks. The primary custodian notifies the secondary custodians in the (relatively) infrequent event of an update. When they receive the notification, the secondary custodians discard their copies and fetch new ones at their convenience. Unresolved questions include the possibility of periodic polling by the secondary custodian instead of automatic notification and whether secondary custody is granted statically or dynamically.

It is advisable to locate user files at the "closest" (in terms of access time) cluster server to the user's workstation. Information needed by the VICE operations staff to make and modify such placement can be detected by counting local and remote requests. An excessive number of remote requests should trigger moving the user's files to the requesting cluster. Ultimately the whole process should be automatic.

VICE does not support physical links, since they are meaningless across machines without some notion of universal low-level file identifiers. Cross-machine symbolic links pose no logical problem, but may impose a performance penalty.

There are no cluster-initiated communications with workstations, with the possible exception of information messages.

VICE REFINEMENTS

This section identifies certain unresolved issues, mostly dealing with implementation rather than the VICE interface proper. As such, they can be left to experimentation without major impact on the fundamental interface.

PreFetch

Format

PreFetch(pathname)

Input Parameters

pathname	The string representing the file or directory to be fetched.

Description

PreFetch asks a cluster's file server to prepare to handle further operations on the file directly. Although functional behavior should be identical with that of the custodian, both performance and space accounting may differ between custodians and cluster servers which have prefetched files.

The purpose of PreFetch is to allow the workstation to advise VICE of the need for caching in the local cluster server, for example if the workstation is incapable of caching for itself. Since it is advisory in intent, Prefetch may be ignored by a cluster server.

Errors

Pathname is not a file

Pathname does not exist

Not authorized to read pathname

Unresolved symbolic link

Cfetch

Format

Cfetch(pathname, timestamp, READ|WRITE|NONE) returns (file contents, status)

Input Parameters

pathname	names the file to be locked and fetched
timestamp	is the currently cached copy's modification timestamp

READ|WRITE|NONE identifies the lock to be set on the file

Output

file contents is a fresh copy of the file.

status is the corresponding file status

Description

CFetch sets a lock in the specified mode and performs a Fetch if the given timestamp is less than the modification timestamp of the cluster server's copy. This condenses the commonest sequence we anticipate for locking and fetching files from VICE into a single exchange.

Cfetch is typical of combined operations we may choose to provide to reduce message traffic after we have had an opportunity to implement VICE and VIRTUE and observe their behavior.

Errors

Cluster is not a custodian

Could not obtain lock

Not authorized for lock

Fetch unnecessary

Pathname not a file

Unresolved symbolic link

Cluster caching

The PreFetch operation asks a cluster server to cache a file. It would be possible for the cluster server to do so even if not asked to, by simply behaving as if it had received a PreFetch instead of returning the "I'm not the custodian of this file" error.

This would allow transparent caching on the cluster servers. Those work stations that did not want the caching to take place could issue WhereIS calls to locate true custodians and thus prevent caching in a non-custodian cluster server.

Apart from caching user files, there is a serious question about how to replicate system files and directories among cluster servers. Possibilities include:

1. Replicate top level directories and system files using the standard caching mechanisms. This is perhaps the most economical approach, but it's likely to overload the caching mechanism with conflicting performance requirements.

2. Design the system as if the global files never change. Actually deal with changes during idle periods, using bulk updates (possibly coupled with partial system shutdowns.)

3. Reverse the direction of message flow between the primary and secondary custodians of global files. That is, have the primary custodian notify all secondary custodians in the (infrequent) case of global file updates.

Checkin/Checkout

An authorized workstation or cluster may check out an entire subtree from VICE. Checking out a subtree effectively sets a write lock on its root and every file and (recursively) subdirectory it contains. As with write locks, it is not necessary to check further with the VICE primary custodian on individual accesses to cached files in that subtree. Mastery of the subtree always reverts to the VICE primary custodian in the long term; either by timeout or by explicit checkin of a subtree.

The corresponding VICE calls are:

Checkout(pathname, flag)

Set a write lock on pathname and all the pathnames it dominates. Flag is used to tell the cluster to do the checkout for the workstation, assuming that cluster caching has been provided.

Checkin(pathname)

Cancel a previous Checkout.

UNRESOLVED VICE ISSUES

Locating custodians

There are two proposals for how to identify custodians: in a separate data base or as part of the VICE directory. The particulars of the proposals are:

1. Subtree custodians are identified in a file owned and maintained by some system administrator. Changes to the file are most likely to occur either

because users are added and deleted, or because of attempts to redistribute the load on different cluster servers. These are relatively infrequent operations, so batched updates (for example one a day) are acceptable. This file is replicated at all cluster servers using the same mechanisms used to replicate other system-wide files.

2. The information is made part of the directory and is updated by the system administrator using SetFileStat. It is propagated to cluster servers using the standard directory replication mechanism.

Moving a user's files would involve locking the user's subtree while running a straightforward recursive copy utility program. In both cases this must be coordinated with changing the custodian location information itself.

The VIRTUE file system runs in a workstation and stores files both on the local disk (if any) and remotely in VICE. Its main function is to intercept the workstation operating system's standard file system calls and redirect them transparently to VICE when appropriate.

Since one of our goals is to make it easy to import Unix application programs into VIRTUE, the set of VIRTUE file system calls is modeled after those of Unix.

With one exception, the VIRTUE file name space is identical to that of VICE. The exception is the subtree "/local," which contains files strictly local to the workstation.

VIRTUE SYSTEM CALLS

The following functions are available to application programs on a VIRTUE workstation.

Open

Format

Open(pathname, flags, mode) returns (FD)

Input Parameters

pathname	The string representing the file to be opened.
flags	Specifies the use that will be made of the file. Possible values are: FRDONLY says that the file will be read, default positioning is to the beginning of the file. FWRONLY and FAPPEND are to write the file. FWRONLY positions to the beginning and will overwrite the file, FAPPEND positions to the end and will add to the file. FRDWR allows both reads and writes to the file. FCREATE creates the file if it doesn't already exist. FTRUNCATE truncates the file to size 0 if it already exists.
mode	Specifies the access mode of the file, if it is created.

Output

FD	The file descriptor returned from an Open request. Value is -1 if the open failed.

Description

Open is used to gain access to a file. The flags operand is used to specify what kind of access is desired. FD is returned to allow I/O calls to indicate which file is to be operated upon.

Errors

Pathname invalid

not authorized for operation requested

Close

Format

Close(FD) returns (value)

Input Parameters

FD	The file descriptor returned from an Open request.

Output

value	is 0 if the close succeeded and -1 otherwise.

Description

Close is used to release access to a file. Any outstanding buffers will be flushed and any locks are released.

Errors

Invalid file descriptor

Error on data transfer

Fsync

Format

Fsync(FD) returns (value)

Input Parameters

FD	The file descriptor returned from an Open request.

Output

value	is 0 if fsync succeeded and -1 otherwise.

Description

Fsync is used to force any buffered changes for a file to external storage.

Errors

Invalid file descriptor

Error on data transfer

File not open for writing

Read

Format

Read(FD, buffer address, length) returns (count)

Input Parameters

FD	The file descriptor returned from an Open request.
buffer address	The address of the buffer to receive the data.
length	The number of bytes to transfer.

Output

count	Specifies the number of bytes actually transferred on a successful read. If zero bytes are transferred, end of file has been reached. A value of -1 indicates an error occurred.

Description

Read is used to transfer data from a file to a buffer. The file may be open for reading or reading and writing.

Errors

Invalid file descriptor

Invalid buffer address

Error on data transfer

File not open for reading

Write

Format

Write(FD, buffer address, length) returns (count)

Input Parameters

FD	The file descriptor returned from an Open request.
buffer address	The address of the buffer that contains the data to be written.
length	The length of the data at buffer address.

Output

count	Specifies the number of bytes actually written on a successful write. A value of -1 indicates an error occurred.

Description

This is used to transfer data to a file. The data is contained in the buffer at buffer address. The file must be open for writing or appending. The data may actually be buffered by the system and may not actually be written until a Fsync or Close call is issued.

Errors

Invalid file descriptor

Error on data transfer

File not open for writing or appending

File full

Buffer address invalid

Lseek

Format

Lseek(FD, offset, whence) returns (position)

Input Parameters

FD	The file descriptor returned from an Open request.
offset	The offset used to calculate the new position from which the next read or write starts in the file.
whence	Specifies how the position for the file should be set as follows: if L_SET the next read or write will start at offset. If L_FROMHERE, the next read or write will start at its current position plus the offset. If L_FROMEOF, the next read or write will start at the end of file plus the offset. Note that offset can be negative in the last two cases.

Output

position	Specifies the number of bytes from the start of the file that the next read or write will start at if the call to Lseek is successful. If an error occurred, it is -1.

Description

Lseek is used to retrieve data in a file in a non-sequential manner. It causes the next data transfer command (Read or Write) to start transferring data at the offset into the file indicated by position. Note that it is possible to get the current position in the file by calling Lseek with offset set to 0 and whence set to L_FROMHERE.

Errors

Invalid file descriptor

Offset outside of file

Illegal value for whence

Remove

Format

Remove(pathname)

Input Parameters

pathname	The string representing the file to be removed.

Description

This causes a file to be removed from the name space. The issuer must have write authorization or be the owner in order to issue this call. This call will fail if the file is in use.

Errors

Invalid pathname

File in use

Not authorized to remove file

GetFileStat

Format

GetFileStat(pathname, status, LINK)

Input Parameters

pathname	The string representing the file or directory to be operated upon.
status	The formatted data about the file or the directory.
LINK	This keyword causes information about links to be returned instead of being resolved.

Description

This is used to find out information about a file or a directory. The information for directories will include data about all of the children of this entry. If the LINK keyword is specified any links will be left as is instead of being resolved. For files it will have the information described in section -- Heading id 'meta' unknown --. In addition the information about whether a file is cached locally will be included.

Errors

Invalid pathname

SetFileStat

Format

SetFileStat(pathname, status)

Input Parameters

pathname	The string representing the file or directory to be updated.
status	The status information about the file that is to be changed.

Description

This call is used to change file status, such as protection information.

Errors

Invalid pathname

Invalid status

MkDir

Format

MkDir(pathname, mode) returns (value)

Input Parameters

pathname	The string representing the file directory to be created.
mode	Specifies the access mode for the newly created directory.

Output

value	is 0, if MkDir succeeded in creating the new directory, and -1 otherwise.

Description

This creates a new directory with the name of pathname. All elements of the pathname must be directory files. The issuer must be authorized to create a file in the directory that is the parent of the one specified. The issuer will become the owner of the directory.

Errors

Invalid pathname

Not authorized to create a directory

I/O error

Name already exists

RmDir

Format

RmDir(pathname) returns (value)

Input Parameters

pathname	The string representing the directory to be removed.

Output

value	Indicates if an error occurred. It is 0 if RmDir succeeded, and -1 otherwise.

Description

This removes an empty directory. The user must be authorized to Remove entries in the parent directory.

Errors

Invalid pathname

Not authorized to Remove a directory

Pathname not a directory

Directory not empty

I/O error

Partition

Format

Partition( )

Input Parameters

NONE

Description

This call is used to voluntarily partition yourself from the network. It will allow a user to ensure that there is enough data in local storage to continue to operate.

Errors

Already partitioned

UnPartition

Format

UnPartition( )

Input Parameters

NONE

Description

This call is to reconnect to the network, after a Partition. It will invoke the same data merging that an involuntary Partition would have caused.

Errors

Not partitioned

Network not available

SetCacheMode

Format

SetCacheMode(UNUSABLE | READ-ONLY | READ-WRITE)

Input Parameters

UNUSABLE | READ-ONLY | READ-WRITE Set the mode that any data in the workstation cache will be used during a partition from the network.

Description

This allows the user to specify how the data that is in a workstation cache is to be treated during a partition. UNUSABLE indicates that the cache is to be treated as empty. READ-ONLY indicates that files in the cache can be used, but only read, they cannot be written. READ-WRITE indicates that the files are to be treated as writable. This means that changes made to the files, may have to be merged into the network copies of them when the partition is resolved.

Errors

Not partitioned

Cache empty

VIRTUE IMPLEMENTATION

The VIRTUE file system uses a standard Unix file system in which it stores local ("cached") copies of files obtained from VICE. Read, Write, and Seek requests refer directly to the local copies; only Open and Close requests involve data transfer between VICE and VIRTUE. Such transfers are, however, completely transparent to the application programs.[2]

Caching Files

In handling an Open request, VIRTUE will use the CFetch operation, which locks the file if it is to be written, verifies that its timestamp, and fetches it if not. (No lock is used automatically for reading files.) When the file is closed, VIRTUE will Store the modified copy and unlock it for writes; no special action is needed for reads.

At any instant of time, the cached copy of a file can be one of the following states:

Guaranteed The file has been locked in VICE and an up-to-date copy obtained if necessary. Since VICE will never alter a locked file, it is not necessary to check with VICE before using the copy.

Suspect The file is not guaranteed. This can occur because the file is not open for writing or the lock was lost because of partition.

An second attribute indicates whether the cached copy of a file has been modified since the most recent check for it with VICE:

Modified The file has been changed locally since the last time VIRTUE confirmed it with VICE.

Unmodified The file has not been changed locally.

By locking files before changing them, and storing them back before unlocking them, VIRTUE avoids having Modified Suspect files. This may not be feasible if the workstation can't communicate with the file's custodian (for example, because of network partition.) If so, VIRTUE stores the file back when the workstation is reconnected, using a special directory merging algorithm to deal with conflicting updates.

[2] VICE and VIRTUE may support remote Open of files in a later version of the system -- however, that is outside the scope of this document.

Local files

Certain system files are essential for independent operation of a workstation, initialization, and other circumstances in which the workstation cannot obtain files from VICE.

VIRTUE stores such files in the /local subtree. A special application program updates such files according to directions contained in a special VICE file. This program is run automatically when the user logs on, and may be run any time at a user's request.

The /local subtree could also be used to avoid locking and checking overhead for frequently used system programs and files.

Locating Custodians

How does VIRTUE find files' custodians? The WhereIS call to VICE identifies custodians at the cost of additional messages. In the worst case, each file access from VICE may receive a return indicating the the file is on another cluster. To reduce this overhead, VIRTUE can maintain a table of VICE pathname prefixes to custodians.

The WhereIS operation returns not only the custodian for the specified file, but also the briefest prefix of the specified pathname for which the custodian list is the same. For example, suppose the custodian of "/usr/bovik" is ClusterA. WhereIS("/usr/bovik/doc/thesis.mss") would return the prefix string "/usr/bovik" as well as the ID of ClusterA. By remembering this pairing, the workstation can go directly to ClusterA for all future references to pathnames which start with "/usr/bovik".

Now suppose that for some reason the sub-subtree "/usr/bovik/lists" is in the custody of ClusterB. A reference to "/usr/bovik/lists/pizza" should then be directed to ClusterB rather than ClusterA. This implies that when looking up pathnames to find custodians, the workstation must use the longest matching prefix it finds.

When the custodian of a file changes, requests addressed to the old custodian will fail. The workstation can update its table by simply deleting entries which led to failures and building a new entry in the standard manner.

VIRTUE REFINEMENTS

The minimal design for VIRTUE uses the cache to minimize physical movement of data. However, each use of a cached file copy requires communication with VICE to validate the copy. The following strategies, which may be used independently of each other, can be used to reduce this interaction:

Using /local

As already mentioned, frequently used system files could be stored in the "/local" subtree and updated periodically by an application program run in the workstation. The presence of "/local" in the pathname can be masked using symbolic links.

Don't Care cache state

Marking a cached file "don't care" allows VIRTUE to bypass locking the file and checking the creation date when it is read. Either the user or an automatic aging algorithm periodically purges such a file from the cache. The next use of the file will cause a Fetch, resulting in an updated copy.

Commands to identify the "don't care" files could be included in the logon initialization procedure (the user's profile.)

CheckIn/CheckOut

CheckOut is a possible VICE refinement whereby a workstation sets a write lock on an entire subtree. This could be done automatically at login, again as a profile option. As cache misses occur, the fetched files are marked Guaranteed since they remain locked. Checkout would be part of logout, could be done any time at the user's discretion, or could be forced from another workstation to handle failures and partitions.

Although it's not strictly necessary to write back Modified files immediately, it would be a good policy to do so at least in a background process. This minimizes potential problems in the event of a workstation or network failure, or a forced checkout.

PARTITION

Partition between workstations and cluster servers are detected by failure of an attempt to communicate between them. We assume the transport subsystem notifies both the workstation and the cluster server of a partitioned connection.

When a custodian detects partition it breaks all locks held by the partitioned workstation. If the checkout/checkin mechanism is implemented, this includes automatic checkin of all subtrees checked out by the workstation. Note that it is possible for a workstation to be partitioned from some custodians, while other custodians are still available.

The default assumption made by VIRTUE is that cached copies of files whose custodians are inaccessible may neither be read nor written. A user may explicitly request VIRTUE to make such files read-only, or to make them read-write. Files in "/local," of course, can always be read or written.

To run a workstation completely stand-alone, a user can request VIRTUE to deliberately partition him from VICE. The file cache is handled just as it is in the case of accidental partition. In this case, the user will indicate to VIRTUE when he wishes to terminate the partition.

VIRTUE remembers any Suspect Modified files written during partition and attempts to update VICE using a special directory merging algorithm when the partition ends. The general strategy is for both sides of the partition to journal all updates (including Remove functions) during a partition and to merge the journals when reconnected, thus reconstructing the combined subtree. When updates conflict, one of the two files involved should be renamed and the conflict should be reported to the files' owner for final resolution.

In addition to dealing with partition, the update journaling mechanism can help with archiving and may be used as part of a general "undo". Custodians should always journal, since they can't detect the completely disconnected use of a workstation. It seems reasonable to assume that stale journal entries can be discarded after a few weeks, so the journal shouldn't become unreasonably large.

This section describes a proposed Authorization Server for the VICE system, providing user authorization for the file system and other system services. It assumes the availability of guaranteed secure network connections, protected by encryption, which guarantee to the receiver the identity of the sender.

Authentication, in the context of the file system, is the mechanism by which a file server, as well as other system servers, identify the users making use of their services. It starts when the user workstation VIRTUE software makes a secure connection to the *Login* service of the Authorization Server in the nearest cluster machine, and is ended by a connection to the *Logout* service. In between the VIRTUE software makes many file-access calls, such as *Fetch* and *Store*, each of which provides a service. (Recall that the VIRTUE system must make a separate *ConnectFS* to each file server to which it intends to make access calls; VIRTUE may have to keep track of several conversations to the file servers on different cluster machines.)

The Login procedure will consist of

- Determining the identity of the user making the Login call.
- Assigning to the user an authorization token, to be used as an identification for all further communications with the file servers and other system servers running on various clusters.
- Saving sufficient information in the Authorization Server to allow other servers in that machine, and other cluster machines, to make use of it.

LOGIN SERVICE

The user workstation VIRTUE software will first obtain from the user his computer user name and his password. It will establish a secure network connection to the *Login* service of the nearest cluster machine's Authorization Server, providing the user name, password, and billing account number as arguments.

The Login service will look up the user name in a user authorization file and find the password stored there; if there is no such user, it can immediately return an error. It will check the password, and return an error if it is not correct. It will assign a "job number," unique over all the users currently logged in to that cluster server; it will combine that with the cluster number and a random number, to form an authorization token, and store that, along with the user name and job number, in a table in the Authorization Server, for future reference It will return to the user the authorization token.

It will also perform whatever initialization functions are required by the accounting mechanism, as detailed in the accounting proposal.

CHECK AUTHORIZATION

To obtain services from a cluster server, the VIRTUE software makes calls (not necessarily using high-security connections?) to servers in the cluster machine. Each call should include as part of its arguments the authorization token.

The server (whether this is the file system, the mail system, or anything else is immaterial) will use the internal IPC mechanism to call the *Check Authorization* service of the Authorization Server on that server's cluster machine, and send it the user's purported userid and the authorization token. The Authorization Server will either look up the entry in its own tables, or otherwise obtain confirmation from the Authorization Server which created the token. If it is valid, it returns a confirmation to the requesting server, including any financial information required by the accounting mechanism. The server now knows that the user is who he says he is.

LOGOUT

To disconnect from VICE, the VIRTUE software should call the *Logout* service, providing the authorization token. The Authorization Server will remove the job from its tables, and whatever else is necessary to update Authorization Servers running in other clusters. The workstation should have first used *DisconnectFS* to disconnect from any file servers which it may have been using.

MAINTENANCE

There will have to be a program to maintain the authorization file, which will add and remove users and assign passwords.

ISSUES

This does not provide for any encryption of the messages, nor does it require that guaranteed network links be used for any connections other than the *Login* calls, and perhaps *ConnectFS* calls. If the network is insecure enough, there may be problems with unencrypted data passing through the network. It also may be unacceptable to have the authorization token passed around in the clear, since it is then open to use by unauthorized third parties. If these problems seem severe enough, an encryption scheme based on the Authorization Server may have to be implemented, where the server would issue the user an encryption key over a secure link, and the user would then encrypt all communications with the cluster system with that key. It may also be good to include an expiration time in the token, to keep stale tokens from being passed.

ACCOUNTING AND DISK ACCOUNTING

Accounting services will enable the system to keep a record of the resources used by users, so as to properly keep track of funding for computing. Unlike in conventional computing systems, there will be no need to keep track of user computing cycles provided by the workstations, since those are outside the scope of the network. The central accounting system will only deal with billing for services provided by the network itself and by distributed services used through the network.

The ultimate goal of accounting is the timely production of bills for computer usage; a lesser goal is the notification of the user that he has used all the funds allocated to him. So, first of all there must be a record kept of usage, which can be processed to produce bills; there must be a file containing the user accounts, their allocations, and their balances (a daily balance and a monthly balance); there must be automatically run programs to perform billing processing.

Related to simple money-accounting is the issue of disk usage accounting, and also the issue of limiting disk usage by a quota system. This chapter contains proposals for all three.

ACCOUNTING

There will be an Authorization Server, to which the user workstation software makes a connection to access system services. This server will, in addition to keeping track of authorized users, maintain billing information for each user who is logged into that particular cluster machine's server. Server processes will use IPC facilities to ask the Authorization Server about the identity and finances of a user requesting services, and will send "bills" to the Authorization Server. Users can arrange with the Authorization Server to allow other users to submit bills, for providing "services" known only to the two users.

The model of accounting used here is that an account administrator (someone with real money to spend) asks the central administration to establish an account, adds some users to that account, and sets a limit on the number of dollars each user can spend. While the users expend resources, the accounting system keeps track of the facilities used (for ultimate billing) and of the funds used by each user (to terminate service when the user exceeds his limit). On a monthly basis, the account administrator gets a bill for services provided, in the form of a journal entry being sent to the university accounting office, to transfer money from the buyer's account to the Computation Center's account.

Data bases

The Authorization Server will maintain a file listing for each user, all of the accounts which the user is allowed to use, and some billing balance information for the account: the amount of money allocated to the user, and the amount of money spent by the user.

To produce the raw data, there will be a record kept in the Authorization Server of the cluster machine to which the user is logged in; this "job data base" will contain the user's "job number," his user name, his account number, a list of accounting transactions for the user ("bills"), a total amount spent in this session, and a remaining balance. As billable services are provided, the billing information will be updated.

At Logout, the billing information will be written to an accounting transaction file, and the balances updated. When the Authorization Server wishes to spill its transaction file, it will copy it across the network to a common directory, to be processed later.

Login

To the *Login* service of the Authorization Server, along with any other information required by the authorization mechanism, the user will provide a billing account number, which will be checked against a valid user name-account list. If there is no money left for that user, he is not allowed to log in. The user name, account number, and remaining allocation are recorded in the Authorization Server, for that session, and the billing data for that session is initialized to be empty.

Also at this point, the user can be warned if his remaining allocation is less than a certain amount, to let him know he is running out of money.

Server processes

As the user requests services from servers, the servers will make connections to the *Check Authorization* service of the Authorization Server to determine if the user is who he says he is; the return message will say if he is, and if so will also return the user's remaining allocation, so the service-provider can judge whether the user has enough money to complete the operation. The server, if it then wishes, will perform the operation, and will then send a message to Authorization's *Bill User* service to bill the user, providing the dollar amount to be billed, and a commodity code for the service provided (e.g., code 6 might be "mail service").

The accounting system will make a record of the service, updating the billing information blocks in the job context area, combining multiple instances of a particular service into one to save space. It will also deduct the amount billed from the user's remaining allocation.

To reduce overhead, the file server may be able to maintain a small number of counters, perhaps of requests made, or files transferred, or blocks transferred.

Logout

The Authorization Server's *Logout* service will process the billing information for the job, writing billing entries into a file specific to that cluster machine (perhaps a /local file); if there is any special handling to be done for file server information, *Logout* will do that also. It will also update the permanent data base with the new remaining-funds count.

Daily billing

The Authorization Server should occasionally spill the contents of its ever-growing transaction file, when it reaches a certain size or every few hours. It should rename it locally and then copy it across the network to some common directory, where it can be saved until needed for monthly billing. Since user balances are recomputed on the fly, there is no need for a separate daily billing program; there is likewise no need for a central program to distribute updated accounting balance files, since they are owned and maintained by individual cluster machines.

Monthly billing

Once a month, one some one machine, a program should go through the accounting data base and produce bills for all accounts with usage that month. The usual practice has been to generate paper bills to send to the account administrators, listing all the commodities for each user (tempered by how much detail each administrator wants), and in parallel to generate journal entry records to deliver to the University Accounting office, to transfer real money from the using department's books to the Computation Center's books.

Offline billing

There will be some need for the billing of services to a user when the user is not logged in to any cluster server; examples which come to mind immediately are disk storage billing and printing. A *Bill Offline* service to the Authorization Server must be provided which will do the entire process of billing without going through the *Login - Bill - Logout* process; needless to say, this should only be usable by "system" servers.

Maintenance

There must be a program to add and remove user/account entries in the data base, to update the allocations, and to adjust or reset the balances.

User-to-User transactions

To provide an organized means for users to provide services for one another, without the explicit intervention of the central administration, we will provide a means for users to obtain "electronic money orders" from the accounting system. These will be encrypted so that only the accounting system which printed it will be able to see what it has on it, and thereby assuring that the users among whom it is passed do not modify it.

A money order is a string of bytes, partly in the clear and partly encrypted; the information on both parts is the same: the dollar amount, the buyer's user name, the cluster number of the issuing Authorization Server, and a serial number.

The user wishing to get an unofficial service (the "buyer") obtains from the unofficial service provider (the "seller") the expected price for the service (whether by word of mouth or by making a network connection is immaterial).

The buyer makes a connection to the *Money Order Purchase* service of the Authorization Server to which he has logged in, and asks for a money order for a specific dollar amount, presumably enough for the service. The Authorization Server deducts the money from the user's account, records in its own temporary records the issuance of the money order, and returns to the buyer the money order.

The buyer sends the money order to the seller, who performs the service and determines the final price (the first price may have been only an estimate, an upper limit); the seller connects to the *Money Order Redemption* service of the Authorization Server which issued the money order, and sends it the money order itself and a final dollar amount.

The Authorization Server decrypts the money order, confirms that it is an outstanding money order (according to the serial number), cancels the money order (wiping out the temporary record), adds the requested amount to the seller's account, and then returns any unused portion to the seller by issuing a brand new money order.

The seller, satisfied that he has been paid, returns the answer to the buyer (or otherwise returns what the buyer wanted), along with the money order obtained as change. (Or, the seller may wait until the money order has cleared before actually performing the service, if it is suspicious enough; this will only work when it knows the exact price.) The buyer can then redeem the money order, or save it for any other purchases he may make in the same session.

There are possible problems with this system, most notably its dependence on the honesty of the seller with regard to actually performing the service requested, and for setting a fair price. There do not seem to be automatic mechanisms available to force users to be good with each other; if there are serious abuses, it may be necessary to make the system refuse to honor money orders from certain disreputable sellers.

DISK ACCOUNTING

To bill for disk usage, there should be a program which runs automatically every day and generates accounting records for the disk storage used by the owner of each file. For all user directories (those starting with "/usr") it is sufficient to examine each file in the directory tree (including the directory directory files themselves) and to bill its storage to the owning user. Each file should have as one of its attributes the computer account number the user was using when he created the file; this information is available from the Authorization server. In the case of files created in a user's directory by another user, the account number will have to be checked against those used by the directory owner; the mechanism used in the past has been to give all files with invalid account numbers some valid account.

The information provided in the accounting record will be

- The user's user name and account number.
- The "disk storage" commodity code.
- The number of blocks billed for.
- The charge in cents.

This will have to use the *Bill Offline* accounting service, since the user is not logged on at the time.

DISK QUOTAS

Although it might be possible to keep disk usage within manageable levels simply with economic influences (i.e., setting the billing rates so that users with too much storage will get unpleasantly high bills), it seems safer to provide a system to limit user disk allocation on a more forceful basis.

For the first implementation, we propose that users be assigned two disk quotas, a logged-in quota and a logged-out quota; the administrative quota assignments should always provide a logged-in quota not smaller than the logged-out quota. A user has a quota on each cluster server on which it he is allowed to store files; ultimately, there should be network-wide quotas. All file creations done in a user's directory cause the storage to be charged against his logged-in quota (if he is logged in, otherwise his logged-out quota); if he exceeds that limit, the file creation fails. Whenever a user logs out of a cluster server, his current usage is checked against his logged-out quota; if he is over the limit, the logout fails. Since it is possible for a user to be logged in twice to the same cluster server, there will have to be context information kept for each active user separate from the cluster server "job" information.

As with all known quota systems, there are problems with this one. If a user runs out of disk while running a program, he is out of luck; there is no automatic forgiveness beyond the logged-in quota. The need to have quotas on many cluster machines is unpleasant, but may be all we can do in the first implementation of the file server. The problem of enforcing logged-out quotas is not really solved by just refusing the *logout*, unless there are penalties for not logging out of cluster servers. The common problem of lack of administrative restraint (i.e., over selling the disk by giving out large quotas in excess of the physical disk size) is not eliminated by this scheme (as it is, for instance, in the VM/370 system).

There will have to be code in the file system itself to enforce the quota policies. There will also have to be a maintenance program to add, delete, and remove entries in a quota file. If over-allocation is allowed (as it probably will be) there will have to be a program available to forcibly migrate user's files to off line storage; if this mechanism exists, there should also be a way for users to voluntarily move their files to this off line data base as well. This facility might or might not be related to the file backup system.

PROTECTION

File protection is clearly a requirement of VICE. In a large user community, however, it is only one instance of a general class of protection problems. VICE will be composed of many relatively independent subsystems, each providing different services. What we propose here is a general mechanism for them to control access to the objects they are responsible for. The file system is, of course, a very important such a subsystem.

The principles guiding this design are that it should:

- be easy to administer.
- allow revocation of privileges after they have been granted.
- be richer and more flexible than the Unix file protection scheme.
- allow workstations to simulate Unix protection on cached copies of VICE files.

Note that we do not consider the problem of low-level authentication here. In other words, it is assumed that there is some orthogonal mechanism for determining who the requester is, and for ensuring that communications with the requester are not public.

Note: For expository reasons, this chapter differs in style from the rest of the document. The other chapters first present a simple design and then explain the refinements possible. Here we first present the complete conceptual model and then explain, in Section "Implementation Phases" on page 66, how it may be implemented as a succession of refinements starting from a simple base.

INTRODUCTION

Associated with each protected object in VICE is an *Access List*, which is a function from a *Protection Domain* to a set of rights. For every member, *m*, of the protection domain, such an access list answers the question "What rights does *m* possess on this object?" The protection domain consists of two kinds of entities: *Users* and *Groups*.

A user represents an accountable entity in the system: one who can be charged for resources and who can be held responsible for actions performed on his behalf. Typically, a user is a human user of the VICE/VIRTUE systems.

A group is a set of users and/or a set of other groups. The implication of this recursive definition is illustrated by Figure 1 on page 59. The user *U* is a direct member of group *A*, and *A* is a direct member of groups *C* and *D*. The *"is a*

member of" relation is transitive, and hence *U* is an (indirect) member of groups *C* and *D*. The rights that *U* possesses on an object is given by the union of the rights that are specified for *U*, *A*, *C*, and *D* on the access list for the object.
More generally, the transitive closure of the "is a member of" relation yields the set of all groups that a user is a direct or indirect member of. This set, together with the user himself, is referred to as the *Current Protection Subdomain* of the user. The rights that a user has on an object is the union of the rights specified for his current protection subdomain on the object.

Conversely, the transitive closure of the *"has as a member"* relation yields the set of all members (direct and indirect) of a group. In Figure 1 on page 59, for example, the membership (direct and indirect) of *C* is *A*, *U*, *V*, and *W*.

Before protected objects may be used, the requester must *Connect* to the VICE subsystem that deals with those objects. Connecting establishes two bindings for the duration of the connect session: the identity of the requester, and his current protection subdomain. Only users may connect to VICE: groups may not. In most cases, processes in VIRTUE will perform the necessary connections to VICE subsystems, thus avoiding human intervention.

While the protection domain is universal in VICE, the interpretation of rights is object-specific. Each VICE subsystem that implements a type of protected object has to provide the following:

- A correspondence between operations on objects and rights in access lists.
- An access list for each instance of an object, and primitives to manipulate it.
- Enforcement of the protection specified in the access lists.

At the present time, we are concerned with only two VICE subsystems: the subsystem responsible for managing the protection domain, and the file system. Sections "Protection Domain Management" and "File System Protection" on page 64 discuss protection in the context of each of these subsystems. As other object types are implemented in VICE, the requirements mentioned above will have to be addressed for each of them.

PROTECTION DOMAIN MANAGEMENT

The protection domain management subsystem handles the creation and deletion of users and groups, and the modification of group membership. The protected objects manipulated by this subsystem are users and groups.

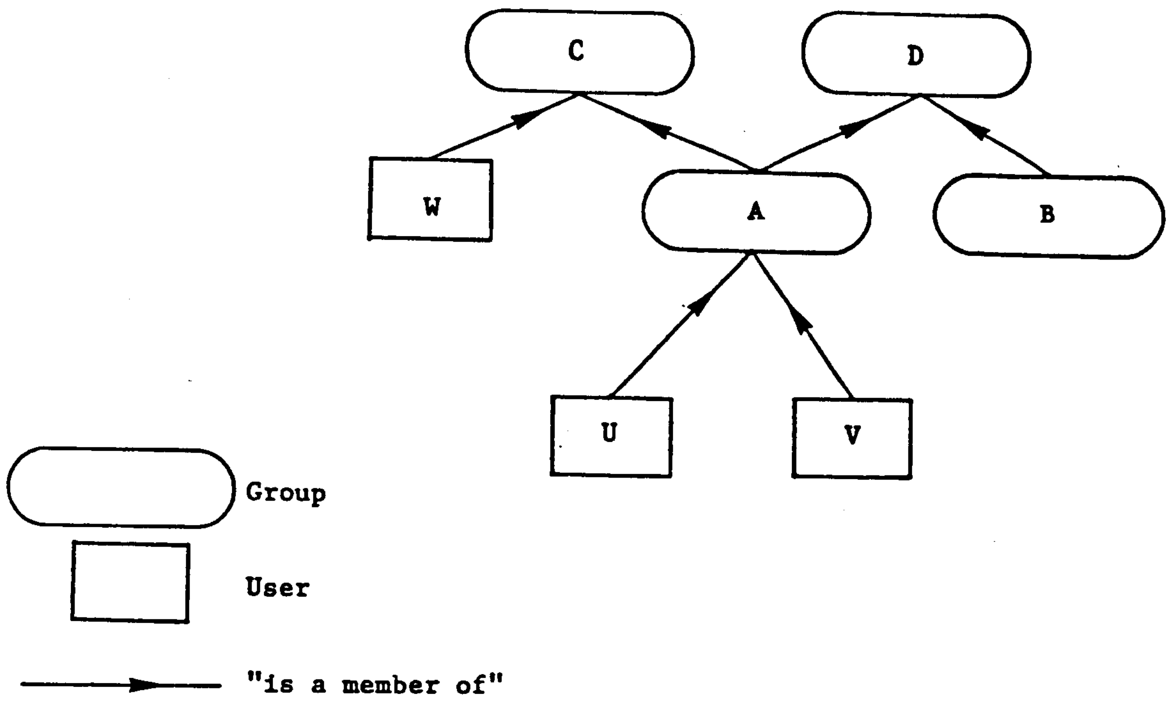

Figure 1. Users and Groups

Each group has one user who owns it, possesses all rights on it and who is responsible for its administration. Initially, the user who requests the creation of a group is its owner. Ownership can be transferred to any other user. An owner can delegate responsibility for administering a group by allowing group modification privileges to other members of the protection domain.

A distinguished user called "System" corresponds to the system administrator and possesses certain unique privileges. While all users can create groups, only System can create other users. System is the only user that VICE starts out with.

When a user is created, a *Root Group* is also created for him. A user's root group is the default point of attachment of all groups created by that user. By convention, the name of this root group is of the form "*Username*Groups". For

---

There is a school of thought that believes that joint ownership should be possible. For example, the set of users and groups who have the ability to modify a group could be its owners. Single ownership does, however, simplify issues regarding accountability.

Does the new owner have to acknowledge his willingness to bear this responsibility? This question is one of a set of related questions. Do you have to give permission to be put on an access list? To be included as part of a group? The principle adopted here is that such acknowledgement is unnecessary. If it were necessary, VICE would have to provide some kind of handshaking mechanism, whereby the donor and the recipient of a privilege could both confirm the acceptance of their roles.

example, the root group of user "Bovik" would be called "BovikGroups". Being a user, System has a root group called "SystemGroups"; this essentially corresponds to the notion of "World" in systems like Unix.

Every group that is created has, for naming purposes, a parent group. The name of a group is thus similar to a Unix file name: if SystemGroups is the parent of A, and A the parent of B, then a group C created with B as its parent will have the name SystemGroups.A.B.C. By convention, the "SystemGroups." prefix can be omitted, and hence the above group can be referenced as A.B.C. Note that parenthood implies membership. In the example just cited, all members of A.B.C are members of A.B, and all the members of A.B are members of A. A user may create a group without an explicit parent group: in that case, his root group is its parent. Figure 2 on page 61 illustrates group naming for a small organization. The purpose of this naming scheme, is to prevent the uncontrolled generation of groups which are of limited general interest, but whose names have widespread mnemonic significance.

In understanding this scheme, it may be helpful to view System as the most selective member of the protection domain, and SystemGroups as the least selective. Similarly, of the elements of the protection domain associated with a user X, the most selective is X, and the least selective is XGroups.

PROTECTION SUBSYSTEM RIGHTS

Associated with each group is an access list, specifying who may examine the group or modify it. As mentioned before, only System can create and delete users. The rights associated with a user are:

ListMemberShip Allows you to list the groups that this user is a direct member of.

Groups, however, may be manipulated by users if they possess adequate rights. The rights associated with a group are:

ListMembers Allows you to find out who the direct members of the group are.

ListMembership This allows you to find out which groups this group is a direct member of.

ModifyMembers This allows you to add members to or delete members from a group. Ownership automatically bestows this right.

The reason for this restriction is ease of user understanding, rather than implementation difficulty. It would be trivial to implement a system in which parenthood and membership were orthogonal concepts, and we may therefore relax the restriction in the light of experience.

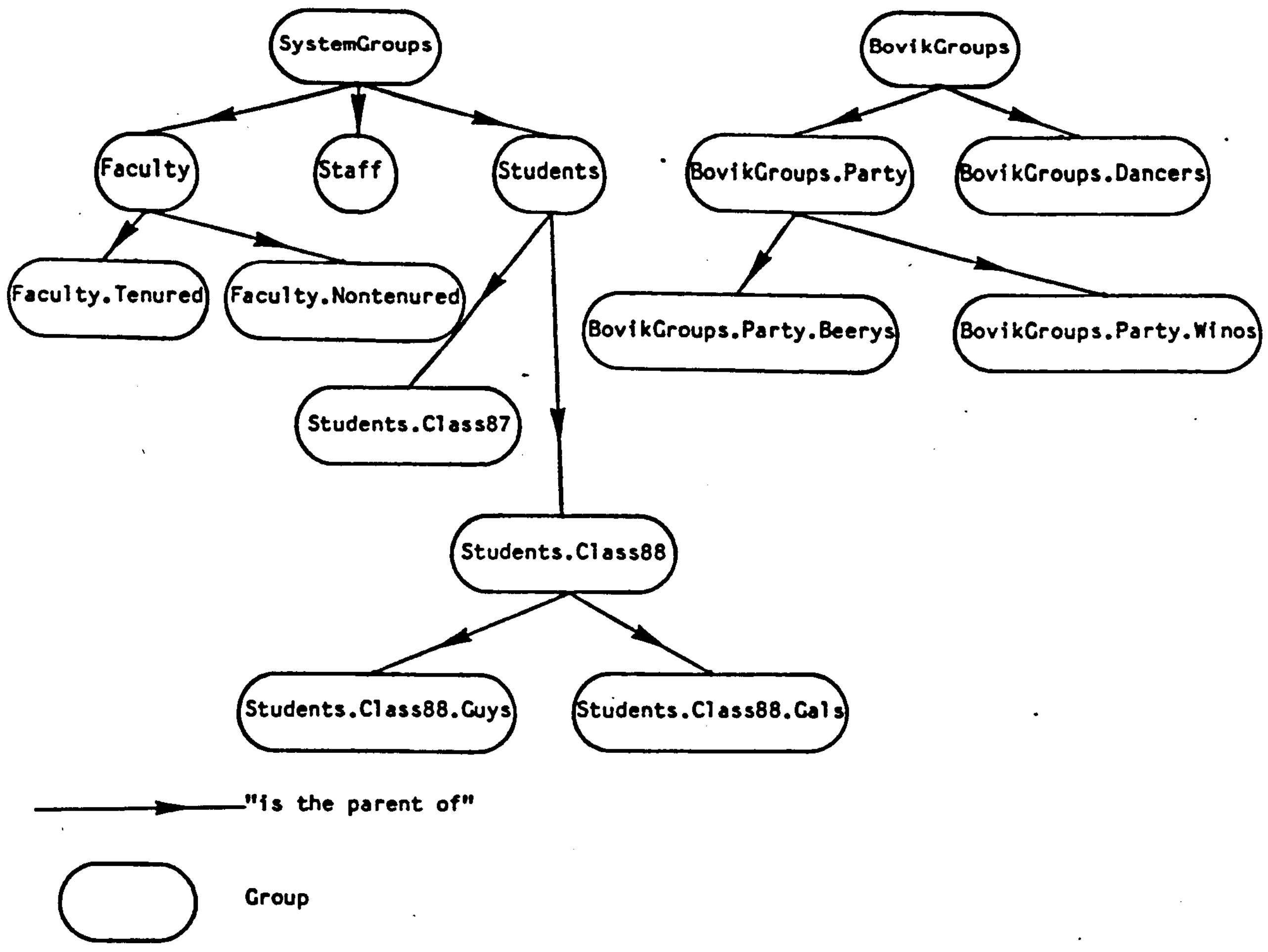

Figure 2. Naming Groups

PROTECTION SUBSYSTEM CALLS

The following calls are supported:

ConnectPS (user, Authentication Information) Establish one's identity.

DisconnectPS() Obvious

CreateUser (username) Create a new user, and make him a member of SystemGroups. Also create his root group and make him a member of it. Only System may make this call.

RemoveUser (username) Purges the user and his root group from the protection domain. Only System may make this call.

CreateGroup (groupname, parent group) Create a new group and make it a member of the parent group. The name of the group will be *parent group.groupname*. The caller must have *ModifyMembership* rights on *parent group*. If *parent group* is omitted, the name will be *user*Groups.*groupname*. The calling user is the owner of the group.

RemoveGroup (group) Delete the specified group. All membership links associated with this group are lost. Only the owner of a group can delete it.

AddToGroup (group1, group2) Makes *group2* a direct member of *group1*. The caller must possess *ModifyMembership* rights on *group1*.

RemoveFromGroup (group1, group2) Make *group2* no longer a direct member of *group1*. The caller must possess *ModifyMembership* rights on *group1*.

GetDirectMembers(group) Return the list of direct members of *group*. The caller must possess *ListMembers* rights on *group*.

GetDirectMembership(group or user) Return the list of groups of which the specified *group* or *user* is a direct member. The caller must possess *ListMembership* rights on it.

GetSubdomain(user) Return the current protection subdomain of *user*. The caller must possess *ListMembership* rights on *user*. Typically all VICE subsystems will possess this right on all users, and will make this call when a user tries to connect to them.

ChangeProtection(group1, group2 or user, rights list) In the access list entry for *group1*, replace the entry for *group2* or *user* by *rights list*. Only the owner can modify the protection on a group. If the caller is System, *group1* may be a user.

In a simple, non-paranoid implementation of VICE, SystemGroups would have *ListMembers* and *ListMembership* rights on all groups (and users), and only owners would possess *ModifyMembers* rights on groups.

SKETCH OF PROTECTION SUBSYSTEM IMPLEMENTATION

For ease of representation, groups and users should be represented as fixed length integers, called group and user IDs. There are mapping tables to convert from names to IDs. IDs are for the internal use of VICE, and are not visible outside it.

The current protection subdomain and the access lists are sorted according to IDs. So finding the available rights on a file is just a matter of running down two sorted lists and ORing the rights masks together.

The relations implicit in the protection domain are captured by three tables:

TableA: "Is a direct member of" For each user and group, this table yields a list of groups of which it is a direct member.

TableB: "Is a member of" For each user and group, this table yields the current protection subdomain. It essentially contains the transitive closure of each entry in *TableA*

TableC: "Has as direct member" For each group, this table yields the list of users or groups who are its direct members.

The most common request to the protection subsystem is likely to be the *GetSubdomain*, from other VICE subsystems, in response to a user connect request. This merely involves a lookup of TableB.

When a change is made to a group, the relevant entries in tables A and C are changed. *TableB* now has to be recomputed. In the most general case finding the transitive closure of *TableA* can be an expensive operation. There are a number of approaches to address this issue:

- Restrict groups to have only users. This makes the transitive closure trivial, but severely reduces flexibility.

- Allow groups to have other groups as members, but limit the depth of nesting. This limits the transitive closure to an $O(N^2)$ algorithm rather than $O(N^3)$.

- The direct membership matrix is likely to be sparse. Develop a transitive closure algorithm which exploits this property to run efficiently in the average case.

How is change propagated in the system? There are two views on this, and no consensus has been reached on which of these is preferable:

Slow update The tables A, B, and C, are replicated at each VICE node and are read-only most of the time. Requests for change are sent to one node which batches them, computes the new tables, and atomically updates them at each node. The slow propagation mechanism is similar to that used by the file system to maintain its replicated database of file locations. The frequency of updates is a system parameter, and is typically about once a day.

Immediate update The changes take place as they are made: there is no batching of requests. The details of such a mechanism remain to be worked out. It is not clear that the full-fledged protection scheme can support immediate update efficiently enough to make this possible.

Undetermined: Should there be a mechanism to subtract rights? Example, entries in an access list whose rights masks are NANDed rather than ORed during a rights check? This allows rapid, selective revocation. It also provides a means of overcoming the parenthood-membership coupling mentioned earlier.

FILE SYSTEM PROTECTION

In the file system, the protected objects are files and directories. We feel that it is perfectly reasonable to require that all files in a directory share the same level of protection. Therefore protection can be specified only at the granularity of a directory, and not of an individual file. It is, of course, possible to have directories with exactly one file in them, to handle pathological cases. The set of rights on a directory are:

LookupFiles Allows GetFileStat() to be performed on files in the directory. Also allows the access list for the directory to be examined.

CreateFiles Enter files into this directory. This is distinct from *WriteFiles* in order to support mail and other similar functions.

ReadFiles Examine the contents of a file in this directory. In the current file system design, this effectively means that Fetches may be done on files in the directory.

WriteFiles Replace an existing file. In the current file system design, this is equivalent to allowing Stores on files in the directory. This right also allows files to be removed from the directory, and allows the access list for the directory to be altered.

The primitives provided to manipulate an access list are:

CreateEntry(pathname, group or user, initial rights) Creates a new entry for *group* or *user* in the access list for *pathname* and assigns *initial rights* to it.

RemoveEntry(pathname, group or user) Deletes the *group* or *user* from the access list of *pathname*.

ModifyEntry(pathname, group or user, new rights) The entry for *group* or *user* at *pathname* is changed to *new rights*.

ReadAccessList(pathname) Returns the access list for the directory at *pathname*. The requester must possess *LookupFiles* rights on the directory.

When accessing a file, the protection check is made only on the immediate parent of the file: a requester does not have to possess any rights on the intermediate directories of a pathname.

The purpose of this restriction is to limit the number of access lists that a typical user has to deal with. If experience demonstrates that this is overly restrictive, we can support per-file access lists. Note that the inverse is likely to be painful, because application programs may have come to depend on the assumption that there are per-file access lists.

Unless otherwise specified, the requester must possess *WriteFiles* rights on the directory in question.

SIMULATING UNIX PROTECTION

Every file in Unix has an *owner* and a *group* associated with it. The Unix protection mechanism is based on a per-file access list that is exactly 3 elements long: one element corresponding to the owner, one to the file's group, and one to all other users in the system (called the *World* group). At any instant of time, a logged-in Unix user is associated with precisely one group. When accessing a file, the rights that this user has on the file is determined as follows:

- If the user is the owner of the file he obtains the owner rights on the file.
- Else, if the user's group coincides with the file's group, he obtains the group rights on the file.
- Otherwise he obtains the World rights on the file.

The rights associated with a file are: *Read*, *Write*, and *Execute* Read rights are needed on a directory to enumerate the files in it, while Write rights are needed to enter files into it.

How can VIRTUE support Unix-style protection on VICE files? The general principle applicable here is that there is more information available in the VICE protection mechanism than there is in Unix: it is therefore a matter of discarding irrelevant information when simulating Unix protection.

VIRTUE can maintain, as part of a logged-in user's state, the notion of a "current user" and "current group". On all file system calls involving a protection check, VIRTUE examines the access list in the status portion of a Fetched VICE file. The rights specified for the "current owner", "current group", or "world" are obtained from this access list and used to determine the validity of the call as in Unix.

Note that the distinction between execute and read rights is meaningless in a distributed system with untrustworthy nodes. Therefore, VIRTUE may have to treat read and execute rights as synonymous. Alternatively, some bits in a VICE rights mask may be explicitly reserved for VIRTUE's use. Since protection in VICE is only at the granularity of a directory, calls to VIRTUE specifying the protection on individual files will be disallowed.

In a simplified version of this scheme, VIRTUE would not try to restrict the rights that VICE gives to a user. The Unix protection simulation would then only apply to setting the protection of directories.

One feature used in some Unix system programs is a "setuid" call. The purpose of this feature is rights amplification: for the duration of the execution of the program, a user acquires the rights of the owner of that program. While VIRTUE workstations can be sure that the owner of a file is indeed the person whom VICE specifies, the inverse is not true. Consequently the "setuid" concept cannot be supported by VICE. However, VIRTUE may support this concept with respect to its cached copies of VICE files: in that case VIRTUE file systems will have to be VICE users in their own right, in contrast to their human users.

IMPLEMENTATION PHASES

The purpose of this section is to suggest an evolutionary path for the implementation of protection in VICE, starting from a base that is close to Unix, and culminating in a full-fledged implementation of the proposed protection mechanism. Such an evolution serves a number of purposes:

- It permits a swift initial implementation, adequate to fulfill the needs of the ITC users. If all else fails, this is adequate to provide a low-aspiration protection scheme for deployment.
- It allows the implementers to gain practical experience with the proposed ideas, and to allow timely correction of conceptual errors.
- It provides natural milestones at which the implementers may divert attention to higher-priority issues that may arise during the course of the file system implementation.

The proposed phases are as follows:

1. Put in place the access list mechanism in the file system and the routines to examine and modify them. Once these are in place, modifications to the protection mechanism will be transparent to the rest of the file system. Further, they provide the hooks for the enhancements that will follow.

   In this implementation the level of functionality is that of Unix. There are 3 entries in each access list: one for the owner, one for an arbitrary group, and one for World. Only the owner can modify the access list. Groups consist only of users, not of other groups. Everyone can examine groups for their membership. Only System may create or modify groups. Information on groups is maintained in a very simple data structure (probably a single file). Replication of this data structure is done by manual transfer to all VICE nodes.

2. Relax the restriction on length of access lists. Note that there is always likely to be an implementation limit on the length of such lists, even in the final system. Introduce the notion of owners of groups and group management rights, and implement all the protection domain management routines in a very simple manner. Groups can still consist only of users. Allow authorized users, other than the owner, to modify access lists. Incorporate a simple automatic nightly update mechanism for reflecting group membership changes, and the introduction of new users.

3. Examine the use of groups as distribution lists for mail. Make modifications as necessary to support this function. Other VICE subsystems may also use the protection mechanism now.

4. Allow groups to contain other groups. Severely restrict the branching and depth of nesting of each node of the protection domain tree. In other words, a group can only consist of a small number of other groups, and the overall depth is also limited. Gain experience, and tune the protection management routines.

5. Investigate rapid update mechanisms for propagating changes to group membership.

6. Relax the restrictions of branching and nesting: once again, there are always likely to be some upper bounds on both these parameters. Evaluate the performance of different transitive closure algorithms and observe their performance in practice. If this relaxation conflicts with the need for rapid update, use the usage experience gained so far to make a reasonable design compromise.

As the above sequence demonstrates, the proposed protection mechanism is not an all-or-nothing scheme. There is a progression of implementation effort, from the simple to the complex, yielding increasingly functional subsets of the full-fledged design. If at any of the above milestones it is felt that the complexity of implementation is overwhelming, further enhancements can be forsaken and the system deployed at the then-current level of functionality. In any case, the level of functionality will be equal to or higher than that provided by Unix.

PAGING AND DISK SERVERS

Although we hope to support diskless workstations, we lack adequate workload, performance, and cost data to make informed design decisions about them at the present time. This situation will change as we get a pilot system going and measure its performance and the user's behavior.

The initial system will be developed on diskless SUN workstations, using SUN's remote disk servers. These servers simply partition physical disks into multiple virtual disks, relocating seek addresses appropriately. Requests to the disk servers use straightforward physical addressing. It should be an easy matter to port or re-implement such disk server code on whatever machines we use for cluster servers.

A more ambitious approach would be to support remote open in cluster servers. We would copy the entire VIRTUE file management package in the cluster server, leaving behind only simple interfacing routines which convert local file system procedure calls (the VIRTUE interface) into remote procedure calls. Instead of caching files on disk, the workstation code would cache pages or records in memory.

Within the cluster server, certain operations (notably Fetch and Store) would become trivial. Data would actually move as a result of the remote Read and Write operations.

Paging could continue to use disk servers, or could be converted to read and write a large file. This may have unacceptable performance implications, but might provide an opportunity to do "lazy fetch" and memory-mapped files by uniting the VIRTUE page maps and buffer pool.

ARCHIVING AND BACKUP

Backup protects against loss of files, whether because of device failure (for example a head crash), program bugs, or operator errors. *Archiving* generally refers to long-term storage of files on some very cheap medium such as magnetic tape, and provides some backup capability as well as protecting users against inadvertently deleting their own files.

Ideally the probability of losing a file should be so small that users simply don't think about it. It seems to be generally agreed that making one backup copy is sufficient, provided that the copy is stored on a different device from the original. If immediate availability of the backup copy is important, it is common to provide a redundant access path as well. Very important backup data is sometimes located in a distant building to protect against fire and acts of God.

The commonest backup and archive mechanisms are:

- *Periodic dumps* of entire disks to tape or another disk. In the event of a disk failure, a replacement disk is obtained (from a pool of spares) and reloaded from the backup copy. Files created between the last dump and the failure are lost.

- *Mirrored disks*, which update the backup disk continuously by writing every record to both disks at once. This doubles the number of disks required. As with periodic dumps, recovery usually requires a short outage to copy the remaining good disk to a new mirror copy before resuming operation.

- *Incremental backup*, which writes changed files to a tape journal. Changes may be detected either continuously or by periodic polling of the on-line disks. Recovery from failure involves reading the entire set of journal tapes involving the failed disk. To limit the number of tapes which must be scanned, incremental systems usually employ either a periodic dump or a journal merging mechanism.

Archive systems, which typically never delete old files, generally use journals together with an on-line index so that a file to be retrieved can be located without scanning more than one tape reel. They may also stage archived files through various levels of storage, such as relatively large but slow disks, mass storage devices, optical media, and operator-mountable tapes.

It would be premature to select particular backup or archive mechanisms for the ultimate VICE file system at this point. The projected numbers are too large and workload characteristics such as the rate of creation of new files are simply unknown. What is needed is an adequate mechanism for the 200-user test system projected for 1984. The 1984 system should provide some of the workload information we need to design a viable system for thousands of users.

In order to provide a satisfactory system for the short term, and recognizing the likelihood that it will be replaced later, we plan to back up files in a special backup server machine which looks to the rest of VICE much like a secondary custodian. The backup server will Fetch all changed files from their primary custodians and store them on large disks and/or magnetic tape. In the event of a failure, the archive server can take over the role of the primary custodian, restoring the saved files at the request of recovery programs run by the recovering custodian.

Although the main goal of the initial system is backup, it can evolve into an archival server by writing old files to tape and by including creation dates in file names. Using tape is conceptually straightforward, though it requires careful management. Adding dates to file names is somewhat trickier, and will require some care resolving questions such as the implications of renaming a directory. In principle it means that delete operations are ignored and dates are used to discriminate between identically named versions of files. Undated references use the most recent version of a file. To keep things simple, only files will be dated; a simple extension to pathname syntax should then suffice to allow including a date.

MAIL, BULLETIN BOARDS, AND PRINTING

Mail, bulletin boards, and printing are not basic file system functions, but are proposed here as examples of applications which would use the file server, and which would require servers running in the cluster machines. This proposal includes design decisions which are less then those which would be made in the perfect world, but which have the advantage of being within reach in the coming year. In addition, some fundamental decisions were made in the ways they were because of the low-risk nature of the file system.

This does not propose a design for any gateways to other networks which might be available; if gateways exist, and provide servers for these services, the workstation software should also have programs to access them.

MAIL

The primary requirements for a mail system are:

- Any user must be able to put mail in any other user's mailbox.
- A user must be able to read the mail in his mailbox.
- A user must be able to take things out of his mailbox, whether to throw them away or to move them to a local "filing cabinet." It might be nice if the user could modify things in his mailbox and put them back there, making the mailbox the file cabinet; but whether this is actually required depends on the workstation mail system.

Storage

In time-sharing systems with which we have familiarity, incoming mail has been stored in different ways:

- As separate files in a queue owned by the system, from which the user must remove messages.
- As bytes appended to the user's mailbox file, which is one of the many files in the user's directory.

In the present situation, both of these approaches have unpleasant drawbacks. In the first case, the messages which have not been received are stored in a system directory, where disk quotas, accounting, and maintenance are problems. In the second case, there are no "append" primitives in the initially proposed file system, requiring that the entire mailbox be copied into some system (a workstation or a cluster server), be appended to, and copied back to its proper place; this is more network traffic than seems reasonable, and more risk of damaged files.

Instead, as an interim solution, we propose that each user have in his directory a "mailbox" sub-directory, implemented like any other Unix directory within the file system; for now, let its name be "/usr/bovik/mbox" for user "bovik". Note that, like all files owned by Bovik, this is a file under the custody of some particular cluster server, which any cluster server process can identify using *WhereIs*. The files in this directory are real files, also like any other Unix files in the file server; each file contains one mail message, in a format to be determined (perhaps there will be control structure besides the message text?), and with names to be determined (perhaps encoded time stamps, perhaps just sequence numbers). The protection of the mailbox directory can be set up so that its files can be read and written by the owner, and be invisible to other users. (It might be feasible to have the directory protection set so that any user can create new files in it, without going through the mail server. This is not the mechanism proposed here, since a mail server seems to be a required element of the system.)

Usage

To send mail to a user, the sending workstation software must establish a connection to the mail server running on the cluster machine which is the custodian for the recipient's mailbox (this information can be obtained by a WhereIs on "/usr/bovik/mbox") and pass it the mail message and the recipient user name; if the users are on different clusters, the sender will have to log in to the receiver's cluster server. The mail server will ask the local Authorization Server whether the user is who he says he is, and then will create a new file in the recipient's mailbox directory and store the message there, adding a line to give the recipient a guaranteed identification of the sender. It will also bill the sender for sending the message, perhaps with one rate for the message and another depending on its size.

Once it is in the recipient's mailbox directory, the recipient's workstation software can do what it wants with the files. There must be a mail receiving program to look at messages, delete them, forward them to other users, copy them to other files, etc. There can be a demon to regularly examine the directory for new messages, and to invoke the mail program to present them to the user.

BULLETIN BOARDS

Bulletin boards are very similar to mailboxes, except that they must be read by any user (or any user in the proper access list, if such things are implemented), and that they are not "owned" by a person in the same sense as a mailbox is owned, and thus will not be routinely read and trimmed by the owner. Even more than mailboxes, they will become very large, and unsuitable for transfer around the network. Some will be "owned" by the system administration, and so will be stored in "system" directories and maintained, if at all, by the system administration. Others will be privately owned, stored in the disk space of the owner and maintained by him.

Storage

To store bulletin boards, we propose a similar scheme to the mail system, where the bulletin board is a directory and the messages are files in the directory; however, the protection of the directory and files can allow all users to read them (in the case of public bulletin boards; private bulletin boards will require more substantial protection primitives). The files stored should contain some control information along with the message text, in particular the expiration date for the message.

Usage

Once again, messages are posted by sending them to a bulletin board server, providing the name of the bulletin board, the expiration date, and the text of the message to be posted; the server will store the message in a new file in the directory (a syntax for specifying a private bulletin board will have to be developed).

To read a public bulletin board, a workstation program need only look at the directory and present to the user the files it wishes; standard file system facilities can be used for all of these functions.

For performance reasons, it may be desirable to have some or all of the bulletin board data replicated on different cluster servers, with bboard servers running to tell users where to find them. This seems like an improvement to be left for later; cluster machine caching may make it unnecessary.

Accounting

There should also be a charge for posting notices on a system bulletin board; it is a good policy question whether the system should charge for posting in a private bulletin board. It seems reasonable to have a cost for each message, a charge based on the size of the message, and a further charge (or perhaps a scaling factor) based on the requested expiration date.

Maintenance

There will have to be a program for the owner of a bulletin board to use to trim messages, whether arbitrarily selected or based on expiration date. This would have to be suitable for system and private bulletin boards alike.

PRINTING

The job of printing files consists of queueing the files (and additional information, such as font selection) to be printed, and printing the queued files. Each printer will be directly controlled by a system server written for it, running in a cluster machine; it will also have a directory, perhaps called "/spool/printer4", in the file system, with the custodian for that directory being the machine in which the printer server runs. The protection of the spool directory should allow any user to create new files in it; if this is not possible, there will have to be a print spooling server.

To queue a file to be printed, the VIRTUE print software determines the printer the user wants to use, performs a *ConnectFS* to its cluster machine's file system if necessary, and stores a disk file in the proper spool directory. The file will contain both control information, specific to the printer and in a format known only to the VIRTUE print program and the printer server, and the data to be printed.

The printer server will continually examine its spool directory, and if it finds anything it will print it in its own way, according to the facilities of the printer it is programmed to use. It can use any information provided to it in the file "header" to control the print. It should bill for its services. It may want to send back a notification to the user when the print is complete.

Note that this is simply an outline for printer spooling functions; design for specific printer servers will depend on there being printers to serve.

--- END ---